VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

R-medžių analizė, taikant juos duomenų gavybos algoritmams

Analysis of R-trees for Data Mining Algorithms

Mokslo tiriamasis darbas. 4 semestras.

Atliko: Laimonas Judeikis (parašas)

Darbo vadovas: Antanas Žilinskas (parašas)

Recenzentas: Julius Andrikonis (parašas)

Vilnius – 2012

Turinys

Įvadas

Duomenys, aprašomi bent keliais atributais (dimensijomis), turintys daug sudėtingesnę struktūrą ir sudėtingesnes duomenų operacijas nei vienmačiai duomenys, yra vadinami daugiamačiais. Su šiais daugiamačiais duomenimis veiksmingai dirba geografinio tipo programos, pavyzdžiui: Geografinės Informacinės Sistemos (angl. Geographic Information System - GIS), kompiuterinės grafikos ir kompiuterinio konstravimo sistemos (angl. Computer-Aided Design - CAD), kurios duomenis naudoja vaizdų analizei, daugiaterpėse duomenų bazėse (angl. Multimedia Databases) ir labai didelėse duomenų bazėse (angl. Very Large Databases - VLDB). Šių programų duomenų bazėse laikoma milijonai ar daugiau duomenų įrašų, optimizuotas indeksavimas yra nepaprastai svarbus dalykas. Sistemoms reikia indeksavimo mechanizmo, kad jis padėtų duomenų bazei gauti duomenų segmentus (elementus) greitai ir efektyviai. Tačiau tradiciniai indeksavimo metodai nėra gerai pritaikyti duomenų objektams nenulinio dydžio daugiamatėse erdvėse paieškai. Todėl būtina sumažinti prieigos prie disko dažnumą sujungiant duomenų objektus, kurie yra arti vienas kito duomenų atžvilgiu.

Dauguma duomenų bazių valdymo sistemų turi efektyviai apdoroti daugiamačius duomenis, pvz., duomenis aprašančius stačiakampius, daugiakampius, arba tiesiog taškus daugiamatėje erdvėje.

Galima išskirti tokias daugiamačių duomenų savybes:

- daugiamačiai duomenys yra sudėtingos struktūros, t.y., duomenys susideda iš daugelio tūkstančių ar milijonų taškų ar objektų, kurie yra įvairiai pasiskirstę erdvėje;
- daugiamačių duomenų erdvinės operacijos (pvz., dviejų taškų, esančių trimatėje erdvėje, sąjunga ar sankirta) yra daug sudėtingesnės bei reikalaujančios daugiau kompiuterio laiko, negu standartinės RDBVS operacijos;
- daugiamačių duomenų bazės yra didelės - paprastai yra operuojama gigabaitiniais atminties tūriais;
- daugiamačiai duomenys neturi nusistovėjusių standartinių algebrinių operacijų - manipuliacijos su duomenimis labai priklauso nuo dalykinės srities.

Tokiems duomenims duomenų bazių valdymo sistemose vietoje operacijų, pritaikomų vienmačiams duomenims, tokiems kaip skaičiai, simbolių eilutės, žodžiai ar kiti pilnai sutvarkomi objektai, reikia įdiegti operacijas, pritaikomas daugiamačiams duomenims. Tačiau algoritmai ir operacijos šiais atvejais yra sudėtingesni ir gali žymiai skirtis nuo klasikinių. Tokių duomenų indeksavimui yra naudojami R-medžiai. Įterpimo algoritmas šio medžio tipui

nepasinaudoja faktu, kad visi duomenų segmentai (elementai) yra jau žinomi. Tai reiškia, kad visi duomenys įterpimo metu nepriklauso vienas nuo kito, nors erdvėje išsidėstę šalia vienas kito.

Šiame darbe bus kalbama apie R-medžius bei K-means ir CURE klasterizavimo algoritmus. Susipažįstama su R-medžių struktūra, aprašomi privalumai. Pateikiami pagrindiniai algoritmai, tokie kaip įterpimas, trynimas, paieška, atnaujinimas, mazgo dalinimas. Prie kiekvieno algoritmo pateikiami jų realizacijų pseudo kodai. Iš pateiktos informacijos galima suprogramuoti savo R-medį.

Darbe taip pat rašoma apie K-means ir CURE klasterizavimo algoritmus. Pateikiami šių algoritmų aprašymai, privalumai ir pseudo kodai. Šios informacijos pilnai užtenka kad pavyktų suprogramuoti savo klasterizavimo algoritmą.

Visus šiuos algoritmus norima pritaikyti elektroencefalogamoms, todėl darbe plačiau susipažįstama su elektroencefalografija bei smegenų bangomis. Aprašyta kaip programos pagalba analizuojamos elektroencefalogramos ir atliekamas gautų duomenų parametrizavimas. Trumpai aprašytas duomenų gavybos algoritmų pritaikymas EEG duomenims.

Tikslai ir uždaviniai

Šiame darbe siekiama išanalizuoti R-medžius [Gut84], K-means [Mac67] ir CURE [GRS97] klasterizavimo metodus. Norima analizuoti R-medžių tinkamumą ir efektyvumą klasterizavimo uždaviniams vykdyti, kai apdorojami duomenys yra daugiamačiai.

Modeliavimas bus atliekamas su medicininiais duomenimis iš elektrofiziologijos srities, t.y., elektroencefalogramos (EEG). Šie EEG duomenys yra daugiamačiai duomenys, jie bus analizuojami ir išskaičiuojami atributai. Paskaičiuoti atributai - tai įvairūs skaitiniai duomenys, turintys reikšmę medikams (KDS indeksas, KSS indeksas, amplitudės maksimumas, pikų forma, pikų dažnis, alfa bangų dažnis, beta bangų dažnis, teta bangų dažnis, delta bangų dažnis, kt.). Kiekvienas duomenų fragmentas kartu su savo atributais sudaro daugiamačius duomenis. Klasterizavimo tikslas - surasti skaitinius ryšius tarp atributų. Šių segmentų bus milijonai. Taikant jiems klasterizavimo algoritmus reikia greitai išrinkti duomenis ir jiems paskaičiuoti egzistuojančius sąryšius tarp atributų. Greitam duomenų išrinkimui bus naudojamas R-medis kaip indeksas.

Atliekami įvairūs veiksmai su dideliais duomenų kiekiais, todėl duomenų analizės (angl. Data Mining) metodai yra reikalingi. Kadangi su šiais duomenimis dirba medikai, taigi greitas duomenų apdorojimas bei analizavimas yra labai svarbus, nes norima greitai gauti rezultatus.

Darbo tikslams pasiekti išskiriami tokie uždaviniai:

- R-medžių sudarymo analizė daugiamačiams duomenims;
- K-means klasterizavimo algoritmo analizė;
- CURE klasterizavimo algoritmo analizė;
- EEG analizė ir parametrizavimas;
- Išrinktų duomenų, pasinaudojus R-medžių indeksavimo metodą, klasterizavimas K-means ir CURE algoritmais;
- Algoritmų testavimas ir įvertinimas.

Norint įvykdyti užsibrėžtus darbo tikslus bus suprogramuotas R-medis, K-means ir CURE algoritmai. Programa bus rašomi nuo pradžių pasinaudojant žemiau šiame darbe pateiktais algoritmų aprašymais. Taip parašytą programą bus lengviau pritaikyti iškeltiems tikslams ir poreikiams.

Praktinio darbo eiga:

1. Parametrizuotų EEG duomenų nuskaitymas;
2. Nuskaitytų EEG duomenų išsaugojimas;
    a. Saugoma R-medyje.
    b. Saugoma į atsitiktinį failą.
3. Skaitomi išsaugoti duomenys;
4. Duomenų klasterizavimas K-means ir CURE algoritmais;
5. Suklasterizuotų duomenų išsaugojimas;
    a. Saugoma R-medžiuose.
    b. Saugoma į failus.
6. Gautų rezultatų palyginimas.

Darbo eigoje numatoma palyginti R-medžių privalumus prieš paprastą saugojimą į failus. Taip pat išsiaiškinti, kuriuo atveju bus įvykdyti visi veiksmai greičiau, efektyviau.

1. Duomenų gavyba (Data Mining)

Kompiuterizacijos paplitimas ir duomenų rinkimo pažanga suteikė galimybę rinkti didžiulius kiekius duomenų. Duomenų generavimas, sukūrimas bei jų rinkimas auga labai greitai. Milijonai duomenų bazių naudojamos verslo valdymui, valdžios institucijų, mokslinių ir inžinerinių duomenų valdymui bei taip pat daugelyje kitų sistemų. Yra pastebėta, kad toks duomenų bazių augimas susijęs su tuo jog šiuo metu yra galingų, efektyvių ir įperkamų duomenų bazių sistemų. Šis augimas sukuria poreikį naujų efektyvesnių technologijų bei įrankių, kurie galėtų automatiškai bei intuityviai apdoroti duomenis ir juos pateikti informatyviai. Būtent dėl šitų priežasčių duomenų gavyba (angl. Data Mining) tapo atskira mokslinių tyrimų sritimi.

Duomenų gavybos taikomosioms programoms reikia panašumų kad būtų galima įvykdyti viršūnių sąjungas *d*-dimensinėse erdvėse. Indeksas sumažina viršūnių skaičių, kurias reikia apdoroti norint atlikti tam tikras operacijas.

Darbas su skirtingų tipų duomenimis.

Kadangi yra daug skirtingo tipo duomenų bei duomenų bazių naudojančių skirtingas taikomąsias programas, yra tikimasi kad sistemos turi efektyviai apdoroti duomenų gavybos algoritmus su skirtingais duomenų tipais. Ne visos sistemos gali apdoroti skirtingo tipo duomenis, tačiau kai kurios sugeba apdoroti kompleksinio tipo duomenis, tokius kaip struktūrizuoti duomenys, kompleksiniai duomenų objektai, hiperteksto ir multimedijos duomenis ir pan. Tačiau tokiems duomenų tipų kiekiams apdoroti reikia skirtingų metodikų ir tai apsunkina duomenų gavybos sistemos darbą su duomenimis. Taigi norint pasiekti gerų rezultatų duomenų analizėje reikia pasirinkti tam skirtas duomenų gavybos sistemas, kurios koncentruojasi į vieną duomenų tipą.

Efektyvumas duomenų gavybos algoritmuose.

Reikia efektyvių ir greitų algoritmų, kad būtų efektyviai išrinkta informacija iš didelio kiekio duomenų. Veikimo laikas duomenų gavybos algoritmams turi būti nuspėjamas ir priimtinas didelėms duomenų bazėms.

Labai svarbu, kad indeksuoti duomenys būtų lengvai prieinami ir vykdant paiešką jų nereikėtų iškoduoti, nes tai stabdo procesą. Plačiai paplitusi ir naudojama technologija, kurios indekso kodas yra priimtinas ir jo nereikia iškoduoti, yra R-medis. R-medžiai sukurti B medžio principu [Juo07].

R-medis yra balansuotas, visi medžio lapai yra viename lygyje, ir atvaizduojamas minimaliais gaubiančiaisiais stačiakampiais. Iš tiesų duomenys yra laikomi žemiausio lygio stačiakampiuose. R-medžio indeksas naudojamas rasti visus susijusius stačiakampius, kurie identifikuoja ieškomą informaciją. Plačiau apie R-medį aprašyta „R-medžiai“ skyrelyje.

2. Paplitusios duomenų indeksavimo struktūros

Turint daug duomenų visų pirma reikia juos kažkur padėti, bet reikia sugalvoti gerą būdą, kaip ir kur tai padaryti, nes nuo to priklauso sugaištas laikas duomenų paieškai. Taigi buvo sugalvotos duomenų indeksavimo struktūros. Jų yra įvairių tipų. Kiekviena skirta kažkokiam tikslui įgyvendinti. Pavyzdžiui, koduotiems simboliams laikyti ir medžio pagalba juos vėl atkoduoti, laikyti nuorodoms (rodyklėms) į duomenų blokus, kuriuos būtų galima greitai surasti pasitelkiant duomenų indeksavimo struktūrą, archyvuoti IP adresams ir t.t. Taipogi jų yra ne tik įvairių paskirčių, bet ir įvairių pavidalų, tokių kaip medžio pavidalo ar lentelės. [Ben75]

Žinomiausios duomenų paieškos struktūros yra medžių pavidalo, tai būtų AVL, 2-3-4, Raudonas-Juodas, B medžiai [Juo07]. Visos šios struktūros yra hierarchinės ir taip pat balansuotos. Balansuotos reiškia, kad visi duomenis turintys taškai yra viename lygyje. Kad būtų tenkinama ši sąlyga, AVL medis atlieka posūkius (viengubas kairėn, dvigubas kairėn, viengubas dešinėn, dvigubas dešinėn) priklausomai nuo esamos situacijos. Raudonas-Juodas ir AVL medžiai daugiausiai gali turėti dvi šakas (vaikus), o jei kalbėtume apie tokį medį kaip 2-3-4, jis gali turėti daugiausia keturias šakas (vaikus). Toks šakų (vaikų) skaičius yra optimaliausias šiems medžiams. Vis dėlto fiksuotas šakų (vaikų) skaičius nebuvo labai priimtinas ir buvo sugalvotas B-medis. Jo šakų (vaikų) skaičius nusakomas intervalu $[M/2] < x < M$, kur M yra maksimalus šakų (vaikų) skaičius. Tai reiškia, kad šakų (vaikų) skaičius gali keistis.

Iškeltam tikslui, saugoti daugiamačius duomenis, geriausiai tinka R medžiai. Tai būtų R-medis, R^+-medis, R^*-medis.

2.1. R-medžiai

„R“ raidė frazėje „R-medis“ reiškia „stačiakampis“ (angl. rectangle). R-medis yra atvaizduojamas minimaliais gaubiančiaisiais stačiakampiais (angl. Minimal Bounding Rectangles – MBR), t.y., duomenys suskirstyti hierarchiškai, kur kiekvienos grupės duomenis galima atvaizduoti minimaliu gaubiančiuoju stačiakampiu, kurio viduje yra visi tos grupės elementai. Stačiakampių pagalba paieška vykdoma greičiau. R šeimos medžiai yra panašūs į B-medį. Jie yra taip pat hierarchinės bei balansuotos struktūros. Mazgo dydis nurodomas intervalu nuo $m<=M/2$ iki M. Tai puiki struktūra išsaugoti duomenis apibrėžtus koordinatėmis. Duomenims parenkama geriausia pozicija pagal jų nurodytas koordinates.

R-medžių savybės:

- kiekvieną medžio lapą sudaro nuo *M/2* iki *M* indeksų viršūnių, kur *M* lygus maksimaliam lapo viršūnių skaičiui;
- kiekvienas indekso mazgas medžio lape yra mažiausias stačiakampis, kuris apima juo nurodomą duomenų objektą;
- medžio šaknis turi bent dvi šakas (vaikus);
- visi medžio lapai yra vienodame gylyje;
- medžio, sudaryto iš *N* viršūnių, aukštis = $|\log_m N| - 1$.

R-medis dažniausiai realizuojamas kaip atmintyje laikoma nuorodų struktūra, kuri dinamiškai susiejama su daugiamatėmis/erdvinėmis sritimis ar duomenų aibėmis, saugomomis antrinėje (arba tretinėje) atmintyje. Šis medis yra subalansuotas duomenų indeksavimo medis, lapuose turintis nuorodas į duomenų objektus. Medžio struktūra sukurta taip, kad atliekant paiešką tereikia „apeiti“ nedidelį skaičių medžio mazgų.

Po R-medžio pirmųjų pristatymų [Gut84] ir šios technologijos nuodugnaus svarstymo [RL85] bei įvairiarūšių eksperimentų buvo pasiūlyta daug variantų ir modifikacijų. Vienas iš tokių variantų – R+-medis [SRF87]. Jis yra labai panašus savo struktūra į R-medį. Tačiau R+-medis skiriasi tuo, kad išvengia persidengiančių sričių. Kadangi elementai vidinėse viršūnėse nepersidengia, paieškos metu yra einama tik viena šaka. Tai žymiai padidina darbo su medžiu efektyvumą, tiek laiko atžvilgiu, tiek atminties sunaudojimo. Bet įterpimo metu reikia tikrinti daugelį šakų, nes naujoji reikšmė turi būti patikrinta, kad nebūtų dviprasmybių paieškos metu.

Beveik tuo pat metu buvo pasiūlytas R*-medis [BKS+90]. R*-medžio sąvoka buvo pasiūlyta siekiant geresnio įterpimo/dalinimo operacijų rezultato greičio atžvilgiu. R*-medžio struktūra labai panaši į R-medį, nes jų abiejų stačiakampiai gali kirstis tarpusavyje. Tačiau R^*-medis labai skiriasi savo skaičiavimo algoritmais. Jis efektyviau palaiko tiek taškinius, tiek erdvinius duomenis. Algoritmų pagalba mažinamas viršūnės įterpimo/dalinimo laikas. Realizavimo, sukūrimo sąnaudos tik šiek tiek didesnės nei kitų R-medžių.

Platesniam nagrinėjimui ir analizavimui pasirinkta hierarchinė duomenų paieškos struktūra R-medis. R-medis pasirinktas dėl to, kad tai yra R medžių pagrindas. Jo struktūra paremti kiti R medžių šeimos nariai.

3. R-medis

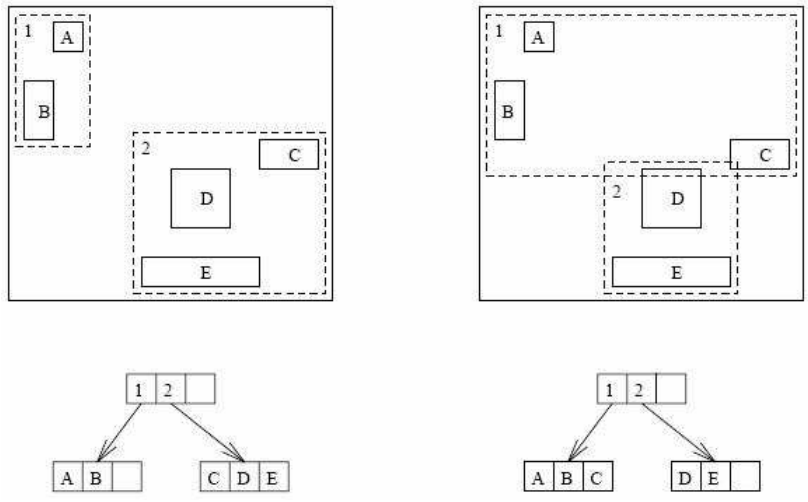
3.1. Indeksavimas bei struktūra

R-medis yra visiškai balansuotas medis labai panašus į B-medį, bet yra naudojamas erdviniu būdu t.y., daugiamačių duomenų indeksavimui, pavyzdžiui, (X, Y) koordinačių geografinius duomenis. „R“ raidė frazėje „R-medis“ reiškia „stačiakampis“ (angl. rectangle). R-medis yra minimalių gaubiančiųjų stačiakampių pavidalo (angl. Minimal Bounding Rectangles – MBR). Jo lapo mazgas susideda iš viršūnių ir rodyklių į duomenų objektus. Imant tokį sulyginimą, kad viršūnės atitinka diske puslapius ir struktūra yra suprojektuota erdvinei paieškai, tai tuomet norint surasti viršūnę reikia aplankyti mažai tarpinių mazgų. Indeksas yra dinamiškas. Įterpimo ir šalinimo algoritmus galima sugretinti su paieškos algoritmu [Gut84].

Erdvinė duomenų bazė susideda iš tam tikro kiekio viršūnių, kurios aprašo objektą. Kiekviena viršūnė turi unikalų identifikatorių, kuris naudojamas surasti lapo viršūnę R-medyje. Indeksas susideda iš laukų (*I, tuple-id*), kur „*tuple-id*“ vienareikšmiškai apibrėžia viršūnės vietą duomenų bazėje, o „*I*“ apibrėžia gaubiantįjį stačiakampį, kuris yra erdvinis objektas $I=(I_0, I_1, ..., I_{n-1})$. Čia n yra dimensijų skaičius ir I_i yra stačiakampio vienos dimensijos intervalas [a, b]. Viršūnės, kurios priklauso ne lapo mazgui, turi štai tokius laukus (*I, child-pointer*), kur „*child-pointer*“ laiko nuorodą į žemesnio lygio šaką ar lapą medyje, o *I* yra mažiausias galimas gaubiantysis stačiakampis, į kurį įeina visos žemesnio lygio viršūnės [Gun86].

R-medis turi du svarbius parametrus, tai yra (*m, M*). *M* ir *m* reikšmės yra sveikieji skaičiai. *M* yra maksimalus mazgo dydis, t.y., kiek mazgas gali maksimaliai turėti žemesnio lygio viršūnių. *m* apibrėžia mažiausią mazgo dydžio reikšmę, kuri lygi $m<=M/2$ [Gut84].

Galima išskirti vieną įdomų faktą apie R-medį, turint tokių pačių viršūnių aibę ir kraunant jas į medį viena eilės tvarka bus gautas vienoks R-medis, tačiau jei tas pačias viršūnes krautume kita eilės tvarka, gautume visiškai kitokios struktūros R-medį, bet su tais pačiais duomenimis. Tai reiškia, kad R-medžiai nebūtinai turi būti tokie patys, jei jų turimos viršūnės vienodos. Pavyzdys pateikiamas 1 paveikslėlyje. Šiame pavyzdyje tiek a), tiek b) variantai naudoja R-medį su $M=3$ reikšme. Tačiau medžiai skiriasi, nes duomenų įterpimas buvo atliktas skirtinga eilės tvarka. Todėl gavome su tais pačiais duomenimis suformuotus du skirtingus medžius.

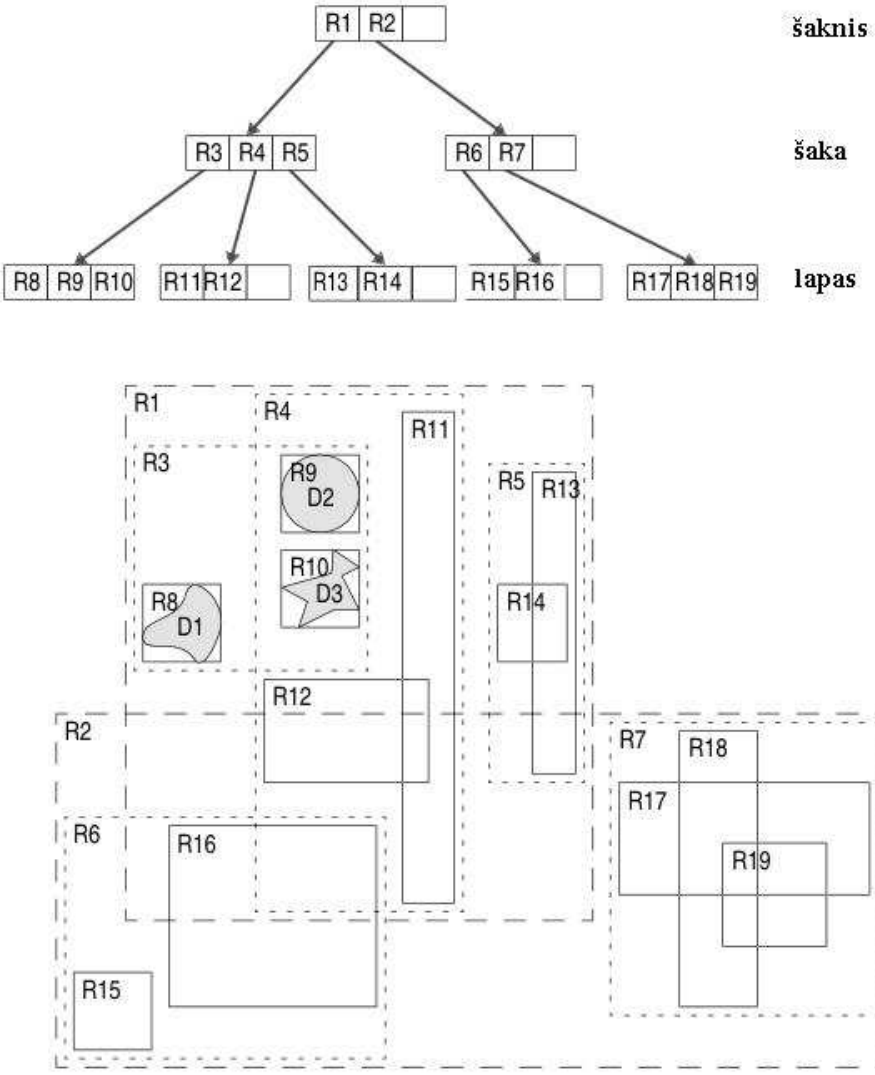


a) b)

Pav. 1: a) ir b). Du skirtingi R-medžiai su tais pačiais duomenimis [Gut84].

R-medis yra tinkamas laikyti antrinėje atmintyje. Kiekviena viršūnė yra padedama atskiroje atminties vietoje. Tai daro šį medį labai naudingą taikomosioms programoms, kurios naudoja labai dideles duomenų saugyklas, kur indeksas yra per didelis, kad tilptų pagrindinėje atmintyje.

Paveikslėlyje 2 rodoma R-medžio struktūra, kuri iliustruoja sujungimus ir perdengimus tarp stačiakampių. Duomenų objektai rodomi tik R8, R9 ir R10 gaubiančiuosiuose stačiakampiuose [HJR97]. Kiti gaubiantieji stačiakampiai lapuose (nuo R11 iki R19) taip pat turės duomenis, bet jų figūros neatvaizduotos tiesiog tam, kad būtų paprastesnis grafikas. Kaip matoma paveikslėlyje, gaubiantysis stačiakampis gali saugoti tik vieną duomenų objektą arba vieną ar daugiau žemesnio lygio stačiakampių (maksimalus stačiakampių skaičius nurodomas *M* dydžiu). Stačiakampis R8, kuris yra tam pačiam lygyje kaip ir kiti lapai, nes R-medis yra balansuotas medis, turi savyje duomenų objektą D1. Stačiakampis R3, kuris yra medžio šaka, turi savyje maksimaliai 3 viršūnes R8, R9, R10. Stačiakampis R1, kuris yra šaknis (šaknis medyje būna tiktai viena pirmutinė viršūnė), turi nuorodas į R3, R4, R5 stačiakampius. Visa tai matoma 2 paveiksliuke.



Pav. 2: R-medžio struktūra ir gaubiantieji stačiakampiai [Gun86].

R-medis apibrėžiamas tokiais reikalavimais jo struktūrai:

- Kiekvienas lapo mazgas susideda nuo *m* iki *M* indeksų viršūnių, nebent tai būtų šakninis medžio mazgas;
- Kiekvienam mazgo (*I*, *tuple-id*) lapui, *I* yra mažiausias stačiakampis, kuris galimas su tais n-mačiais duomenų objektais;
- Kiekvienas ne lapo (šakos) mazgas susideda nuo *m* iki *M* žemesnio lygio viršūnių, nebent tai būtų šakninis medžio mazgas;
- Kiekvienam mazgui (*I*, *child-pointer*) ne lapui (šakai), *I* yra mažiausias stačiakampis, kuris galimas iš jame esančių žemesnio lygio viršūnių;
- Šakninė medžio viršūnė turi turėti bent dvi viršūnes, nebent tai būtų lapas;
- Visi lapai turi būti viename lygyje;

- R-medžio gylio *d* mažiausiai gali būti m^{d+1} indeksų objektų ir daugiausiai M^{d+1} indeksų objektų, kitais žodžiais tariant [$\log_M$ N-1] <= *d* <= [$\log_m$ N-1].

3.2. Reakcijos laikas

Ieškomas mažiausias reakcijos laikas, kur laikas yra matuojamas nuo tada, kai vartotojas įves sąlygą iki atsakymas bus grąžintas atgal vartotojui. R-medis turi indeksus, kurių pagalba sumažinamas kreipimusi (I/O operacijų) kiekis į diską, tai šioje situacijoje reakcijos kreipimosi į diską greitis nėra labai svarbus. Reakcijos laikas R-medyje yra CPU (procesoriaus) sugaištas laikas, kuris apibrėžiamas kaip reakcijos laikas tipinei užklausai. Jei R-medyje visos viršūnės yra laikomos skirtinguose objektuose, tai tuomet atsako laiku būtų galima atitinkamai laikyti nuskaitytų viršūnių skaičių per užklausą [KF92].

Sakykime, kad R-medis yra vieno disko sistemoje. Naudojant vieno disko modelį, R-medžio reakcijos laiko konfigūracijai, yra galima tikėtis reakcijos laiko $E[T_r]$ kokiai nors paprastai užklausai. Pavyzdžiui, imkime S_j, kuris bus duomenų apimtis *j*-joje dimensinėje erdvėje, q_j bus lango sekos apimtis toje dimensijoje ir r^i_j bus *i* šakų stačiakampių R-medyje *j*-joje dimensinėje erdvėje apimtis. Šioje formulėje *N* apibrėžia šakų skaičių medyje, o *D* dimensijų kiekį [KF93]. Reakcijos laikas R-medyje yra apskaičiuojamas:

$$E[T_r] = (\sum_{i=1}^{N} \prod_{j=1}^{D} (r^i_j + q_j)) / \prod_{j=1}^{D} S_j$$

Dviejų dimensijų R-medis turi keletą kriterijų, į kuriuos reiktų atkreipti dėmesį skaičiuojant ar nuspėjant reakcijos laiką:

1) Stačiakampių užimamą plotą;
2) Stačiakampių perimetrą;
3) Persidengimus tarp stačiakampių;
4) Saugojimo panaudojamumą.

Visi kriterijai yra tarpusavyje susiję. Jei stačiakampių plotas yra palyginus mažas su perimetru, tuomet sakoma, kad yra daug „mirusios“ vietos. Tai reiškia, kad stačiakampiai yra išsibarstę erdvėje ir bendras jų surišimas apima daug suindeksuotos, bet nepanaudotos vietos, tokiais atvejais neišnaudojamos visos saugojimo galimybės. Kitas pastebėjimas tai, kad jei persidengimų tarp stačiakampių yra daug, tokiu atveju sulėtėja paieška medyje. Jei sumažintume

persidengimų skaičių ir stačiakampių perimetrą, tai saugojimas būtų geriau išnaudotas. Tai yra labai naudinga jei norima sumažinti nuskaitymo iš disko kiekį.

3.3. Viršūnės paieška

Paieškos algoritmas eina medžiu žemyn nuo šaknies iki lapo. Taigi leidžiantis žemyn bus aplankoma daug pereinamųjų viršūnių, kurios turi būti išanalizuojamos. Paieškos algoritmas turi būti padarytas taip, kad nereikalingi duomenys nebūtų be reikalo imami iš disko ir analizuojami. Algoritmas praleidžia ir nepaima tų viršūnių, kurios nėra susijusios su ieškomu objektu, o eina gilyn link ieškomo objekto pozicijos.

Jei ieškoma viršūnė nepersidengia per kelis gaubiančiuosius stačiakampius, tai sudėtingumas yra $O(\log_m N)$. Žinoma, jei viršūnė persidengia, tuomet paieška tampa sudėtingesnė, ir jau ji yra nebe logaritminė. Blogiausiu atveju, kai viršūnė įeina į visus gaubiančiuosius stačiakampius, tai tuomet sudėtingumas tampa $O(N)$. Aišku, blogiausias atvejis bus tik tuomet, kai ieškoma viršūnė bus tikrinama paskutinė medyje [RL85].

Pateiksiu paieškos algoritmo pseudo kodą.

Paieška algoritmas. Visų pirma duotąją R-medžio šaknį pasižymime *T*. Tuomet surandame visas mazgo *T* viršūnes, kurios perdengia ieškomą stačiakampį *S*.

[kai ieškome šakoje] Jei *T* nėra lapas, tai tuomet surandame kiekvieną viršūnę, kuri perdengia ieškomą stačiakampį *S*. Kiekvienai rastai viršūnei pritaikome paieškos algoritmą, tik jau *T* bus rodyklė į surastą stačiakampį.

[kai ieškome lape] Jei *T* yra lapas, tai tuomet patikriname visus jo viršūnes tam, kad nustatytume ar ieškomas stačiakampis yra surastas, ar jis egzistuoja mūsų struktūroje.

Jei stačiakampis *S* rastas, tai tuomet nutraukiama tolimesnė duomenų paieška (jei tokia dar butų vykdoma). Jei vis dėlto toje viršūnėje jo nebuvo, tai rekursijos būdu grįžtame žingsniu atgal ir einame į sekančią persidengiančią viršūnę. Jei visos persidengimo viršūnės buvo apeitos ir nebuvo rasta tinkama - vadinasi ieškomo objekto R-medyje nėra.

3.4. Viršūnės atnaujinimas

Norint atnaujinti viršūnės informaciją visų pirma reikia ją surasti. Paieška vykdoma taip, kaip nurodyta „Viršūnės paieška“ skyrelyje. Kai jau viršūnė su nuoroda į objektą randama, tai tuomet galima atlikti objekto atnaujinimą.

Jei surastame duomenų objekte atliekame esminius pakeitimus, atnaujinimus, tai vadinasi turėtų keistis ir duomenys gaubiančiuosiuose stačiakampiuose. Tam, kad būtų išvengta nesuderinamumų, viršūnė turi būti ištrinta ir jos rodyklė iš naujo įdėta į medį, kad ji surastų sau tinkamą vietą medyje.

3.5. Viršūnės įterpimas

Įdedant naują viršūnę į medį svarbiausia jai parinkti tinkamą vietą medyje. Viskas priklauso nuo gerai parašyto rekursinio algoritmo. Medžiu einama nuo šaknies iki lapo, kuriame ir bus patalpinta informacija apie naują viršūnę su nuoroda į duomenų objektą. Einant gilyn reikia analizuoti duomenis ir parinkti geriausią kelią gilyn iki lapo. Kai prieinamas lapas, reikia patikrinti, ar jis nėra pilnas. Prisiminkime, kad mazgo dydis nurodomas *M* kintamuoju, tai vadinasi, jei jau egzistuoja *M* viršūnių su nuorodomis į duomenų objektus, mazgas yra pilnas ir reikia atlikti mazgo dalinimo algoritmą. Apie mazgo dalinimą yra aprašyta kitame poskyryje. Viskas žymiai paprasčiau, jei mazgas nėra pilnas, tiesiog įdedame viršūnę su jai priklausančia nuoroda prie kitų viršūnių [HS93].

Įterpimas algoritmas. Naujos viršūnės *E*, su nuoroda į duomenų objektą, įterpimas į R-medį.

[pozicijos radimas naujai viršūnei] Iškviečiam **Lapo Pasirinkimas** algoritmą, kad būtų išrinkta lapo *L* pozicija, į kur bus įdėta viršūnė *E*.

[viršūnės įterpimas į lapą] Jei *L* turi vietos dar vienai viršūnei išsaugoti tai tuomet viršūnę *E* tiesiog įdedame prie kitų *L* lape esamų viršūnių. Jeigu susidurta su tokia situacija, kad lapas vis dėlto pilnas, reikia atlikti **Mazgo Dalinimas** algoritmą. Tuomet turėsite lapus *L* ir *LL*, kurie turės viršūnę *E* ir taip pat visas prieš tai buvusias viršūnes.

[medžio atnaujinimas lipant aukštyn] Iškviečiamas **Medžio Derinimas** lapui *L*, jei medis buvo dalinamas tai algoritmą taip pat reikia iškviesti ir *LL* lapui.

[medžio lygio augimas] Jei mazgo dalinimas išpropagavo šaknies dalinimą, sukuriama nauja šaknis. Nauja šaknis turės dvi viršūnes, tai yra padalinta senoji šaknis. Tokiu būdu medžio lygis padidėja vienetu.

Lapo Pasirinkimas algoritmas. Pasirenkama lapo viršūnė, kur bus įdėta nauja viršūnė *E*.

[pasiruošimas] Nustatome *N* rodyklę į šaknies viršūnę.

[patikrinimas ar lapas] Jei *N* yra lapas, tuomet gražiname *N* reikšmę.

[pasirenkama tolimesnė šaka] Pasirenkant sekančią šaką reikia apeiti visas viršūnes esančias *N* mazge ir patikrinti, kurios iš viršūnių padidėtų mažiausiai, t.y., jų stačiakampių plotas pakistų mažiausiai arba nepakistų iš viso, kai mes pridėsime naują viršūnę *E*. Jei daugiau nei viena viršūnė atitinka šį reikalavimą tuomet pasirenkamas mažiausią plotą turintis stačiakampis.

[leidžiamės iki kol pasieksim lapą] Nustatome *N* rodyklę į išsirinktą šaką. Vėl atliekamas punktas [patikrinimas ar lapas].

Medžio Derinimas algoritmas. Kylame nuo lapo *L* iki šaknies ir deriname bei reguliuojame stačiakampius kiekvienoje viršūnėje. Jei būtina atliekame **Mazgo Dalinimą**.

[pasiruošimas] Nustatome rodyklę *N* į *L* lapą, kuris buvo padalintas anksčiau. Pasirašome rodyklę *NN*, kad būtų mūsų antrinė rezultato viršūnė, kurią gauname padalinę viršūnę.

[patikriname ar baigta] Jei *N* yra šakninė viršūnė, tai tuomet sustojame.

[atnaujiname persidengiančius stačiakampius tėvo viršūnėje] Tegul *P* bus tėvo viršūnė *N* šakoje. Paimkime E_N visas *N* mazgo viršūnes tėve *P*. Suderiname visus E_N, kad jie atspindėtų žemiau esamą informaciją.

[skleidžiame mazgo dalinimą aukštyn] Jei *N* turi brolį *NN*, kuris atsirado tuomet kai buvo įvykdytas dalinimas. Paimame E_{NN} visas *NN* mazgo viršūnes. Suderiname visas E_{NN} viršūnes ir apibrėžiame kaip stačiakampį *NN*. Pridedame *NN* prie *P*, jei jame vis dar yra vietos. Jei vietos nebėra, kviečiame algoritmą **Mazgo Dalinimas**, kurį atliksime viršūnei *P*, o *PP* susidės iš *NN* ir visų *P* buvusių viršūnių.

[lipame aukštyn] Nustatome *N* rodyklę į *P*. Jei buvo atliktas dalinimas, tai nustatome *NN* rodyklę į *PP*. Sekančiu žingsniu vėl kartojame žingsnį [patikriname ar baigta].

3.6. Viršūnės trynimas

Viršūnės ištynimas iš R-medžio susideda iš kelių dalių. Visų pirma reikia surasti norimą ištrinti viršūnę, žinoma, jei tokia egzistuoja esamame medyje. Kai jau viršūnė rasta, ji yra pašalinama iš medžio. Pašalinus viršūnę reikia patikrinti, ar mazgo dydis nėra mažesnis nei numatytoji reikšmė *m*, kuri apibrėžia minimalų mazgo dydį. Jei mazgo dydis yra lygus ar daugiau už *m*, tai tuomet tereikia sutvarkyti visų šakų turimą informaciją ir trynimo algoritmas yra baigtas. Vis dėlto, jeigu mazgo dydis yra mažesnis nei *m*, tai tuomet reikia sujungti kai kuriuos du mazgus ar pasidalinti jų viršūnėmis bei suderinti visas šakas su jų turima informacija [HS93].

Trynimas algoritmas. Trinama viršūnė *E* iš R-medžio.

[surandame viršūnę turintį mazgą] Paleidžiame algoritmą **Surasti Lapą**, tam kad būtų aptiktas lapas *L*, kuris turi *E* viršūnę. Sustabdome paiešką jei viršūnė nebuvo rasta.

[triname viršūnę] Ištriname *E* iš *L* lapo.

[vykdyti pakeitimus] Paleidžiame algoritmą **Tvarkyti Medį** nuo *L* lapo.

[medžio lygių sumažėjimas] Jei šaknis turi tiktai vieną viršūnę po to, kai medis buvo sutvarkytas, tai tuomet tą vieną viršūnę reikia padaryti nauja šaknimi, o senosios jau nebelieka.

Surasti Lapą algoritmas. Duotame R-medyje šakninę viršūnę pasižymime *T*. Surandame lapą, kuriame būtų *E* viršūnė.

[ieškojimas šakoje] Jei *T* nėra lapas tuomet patikriname kiekvieną mazgo *F* viršūnę *T* šakoje, kad būtų nustatyta ar *F* viršūnė perdengia ieškomą *E* viršūnę. Taigi kiekvienai *F* viršūnei iškviečiame **Surasti Lapą** algoritmą medžiui, kurio šakninis mazgas bus *F* viršūnė. Tai atliekame tol kol *E* viršūnė bus surasta arba visos viršūnės bus apeitos ir patikrintos.

[ieškojimas lape] Jei *T* yra lapas tai tuomet tikriname visas jo viršūnes ar ten yra *E*. Jei *E* yra rastas tuomet gražiname *T*.

Tvarkyti Medį algoritmas. Duotas lapas *L*, kuriame viršūnė buvo ištrinta, ir jis yra eliminuojamas, jei turi per mažai viršūnių, ir perskaičiuojama jo buvimo vieta. Eliminuojame tiek viršūnių, kiek yra būtina lipant aukštyn šakomis. Suderinami visi gaubiantieji stačiakampiai

lipant medžiu aukštyn iki šaknies. Stačiakampių plotas yra mažinamas, kad neliktų perteklinio ploto.

[pasiruošimas] Nustatome rodyklę N į L lapą. Apsirašome kintamąjį Q, kuris bus eliminuotų viršūnių grupė, ir jį padarome tuščia.

[surandame tėvo viršūnę] Jei N yra šaknis, tai eiti į žingsnį [perdėti likusias viršūnes]. Kitu atveju P pažymėsime tėvą, kuris turi N rodyklę. Paimame E_N, kur jis yra visos N mazgo viršūnės, ir pridedame prie P.

[eliminuojame nepilną viršūnę] Jei N turi mažiau nei m viršūnių, tai ištiname E_N iš P ir įdedame visą N į Q kintamąjį.

[suderiname gaubiančiuosius stačiakampius] Jei N yra neeliminuotas, tai tuomet suderiname visas E_N viršūnes.

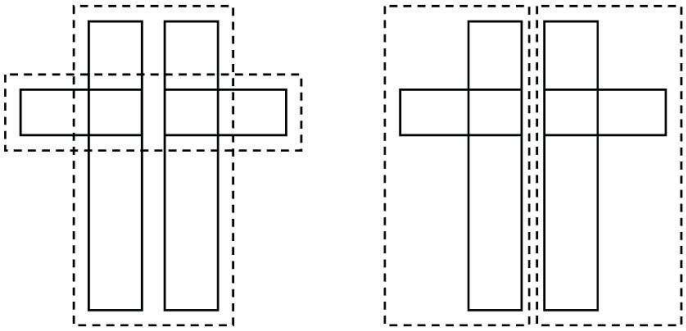
[pakilimas aukštyn vienu lygmeniu medyje] Nustatome N į P ir kartojame žingsnius nuo [surandame tėvo viršūnę] žingsnio.

[perdėti likusias viršūnes] Iš naujo įdedame viršūnes esančias mūsų apsibrėžtame kintamajame Q. Viršūnės įdedamos į medį taip kaip nurodyta **Įterpimas** algoritme vienas po kito.

Pasirinktas pakartotinis viršūnių įterpimas dėl dviejų priežasčių. Pirma, viršūnė, kurią reikia įdėti, jau yra atmintyje ir jos dar kartą kurti nereikia. Antroji priežastis yra tai, kad naujai įdedama viršūnė medyje naujai apibrėžia erdvės struktūrą ir trukdo laipsniškam medžio blogėjimui, kuris gali nutikti, jei kiekviena viršūnė visam laikui būtų pririšta prie to paties tėvinio mazgo.

3.7. Mazgo dalinimas

Norint tvarkingai įdėti viršūnę į jau užpildytą mazgą, kuris susideda iš M viršūnių, būtina dalinti mazgą, $M + 1$ viršūnių, į du mazgus. Dalinimas turi būti unikalus, abu nauji padalinti mazgai turi susidėti iš kruopščiai ištirtų viršūnių aibių, kad jų užimamas plotas būtų kuo mažesnis abiejų mazgų atžvilgiu. Tai vadinasi mes turime paimti tokias viršūnes, su kuriomis sudėjus visus mazgo viršūnių plotus, jis būtų kuo mažesnis, bet taip pat reikia nepamiršti, kad yra ir antras mazgas, kurį taip pat reikia išanalizuoti. Kaip pavyzdį pateikiu paveiksliuką 3, kuriame yra pavaizduotas geras ir blogas dalinimas su turimais tais pačiais duomenimis. Kairioje pusėje pateiktas geras dalinimas, nes jei lyginant su dešiniame šone pateiktu pavyzdžiu, kairysis užima mažiau ploto nei dešinysis.



Geras dalinimas Blogas dalinimas

Pav. 3: Dalinimas [Gun86].

Toks pat būdas buvo naudojamas **Lapo Pasirinkimo** algoritme, kai buvo norima nustatyti, kur geriau įdėti viršūnę einant per medžio šakas. Sekanti šaka buvo pasirenkama ta, kurios stačiakampis mažiausiai padidėjęs.

3.7.1 Pilno perrinkimo algoritmas.

Pats tiesiausias kelias surasti mažiausia plotą tai yra sugeneruoti visus įmanomus sugrupavimo variantus ir pasirinkti geriausia. Taigi tokiu atveju šios sąlygos sudėtingumas yra $O(2^{M+1})$. Toks variantas yra pakankamai geras su mažu mazgo dydžiu, bet kuomet mazgo dydis yra didelis tai šis algoritmas patampa per lėtas [HS93]. Tokiais atvejais galima naudoti kitą algoritmą.

3.7.2 Kvadratinio sudėtingumo algoritmas.

Šis algoritmas stengiasi rasti kuo mažesnio ploto dalinimą, bet nėra garantijų, kad bus rastas tik vienas variantas su mažiausio ploto galimybe. Šio algoritmo sudėtingumas yra kvadratinis atsižvelgiant į mazgo dydį M, t. y. $O((M+1)^2)$. Algoritmas pasirenka dvi iš $M+1$ viršūnių, kurios bus pirmosios naujuose grupėse. Po to imami sekantys elementai ir dedami į tas grupes [HS93].

Kvadratinis Dalinimas algoritmas. Dalinamos $M+1$ mazgo viršūnių į dvi grupes.

[pirmo elemento pasirinkimas kiekvienai grupei] Pritaikome **Pasirinkti Pirmuosius** algoritmą, kad būtų pasirinkti du elementai, kurie bus pirmieji grupėse. Kiekvieną jų priskiriame grupei.

[tikrinimas ar jau viskas atlikta] Jei visi elementai jau priskirti grupėms, tai stabdome algoritmą. Jei grupė turi mažiau elementų, nei m tai jei reikia priskirti tiek elementų iš kitos grupės, kad ji pasiektų tą ribą m.

[elemento pasirinkimas] Iškviečiame **Pasirinkti Kitą** algoritmą, kad būtų pasirinktas sekantis elementas įterpimui. Įdedame elementą į grupę, kurios gaubiantysis stačiakampis bus padidėjęs mažiausiai, kai bus pridėtas elementas. Jei randami keli atitikimai, tai elementą reikia dėti į tą grupę, kuri turi mažiausiai elementų. Toliau kartoti žingsnius nuo [tikrinti ar jau viskas atlikta] viršūnės.

Pasirinkti Pirmuosius algoritmas. Pasirenkami du elementai, kurie bus pirmieji savo grupėje.

[skaičiuojame neveiksmingumą jei sugrupuotume elementus kartu] Kiekvienai porai elementų E_1 ir E_2 sudaromas stačiakampis J į kurį įeina E_1 I ir E_2 I. Paskaičiuojame d = plotas(J) – plotas(E_1 I) – plotas(E_2 I).

[pasirenkame labiausiai išlaidžia porą] Pasirenkame porą, kurios skaičius d yra didžiausias.

Pasirinkti Kitą algoritmas. Pasirenkamas vienas elementas grupė klasifikacijai.

[apskaičiuojama kiekvieno elemento įterpimo į kiekvieną grupę vertė] Kiekvienam elementui E, kuris dar nėra grupėje, paskaičiuojam d_1 = ploto padidėjimas gaubiančiajame stačiakampyje, jei pirmoje grupėje bus pridėtas E I. Apskaičiuojame d_2 taip pat tiktai antrajai grupei.

[pasirenkamas elementas] Pasirenkamas bet kuris elementas, kurio skirtumas tarp d_1 ir d_2 yra didžiausias.

3.7.3 Linijinio sudėtingumo algoritmas.

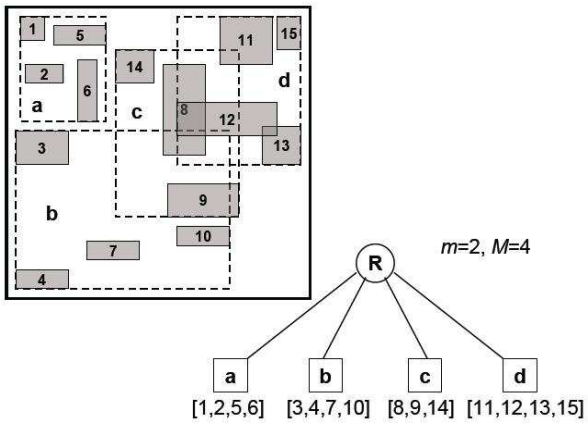
Šis algoritmas yra linijinio sudėtingumo. Linijinis Dalinimas yra labai panašus į Kvadratinį Dalinimą, bet jis naudoja kitokį „Pasirinkti Pirmuosius“ algoritmą. Algoritmas „Pasirinkti Kitą“ yra visiškai paprastas, tiesiog pasirenkamas bet kuris iš galimų elementų [HS93].

Linijinis Pasirinkti Pirmuosius algoritmas. Pasirenkami du elementai, kurie bus pirmieji savo grupėje.

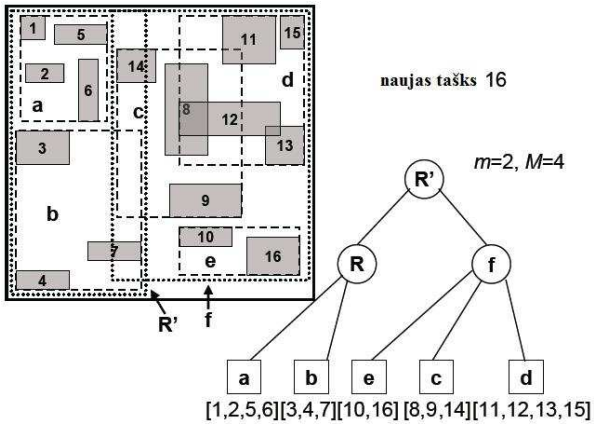
[surandame kraštinius elementus stačiakampyje] Visose dimensijose surandame elementus, kurie stačiakampyje yra viršuje kairėje ir apačioje dešinėje pusėse.

[pasirenkama kraštutinė pora] Iš visų dimensijų pasirenkame vieną porą, kurios atstumas tarp dviejų jos viršūnių didžiausias.

Kai jau pasirenkamas geriausias mazgų sugrupavimas, sukuriami du mazgai, kurie pridedami prie tėvo. Tačiau nutinka ir taip, kad tėvo mazgas irgi yra pilnas ir jį taip pat reikia dalinti. Pateikiu du paveikslėlius 4.1 ir 4.2, kur yra pavaizduota būtent ši situacija, kai tėvą taip pat reikia dalinti. Paveikslėlyje 4.1 tėvas yra taip pat ir šaknis (*R*), maksimalus mazgo dydis *M* yra 4, o minimalus *m* yra 2. Tėvas jau yra užpildytas 4 viršūnėmis (*a*, *b*, *c*, *d*). Mes norime įdėti naują viršūnę 16 į medį. Pagal mūsų paskaičiavimus jam geriausia vieta mazge *b*. Tačiau mazgas *b* jau turi 4 viršūnes, tai imame ir daliname mazgą *b* į du mazgus *b* ir *e*, kur į jų mazgus sudėsime mažiausiai ploto užimančius viršūnių derinius. Tuomet šaknis tampa taip pat su 5 viršūnėmis, o maksimalus viršūnių skaičius yra 4. Tai mes tiesiog paimame ir padaliname šaknį į du mazgus *R* ir *f* lygiai taip pat kaip tai darėme anksčiau, tik šiuo atveju nebėra kam priskirti tų dviejų mazgų. Sukuriama naują šaknį *R*', kuriai galime priskirti tuos du mazgus *R* ir *f*. Tokiu būdu medžio lygis padidėja vienetu, reikia nepamiršti jį priskirti.



Paveiksliukas 4.1. [HS93]



Paveiksliukas 4.2. [HS93]

4. R-medžio realizacija

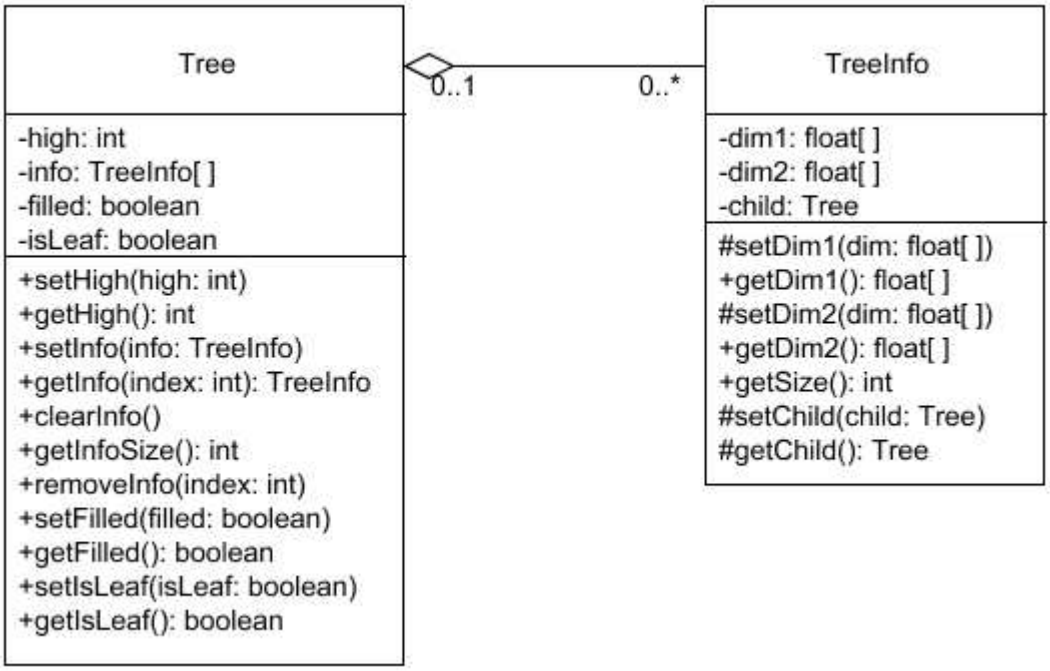
R-medį realizuoti nutariau pats. Taip nusprendžiau todėl, kad savo paties sukurtą medį bus lengviau pritaikyti savo tikslams ir poreikiams. Be to žymiai geriau išanalizuoti ir suprasti R-medį bus galima tuomet, kai pats jį suprogramuosi.

4.1. Projektavimas

Anksčiau pateiktame 3 skyriuje yra pilnai aprašytas R-medis, jo struktūra bei logika, pateikti algoritmų pseudo kodai. Pasinaudojus šio skyriaus pateiktu R-medžio aprašymu galima ne tik jį nuodugniai išsiaiškinti ir suprasti kaip jis veikia, bet ir jį suprogramuoti.

Man gerai žinoma programavimo kalba yra JAVA, todėl nusprendžiau su ja ir programuoti R-medį. Taip pat su šia programavimo kalba medį realizuoti galėsiu objektiniu programavimo būdu. Tokiu būdu sukurtas medis yra lengvai manipuliuojamas bei lengviau jį pakoreguoti ar pataisyti, jei tai yra būtina. Realizuotas klases galima ne sunkiai praplėsti ir sukurti R^+ ar R^* medžius.

Prieš programavimą įsivaizdavau ir pasibraižiau UML diagramą, kaip atrodys programuojamas R-medis, kokia jo struktūra.



Pav. 5: R-medžio UML diagramą.

Nuspręsta kad R-medžiui realizuoti reikės dviejų klasių, kurios atspindi medžio tašką ir to taško įrašus.

Pirmoji bus pagrindinė, dar vadinama šaknimi, šaka ar lapu, ji savyje laikys informaciją ar taškas yra lapas, ar jis užpildytas, koks taško lygis bei svarbiausia ji turės aprašytą mazgą, kuris talpins nuorodas į kitą klasę. Kintamasis, kuris nusako ar taškas yra lapas, naudojamas taško aprašymui ar jis yra lapas, ar tai paskutinis taškas medyje, ar vis dėlto dar galima medžiu eiti gilyn. Reikšmė „filled“ (ar mazgas užpildytas) pažymima tokiu atveju, kai taškas saugo tiek įrašų, koks yra *M* dydis. Kiekvienas taškas turi aukštį, dar vadinamą lygiu. Šis kintamasis nusako, koks taško lygis, lapas yra 0 (nulis) lygyje, o kiekvieno aukštyn einančio taško lygis didėja 1. Aukštis naudojamas medžio balansavimui, jei dviejų šakų aukščiai skiriasi daugiau nei 1, tai atliekami medžio balansavimo veiksmai. Iškarto kyla klausimas, kam reikalingas kintamasis „isLeaf“ (ar taškas yra lapas) jei tai galima nustatyti iš aukščio, nes lapas turi 0 aukštį. Atsisakyti šio kintamojo galima, tačiau medžio reakcijos laikas, nors ir neženkliai, bet padidės. Aukštis yra „integer“ tipo, o „isLeaf“ saugomas „boolean“ tipu („true“ arba „false“), taigi atliekant palyginimą ar aukštis lygu 0 užtrunkama ilgiau nei loginės operacijos atlikimui. Šių palyginimų programoje atliekama labai daug, taigi apsimoka turėti loginio tipo kintamąjį, be to papildomas kintamasis yra žymiai suprantamiau. Iš pirmosios klasės į antrąją bus patenkama per mazgo nuorodas, įrašus. Nuorodų sąrašo saugojimo yra keli būdai, nuorodas galima saugoti įkeliant jas į „Vector“, „ArrayList“ ar „Array“ tipų kintamuosius. Kadangi „Vector“ tipas yra per lėtas šioms operacijoms, o „ArrayList“ užima daugiau vietos nei paprastas masyvas, tai buvo pasirinktas „Array“ tipas. Šis paprastas „Array“ masyvas yra statinio tipo, taigi buvo praplėstos jo galimybės ir papildyta dinaminiu masyvo praplėtimo funkcija. Saugant nuorodas tokiu būdu buvo sutaupyta vietos atmintyje bei paspartintas medžio veikimas, nes taško įdėjimas į šį masyvą greitesnis nei įdėjimas į „ArrayList“ masyvą.

Norint pagrysti teorines žinias apie „Vector“, „ArrayList“, „Array“ masyvus bei sužinoti ar greitaveikos atžvilgiu padėjo pridėti papildomi „isLeaf“ ir „filled“ kintamieji, buvo atlikti testai. Testuose buvo lyginamas R-medžio užpildymo greitis prieš ir po patobulinimo. Plačiau apie atliktus patobulinimus ir testus rašoma 4.2 skyriuje.

Antroji klasė turės informaciją apie stačiakampio dydį bei nuorodą į tašką. Nuorodomis sujungiami taškai su jų informacija bei informacija su žemesnio lygmens taškais. Stačiakampio dydis saugomas „float“ tipo kintamuosiuose, pvz. jei R-medis yra dviejų dimensijų medis, tai išviso bus 4 kintamieji, kurie nusako stačiakampio kairiojo apatinio kampo koordinates (x1, y1) ir viršutinio dešiniojo kampo koordinates (x2, y2).

Žinoma galima viską pasidaryti vienoje klasėje, bet tokiu atveju atsiranda labai daug apibrėžtų ir statinių kintamųjų, kurie padaro medį per daug pririštą ir sunkiai prisitaikantį prie kitokių sąlygų, tokių kaip mazgo dydžio *M* kitimas.

Sekantis dalykas be ko R-medis pilnai negalėtų funkcionuoti, tai keletas pagrindinių algoritmų. Be algoritmų įdėjimas, dalinimas, paieška ir trynimas medžio nebūtų galima sukurti ir juo manipuliuoti. Kiekvienas algoritmas susideda iš tam tikrų žingsnių, kurių negalima praleisti ar sukeisti vietomis. Žinoma sukurtus algoritmus reikia iškviesti, tokiam tikslui sukurta vartotojo sąsają UI (User Interface).

4.2. Atlikti programos patobulinimai

Suprogramuotas R-medis buvo įvertintas. Gauti rezultatai buvo įprasti ir tokių buvo tikėtasi. Tačiau to nepakako ir nuspręsta patobulinti programą. Programą tobulinti ir siekti geresnių rezultatų buvo nuspręsta dėl to kad norimų apdoroti duomenų kiekis yra didelis ir su R-medžiu juos reikia kuo greičiau apdoroti.

Atlikti programos tobulinimo darbai:

1. Įdėtas kintamasis „isLeaf“;
2. Įdėtas kintamasis „filled“;
3. Panaudotas „Array“ tipo nuorodų masyvas;
4. Sugalvotas spartus būdas, kaip greitai suskaičiuoti grupių sekas;
5. Atliktas smulkus kodo optimizavimas.

Atlikti tyrimai parodė, kad saugant duomenis į R-medį programoje atliekama labai daug patikrinimų ar mazgas yra lapas ir ar mazgas yra užpildytas. Taigi nuspręsta apsibrėžti loginės reikšmės (angl. boolean) kintamuosius, kurie turėtų paspartinti programos veikimą.

Atlikus šiuos programos patobulinimo darbus nebuvo tikėtasi labai didelio programos spartos pagerėjimo, tačiau įvykdžius testus ir palyginimus gauti rezultatai nustebino ir pradžiugino. Gautas vidutiniškai 3% programos spartos pagerėjimas. Taigi įvesti papildomi loginiai kintamieji paspartino R-medžio užpildymą, tačiau naudojant daugiau kintamųjų padidėja ir atminties sunaudojimas. Palyginus atminties sunaudojimą prieš ir po pakeitimo gauta, kad atminties sunaudojimas padidėjo 0,72%. Tai tikrai maža kaina lyginant su tokiais spartos pasiekimais.

Elementų sk. / Pagreitėjimas	500 000	1000 000	2000 000	Vidurkis
1 ir 2 patobulinimo % pagreitėjimas	3,64%	3,21%	2,88%	3,24%

Lentelė 1. Programos pagreitėjimas atlikus 1 ir 2 patobulinimus.

Atlikus analizę bei testus pastebėta, kad objektų nuskaitymas ir įrašymas į „Array“ tipo kintamąjį yra greitesnis nei dirbant su „ArrayList“ ar „Vector“ kintamaisiais. To pasekoje buvo atliktas trečias programos patobulinimas.

Paprastasis „Array“ masyvas yra statinio tipo, taigi buvo praplėstos jo galimybės ir papildyta dinaminiu masyvo praplėtimo funkcija. Saugant nuorodas tokiu būdu buvo pasiektas vidutiniškai 36% pagreitėjimas. Taip pat sutaupyta 0,92% vietos atmintyje. Šis programos patobulinimas parodė puikius rezultatus.

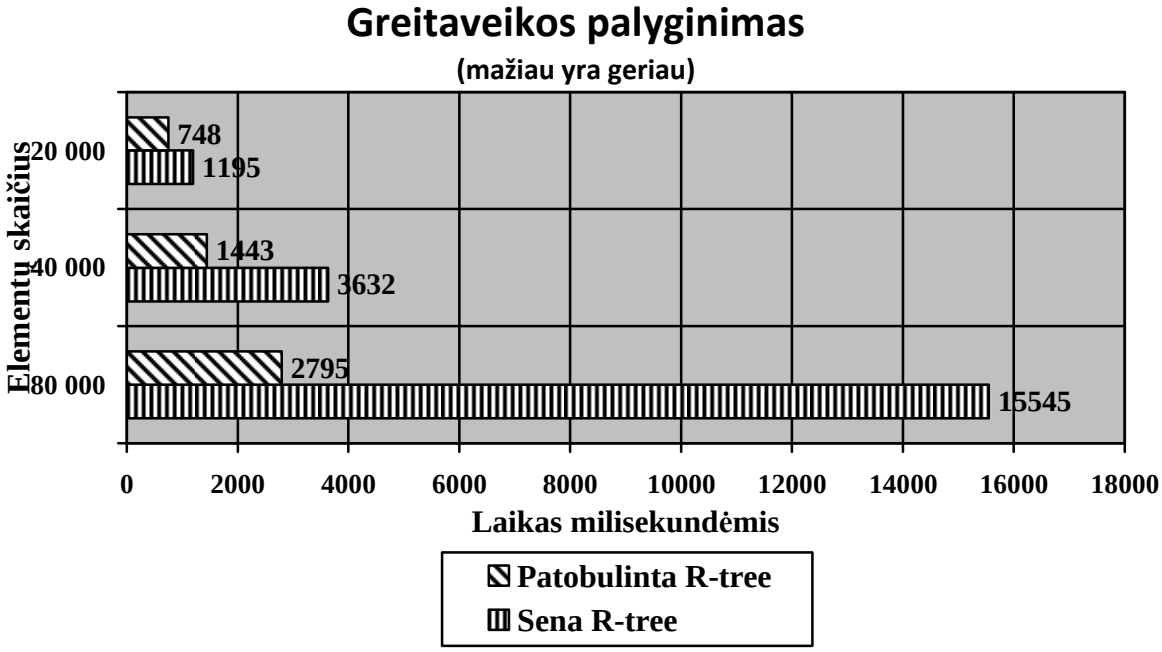
Elementų sk. / Pagreitėjimas	500 000	1000 000	2000 000	Vidurkis
3 patobulinimo % pagreitėjimas	35,45%	36,44%	37,30%	36,39%

Lentelė 2. Programos pagreitėjimas atlikus 3 patobulinimą.

Iš daugelio tyrimų pastebėta, kad silpniausia R-medžio vieta yra mazgo dalinimo funkcija. Todėl nuspręsta optimizuoti „Mazgo dalinimo“ algoritmą. Kaip pagrindas panaudotas „Pilno perrinkimo“ mazgo dalinimo algoritmas. Sugalvotas spartus būdas, kaip greitai suskaičiuoti grupių sekas. Taip pat viso medžio užpildymo laiką tik pirmąjį kartą sugeneruojami visi įmanomi sugrupavimo variantai. Išgeneruojama tik pirmoji grupė, o kas liko automatiškai priklauso antrajai. Šis patobulinimas davė labai gerų rezultatų, medžio užpildymas vidutiniškai 17% tapo greitesnis.

Elementų sk. / Pagreitėjimas	500 000	1000 000	2000 000	Vidurkis
4 patobulinimo % pagreitėjimas	17,05%	17,00%	17,26%	17,10%

Lentelė 3. Programos pagreitėjimas atlikus 4 patobulinimą.



Programa / Elementų skaičius	20 000	40 000	80 000
Sena R-tree programa	1195 ms	3632 ms	15545 ms
Patobulinta R-tree programa	748 ms	1443 ms	2795 ms

Lentelė 4. Elementų įdėjimo į R-medį statistika.

Gauti rezultatai rodo tai, kad patobulinta „R-tree“ programa tikrai dirba sparčiau. Pasiektas 50% ir daugiau spartesnis medžio užpildymas. Šiais rezultatais nesuabejojau, nes tikrai daug laiko praleidau optimizuodamas ir spartindamas programos veikimą. Iš rezultatų matosi, kad kuo daugiau duomenų tuo greitaveikos skirtumas didesnis.

5. Realizuoto R-medžio palyginimas

Įdėjus į programą tiek daug darbo ir pastangų, o dar po to ji buvo tobulinama ir stengiamasi, kad ji veiktų greitai ir patikimai, tikrai norima įsitikinti, kad praleistas laikas programuojant R-medį nebuvo praleistas ne veltui. Su noru palyginti programos veikimo greitį buvo ieškoma internete R-medžio programos ar išeities kodų, kad būtų galima susikompiliuoti programą ir pradėti lyginimus.

5.1. Internetinių taikomųjų programų palyginimas

Prieš pradedant paiešką tikrai negalvojau, kad gali būti taip sunku surasti, ko man reikia. Ieškodamas radau tris interneto svetaines, kuriose buvo R-medžio taikomosios programos (Application), parašytos JAVA programavimo kalba ir integruotos į internetinį tinklapį. Nei viena iš šių taikomųjų programų netiko palyginimams. Jos neturėjo operacijos atlikimui sugaišto laiko skaičiavimo funkcijos, taigi tiksliai negalėjau žinoti kiek programa praleido laiko darbui su R-medžiu. Taip pat programų galimybės buvo ribotos, į vieną iš programų galėjo įdėti tiktai 5000 taškų, ko palyginimui nepakako, o kitose programose nebuvo taškų generavimo funkcijos. Be to truputi padirbėjus su šiomis programomis iškarto pastebėta, kad jos dirba lėčiau nei mano sukurta programa. Tačiau galiu pasakyti, kad labiausiai patiko dirbti su 1-moje internetinėje svetainėje esančia programa.

Interneto svetainių sąrašas:

1) http://donar.umiacs.umd.edu/quadtree/points/rtrees.html;
2) http://gis.umb.no/gis/applets/rtree2/jdk1.1/;
3) http://www.cse.hut.fi/en/research/SVG/TRAKLA2/exercises/RTreeInsert.html.

Svetainė / Funkcija	1 Interneto svetainė	2 Interneto svetainė	3 Interneto svetainė
Sugaišto laiko skaičiavimas	Ne	Ne	Ne
Apribojimas	Tik 5000 taškų		Negalima įdėti savų taškų, tik parodomoji versija
Taškų generavimas	Taip	Ne	Ne

Lentelė 5. Programų, integruotų į interneto svetaines, palyginimas.

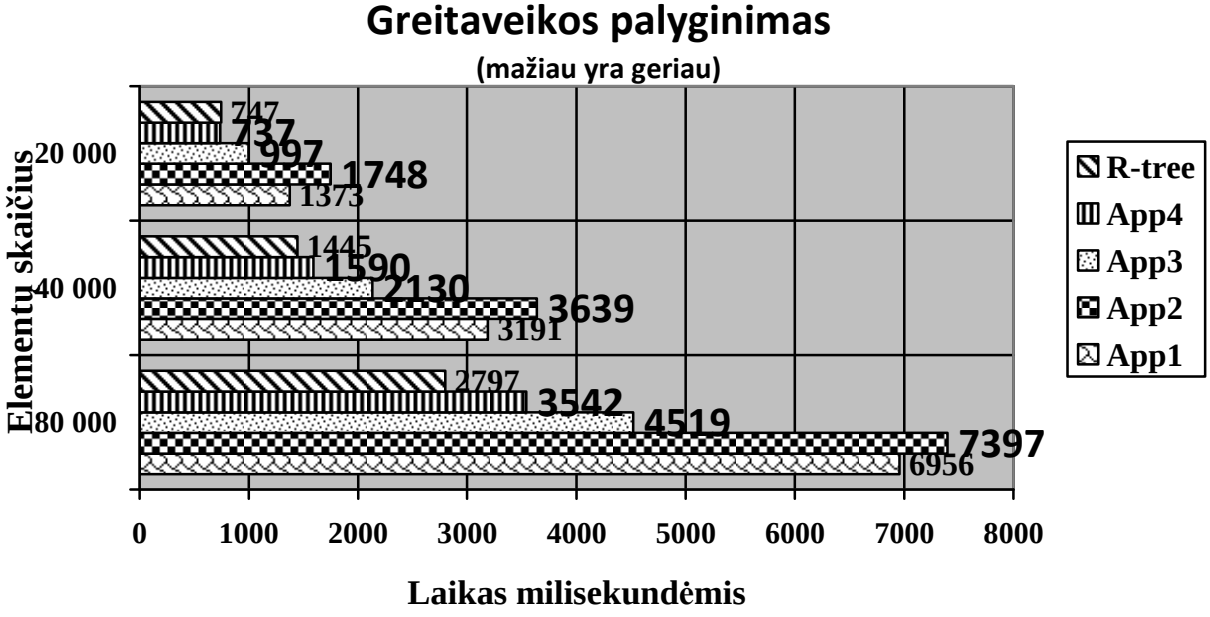
5.2. Įdėjimo operacijos palyginimas su kitomis programomis

Susidūrus su visomis anksčiau išvardintomis problemomis nusprendžiau, kad reikia pradėti ieškoti R-medžio programų išeities kodų ir mėginti programas susikompiliuoti pačiam. Surasti R-medžio programos išeities kodus taip pat nebuvo lengva. Nedaug yra viešai prieinamų R-medžio programos išeities kodų, tačiau kelėta radau ir tuo jau buvau patenkintas. Atsidaręs tuos kodus nustebau, nes aprašyta programa neturėjo vartotojo sąsajos (User Interface) ir nebuvo parašytas programos paleidimas. Nusprendžiau, kad vartotojo sąsajos tikrai neaprašinėsiu toms programoms, juk vis dėlto jas siunčiausi tik palyginimui, kaip greitai jos veikia. Kad programas paleisčiau, teko pasirašyti „main" klases. Šiose „main" klasėse turi būti sukuriamas tuščias R-medis ir procedūros iškvietimas, kuri įdeda duomenis į medį. Pasirašius ko trūko, kad būtų galima paleisti programą, galėjau pradėti testavimus.

Parsisiųsta ir pritaikyta palyginimams buvo keturios programos. Iš jų viena programa parašyta su C programavimo kalba, o kitos trys su JAVA programavimo kalba.

Programos paruoštos palyginimams:

1) Programą parašytą su C programavimo kalba pavadinkime „App1", nuo žodžio Application. Programa leidžia to paties taško įdėjimą į medį antrą kartą, tai reiškia, kad programa netikrina ar taškas jau egzistuoja medyje;
2) Sekančią programą, parašytą su JAVA programavimo kalba, pavadinkime „App2". Mazgo dydis *M* lygus 5, o *m* lygu 2;
3) Toliau kitą JAVA programą pavadinkime „App3". Ši programa yra R*-medžio tipo. Į šią programą buvo leidžiama įdėti tą patį tašką antrą kartą;
4) Ketvirtąją parsisiųstą programą pavadinkime „App4";
5) Mano parašyta programa yra „R-tree" pavadinimu. Mazgo dydis *M* lygus 5, o *m* lygu 2. Į medį galima įdėti kelis vienodus taškus.



Programos pavadinimas \ Elementų skaičius	20 000	40 000	80 000
App1	1373 ms	3191 ms	6956 ms
App2	1748 ms	3639 ms	7397 ms
App3	997 ms	2130 ms	4519 ms
App4	737 ms	1590 ms	3542 ms
R-tree	747 ms	1445 ms	2797 ms

Lentelė 6. Taškų įdėjimo į R-medį statistika.

Gauti rezultatai rodo tai, kad mano sukurtos programos medžio užpildymo greitis yra sparčiausias. Šiais rezultatais nesuabejojau, nes tikrai daug laiko praleidau optimizuodamas ir greitindamas programos veikimą. Antrąją vietą būtų galima skirti „App4“ programai. Ši programa ne tiek ir daug atsiliko nuo „R-tree“, tačiau jei duomenų kiekis būtų didesnis tai ir laiko skirtumas ženkliai padidėtų.

5.3. Paieškos operacijos palyginimas

Po tokios geros pradžios norėjosi įsitikinti, kad ne tik įdėjimas, bet ir taško suradimas medyje taip pat yra toks greitas lyginant su kitomis programomis. Pradėjus palyginimus pasirodė, kad visų programų, taip pat ir mano, paieška medyje su 80 000 elementų yra labai greita ir rezultatai

nesiskiria. Visų programų paieška užtrukdavo nuo 4 iki 10 milisekundžių. Taip išaiškėjo, kad R-medis tikrai pelnytai yra giriamas už savo paieškos greitį.

Nuodugnesniam paieškos tyrimui pasinaudojau vien tik savo sukurta programa. Į medį pakroviau 500 000 elementų, visų elementų užkrovimas užtruko 10916 milisekundžių. Mazgo dydis *M* lygus 5, o *m* lygu 2. Turėdamas tokį kiekį duomenų tikrai tikėjausi, kad paieškos laikas ženkliai padidės, tačiau vieno konkretaus elemento paieška užtruko apie 5 milisekundes. Tai dar kartą įrodo, kad R-medyje paieška tikrai labai greita kad ir kiek duomenų jame bebūtų.

Elementų / Programa	500 000	1000 000	1500 000	2000 000
R-tree	5 ms	7 ms	9 ms	11 ms

Lentelė 7. Taškų paieškos greitis (pateikiamas lėčiausios paieškos rezultatas).

5.4. Kompiuterio atmintie sunaudojimo palyginimas

Sekančiu etapu lyginau programų laisvosios prieigos atminties (RAM) sunaudojimą. Norėjosi išsiaiškinti, kuom greitis ir atminties sunaudojimas yra susiję. Gauti tokie rezultatai.

Programa / Elementų	App1	App2	App3	App4	R-tree
20 000	1,5 MB	3,5 MB	3 MB	5 MB	2,5 MB
40 000	2,3 MB	6,8 MB	5 MB	7,3 MB	3,7 MB
80 000	4 MB	13,8 MB	7,9 MB	13,9 MB	7,1 MB

Lentelė 8. Programų laisvosios prieigos atminties (RAM) sunaudojimas.

„App1“ programa sunaudoja mažiausiai atminties, bet jos taškų įdėjimas nėra greitas. Be to ši programa parašyta su C programavimo kalba, kas tikriausiai ir nuliame mažą atminties sunaudojimą. Kitos programos rašytos su JAVA ir jų atminties sunaudojimo kiekis didėja, bet programos veikimo laikas trumpėja. Iš visų rezultatų geriausiu deriniu pasižymi „R-tree“ programa. Šios programos veikimo laikas trumpiausias ir taip pat atminties sunaudojimas mažiausias, jei lygintume atminties sunaudojimą programų rašytų su JAVA programavimo kalba.

5.5. R-medžio įvertinimas

Suprogramavus R-medį pastebėta, kad jis puikiai tinka daugiamačių duomenų saugojimui. Kaip ir parodė testai, daugiamačiai duomenys buvo greitai apdoroti ir suindeksuoti. Medžio užpildymo greitis tolygiai didėjo augant duomenų kiekiui, t.y. 1 elemento įdėjimas į medį beveik nesiskiria jei medis jau turi 1000 ar 100 000 elementų. To pasekoje medžio užpildymo laikas yra lengvai nuspėjamas ir apskaičiuojamas. Taip pat atlikus paieškos greičio testus paaiškėjo, kad paieška atliekama panašiai vienodu greičiu ir nesvarbu kiek medyje yra duomenų.

Iš atliktų eksperimentų prieita išvada, kad optimalus R-medžio mazgo dydis yra 5. Su tokiu medžio mazgo dydžiu puikiai apdorojami tiek maži, tiek dideli duomenų kiekiai.

R-medžio privalumai būtų tokie:

- R-medžio indeksas yra labai mažas, jis užima mažai disko vietos ir sukūrimo bei palaikymo laikas yra mažas;
- Kilimas medžiu aukštyn yra optimalus greičio požiūriu;
- Greitai vykdomas medžio atnaujinimo algoritmas;
- R-medis laiko informaciją apie duomenų objektą lapo taške.

6. Klasterizavimas

Daugiamačių duomenų analizėje taip pat naudojami klasterizavimo metodai. Šie metodai naudojami pastaruosius 30 metų. Klasteris yra duomenų ar objektų rinkinys, kur tie objektai yra tarpusavyje „panašūs“, o tų objektų grupavimas vadinamas klasterizavimu. Panašumas tarp objektų yra nustatomas „atstumo“ funkcija. Kitaip tariant, kiekvienai duomenų porai p_1, p_2 yra skaičiuojamas atstumas $D(p_1, p_2)$. Taip pat, be atstumo funkcijos dar yra naudojama papildoma „kokybės“ funkcija, kuri naudojama klasterio „gerumui“ apskaičiuoti.

Klasterizavimo algoritmai yra visapusiškai išstudijuoti ir pastebėta, kad kai duomenys gali būti „sutvarkyti“, t.y., surūšiuoti, klasterizavimo algoritmai efektyviausi [Jag90]. Tai yra dėl to, kad turima tvarkos savybė apibrėžia natūralią atstumo funkciją klasterio algoritmams. Duomenų objektams, kurie neturi nei reikiamų atributų atstumams tarp objektų apskaičiuoti, nei ryšių tarpusavyje, nėra taikomi klasterizavimo algoritmai. Taip yra dėl to, jog klasterizavimo algoritmai reikalauja, kad būtų galima apskaičiuoti atstumo funkciją.

6.1. K-Means klasterizavimo algoritmas

K-means yra vienas iš paprastesnių algoritmų, kuris puikiai sprendžia klasterizavimo problemą. Šį algoritmą sugalvojo J. B. MacQueen 1967 metais. K-means algoritmas dirba su duomenimis, kurie yra apibrėžti koordinatėmis. Šis klasterizavimas yra paprastas būdas duotų duomenų klasifikavimui į nustatytą *K* klasterių skaičių. Pagrindinė idėja apsibrėžti *k* masių centrų, kur kiekvienas atitinka klasterį. Šie klasterių centrai pirmąjį kartą turi būti įžvalgiai išdėstyti, nes skirtingos vietos parinkimas duos skirtingus rezultatus. Taigi geriausias pasirinkimas būtų patalpinti juos kuo toliau vieną nuo kito [KMN+02]. Sekantis žingsnis - paimti visas viršūnes ir joms surasti artimiausią klasterio centrą bei priskirti viršūnę jam. Kuomet visos viršūnės priskirtos klasterių centrams, pirmasis etapas ir išankstinis grupavimas yra baigti. Sekančiu žingsniu perskaičiuojame *k* naujų klasterio centrų, kurie būtų prieš tai suskaičiuotų klasterių centrų masyvų centrai. Kai turime šiuos naujus *k* centrus vėl reikia atlikti surišimą. Visos viršūnės yra priskiriamos artimiausiam klasterio centrui. Vykdomas rekursinis ciklas, kuris keičia klasterių centrų vietas ir vis naujai priskiria jiems viršūnes. Procesas užbaigiamas, kuomet pakitimų nebėra ir klasterių centrai nebejuda iš vietos.

Tikslo funkcija:

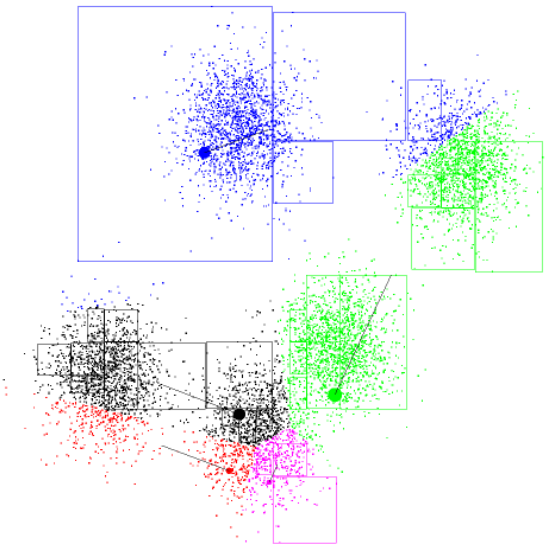
$$J = \sum_{j=1}^{k} \sum_{i=1}^{n} \| x_i^{(j)} - c_j \|^2$$

Kur $\| x_i^{(j)} - c_j \|^2$ yra atstumo skaičiavimas tarp viršūnės $x_i^{(j)}$ ir klasterio centro c_j. Skaičius nusakantis viršūnių skaičių žymimas *n*, o klasterių centrų skaičius yra *k*.

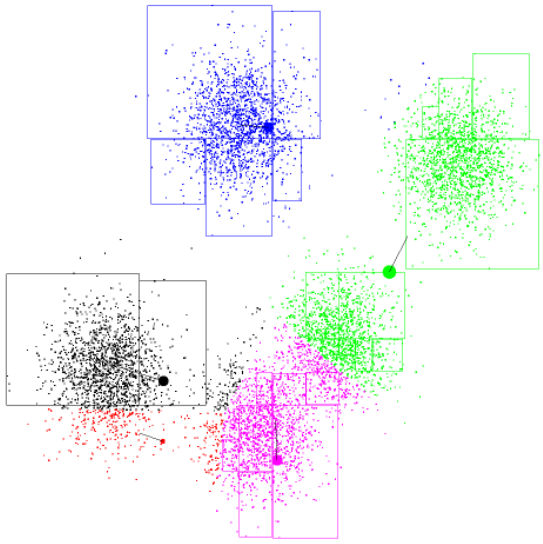
Klasterizavimo žingsnių algoritmas [Pol81]:

1. Nustatoma, kiek klasterių centrų bus naudojama.
2. Erdvėje sustatomi *K* klasterio centrų, kurie kuo įmanoma daugiau nutolę vienas nuo kito.
3. Visos viršūnės priskiriamos artimiausiems klasterio centrams.
4. Perskaičiuojami ir perdedami klasterio centrai.
5. Kartojami 3 ir 4 žingsniai tol, kol klasterio centrai keičia savo pozicijas. Kuomet centrai nebejuda ciklas nutraukiamas.

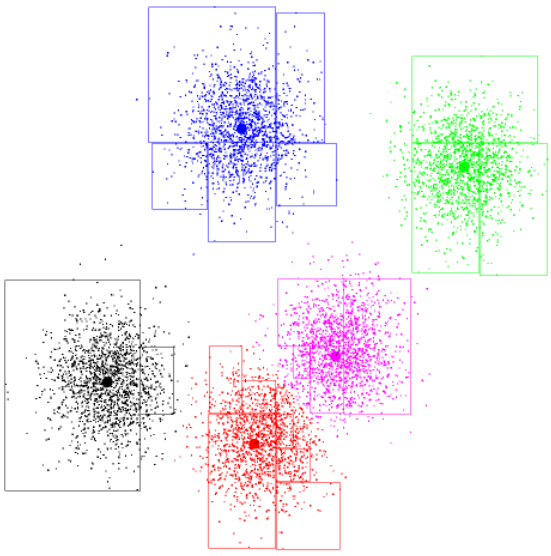
Šiame pavyzdyje (6 paveikslėlis) yra 8000 viršūnių išdėliotų 2 dimensijų erdvėje. Buvo nuspręsta, kad bus naudojamas 5-means (K-means su K=5) klasterizavimas. Iš pradžių visi 5 klasterių centrai yra atsitiktinai išdėliojami erdvėje. Kad lengviau pastebėti, kokiam klasteriui priskirtos viršūnės, jos nuspalvinamos skirtinga spalva. Paveikslėlyje 6.1 matome, kad yra perskaičiuoti klasterių centrai ir linijomis pažymėta naujai norimų patalpinti klasterių centrų vieta. Po klasterių centrų perdėjimo (6.2 paveikslėlis) viršūnės naujai priskiriamos klasteriams ir vėl skaičiuojama nauja klasterio centro pozicija.



Pav. 6.1: Klasterių centrų perskaičiavimas.



Pav. 6.2: Po klasterių centrų perdėjimo.



Pav. 7: Gautas rezultatas.

Visi centrai buvo perdėliojami tol, kol klasterių centrai neturėjo stabilios pozicijos. Kada atsirado pakankamai stabili vieta, gavome 5 klasterių grupes. Galutinis rezultatas yra pavaizduotos 7 paveikslėlyje.

Reikia pabrėžti jog šis algoritmas yra labai priklausomas nuo to, kaip bus išdėstyti klasterio centrai pirmąjį kartą. Norint sumažinti šią algoritmo silpnybę, galima kelis kartus vykdyti algoritmą ir išsirinkti tą variantą, kuris bus geresnis.

Kiek reikėtų naudoti klasterių centrų efektyviam klasterizavimui nėra žinoma, kadangi skirtingiems duomenims gali būti skirtingas skaičius klasterių. Beveik visada yra sunku nusakyti, kiek klasterių geriau turimiems duomenims klasterizuoti. Tiesiog bandymų metu aiškinamasi ir ieškoma geresnio varianto. Tačiau nereikėtų bandyti su dideliu klasterių centrų kiekiu, nes per didelis klasterių centrų skaičius gali blogai suklasterizuoti duomenis [HM04].

6.2. CURE klasterizavimo algoritmas

Tradiciniai klasterizavimo algoritmai puikiai klasterizuoja duomenų grupes, kurios labiau viena nuo kitos atitolusios. Tačiau norint gerai suklasterizuoti duomenis, kurie yra arti vienas kito, į kelias grupes yra sunku. Tokiems duomenims klasterizuoti yra naudojamas naujasis CURE (angl. Clustering Using REpresentatives) algoritmas. Šis algoritmas yra efektyvus duomenų klasterizavimo algoritmas ir naudojamas darbui su dideliais duomenų kiekiais. Jis yra labiau patikimas dirbant su duomenimis, kurie yra arti vienas kito, nei kiti algoritmai [GRS01].

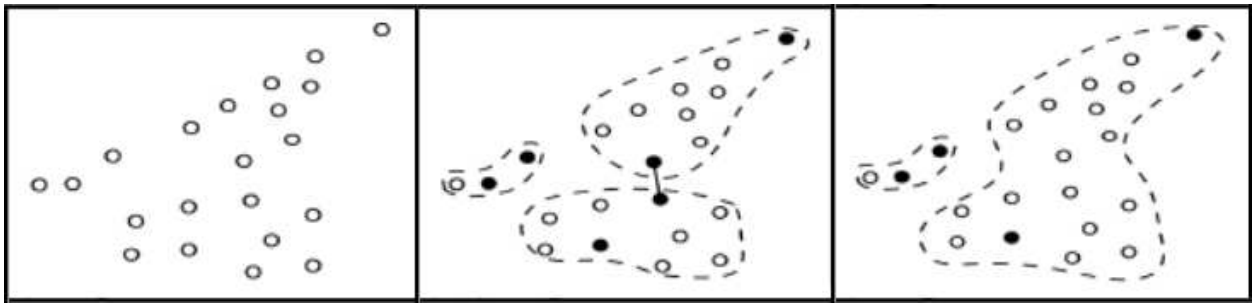
Pagrindinė CURE algoritmo idėja skaičiuoti atstumus tarp klasterių. Du klasteriai, turintys mažiausią atstumą vienas iki kito, yra apjungiami į vieną. Šie veiksmai vykdomi tol kol klasterių lieka tiek kiek norima jų turėti.

Šio CURE klasterizavimo algoritmo sudėtingumas yra $O(n^2 \log n)$ [GRS01].

CURE klasterizavimo algoritmo žingsniai [GRS97]:

1. Kiekvienas taškas u laikomas kaip atskiras klasteris. Iš pradžių kiekvienas klasteris turi tik po vieną tašką.
2. Kiekvienam u surandame artimiausia klasterį.
3. Visi klasteriai u sudedami į masyvą Q. Prieš dedant klasterius į masyvą jie yra išrūšiuojami didėjimo tvarka pagal apskaičiuotą atstumą iki artimiausio klasterio.
4. Kol masyve Q klasterių skaičius yra didesnis nei k (k – skaičius, kiek norima turėti klasterių), tol atliekami veiksmai:
    a. Paimamas viršutinis masyvo Q elementas (tarkime u) ir suliejamas su artimiausiu klasteriu (tarkime v). Tokiu būdu gauname naują klasterį w;
    b. Ištriname klasterius u ir v iš masyvo Q;
    c. Įterpiame klasterį w į masyvą Q;
    d. Visiems masyvo Q klasteriams perskaičiuojame artimiausią klasterį;
    e. Visi masyvo Q klasteriai išrūšiuojami didėjimo tvarka pagal apskaičiuotą atstumą iki artimiausio klasterio.

Šiame 8 paveikslėlyje pateikiamas pavyzdys kaip apjungiami klasteriai. Klasterizavimo algoritmas pradedamas nuo to kad kiekvienas taškas laikomas atskiru klasteriu. Tuomet algoritmas apjungia du artimiausius klasterius. Atlikus keletą apjungimų gaunami 3 klasteriai. Šiuo atveju norima kad klasterizavimo algoritmas rezultate pateiktų 2 klasterius. Taigi toliau tęsiamas artimiausių klasterių apjungimas. Apjungus klasterius gaunamas galutinis rezultatas, 2 klasteriai.



Pav. 8: Klasterių apjungimas.

7. K-means algoritmo realizacija

Klasterizavimo algoritmo realizacijai pasirinktas K-means algoritmas. Šis algoritmas pasirinktas būtent dėl to, kad jis yra pritaikytas dirbti su duomenimis, kurie yra apibrėžti koordinatėmis. Būtent tokie daugiamačiai duomenys ir yra apdorojami.

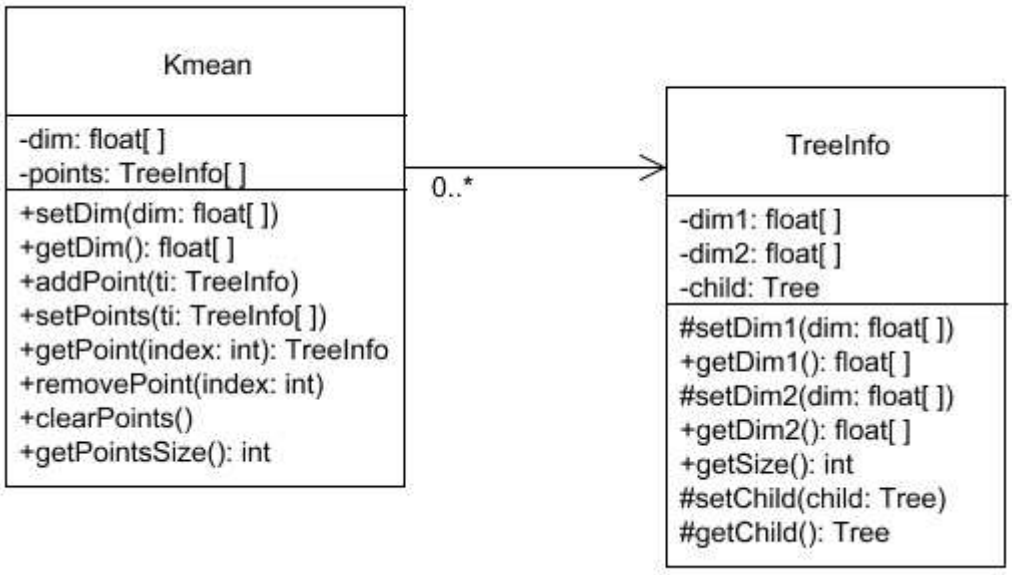
Kadangi R-medis jau yra mano paties suprogramuotas, tai ir K-means algoritmą realizuoti nutariau pats. Tokiu būdu bus lengviau pritaikyti algoritmą savo tikslams ir poreikiams. Taip pat laikausi tokios nuomonės, kad norint pilnai suprasti ir išanalizuoti K-means klasterizavimo algoritmą reikia pačiam jį suprogramuoti.

7.1. Projektavimas

Anksčiau pateiktame 6.1 poskyryje yra pilnai aprašytas K-means klasterizavimo algoritmas, po žingsnį aprašytas jo veikimo principas. Pasinaudojus šia informacija buvo suprogramuotas K-means algoritmas.

Programavimo darbai buvo atliekami praplečiant suprogramuoto R-medžio programą nauju funkcionalumu.

Prieš programavimą pasibraižiau UML diagramą, kaip atrodys programuojama K-means klasė, kokia jos struktūra ir funkcijos.



Pav. 9: K-means UML diagramą.

K-means klasterio centras aprašomas viena JAVA klase. Sukurtas toks objektas savyje saugo klasterio centro koordinates ir nuorodas į aplink jį esančius R-medžio lapus.

Klasterio centro koordinatės saugomos „float“ tipo kintamuosiuose, pvz. jei klasterio centras yra dviejų dimensijų plokštumoje, tai šiam taškui aprašyti reikės dviejų kintamųjų (x, y). Kad programa būtų lanksti ir ją būtų galima pritaikyti bet kokios dimensijos duomenims, „Kmean“ objekte saugomas dimensijos masyvas į kurį galima įrašyti bet kokios dimensijos taško koordinates.

Nuorodoms į lapus užpildyti sukurtos procedūros „addPoint“ vienos nuorodos į lapą priskyrimui ir „setPoints“ priskirti visas iš anksto surinktas nuorodas vienu metu. Taip pat sukurta funkcija, kuri gražina skaičių kiek viso yra priskirtų nuorodų į lapus.

7.2. Įvertinimas

Skirtingai nei R-medis, šio K-means klasterizavimo algoritmo spartinti nereikėjo. Kadangi algoritmas yra paprastas ir sudarytas iš kelių pagrindinių žingsnių, tai tobulinimo ir optimizavimo darbų neprireikė. Šio algoritmo greitaveika slypi jo paprastume.

Tačiau susidurta su vienu šio algoritmo pagrindiniu minusu. Teorinėje medžiagoje buvo rašoma, kad K-means algoritmo greitis yra priklausomas nuo klasterio centrų išdėstymo. Suprogramavus K-means algoritmą ir atlikus analizę pastebėjau kad tai tiesa. Pirmą kartą paleidus algoritmą rezultatai gali būti labai geri, o jau paleidus sekantį kartą - vykdymo laikas žymiai ilgesnis. Jei lyginant šį algoritmą su R-medžiu, tai iš pastebėjimų galima pasakyti jog medžio užpildymo greitis, paleidus kelis kartus ir sulyginus, skiriasi maždaug 1%. Toks skirtumas yra normalus ir priimtinas. O sulyginus K-means algoritmo greitaveiką gaunamas 10% ar netgi truputi didesnis skirtumas. To pasekoje sunku prognozuoti kiek laiko bus vykdomas algoritmas.

Nepaisant nenuspėjamo vykdymo laiko K-means klasterizavimo algoritmas išties yra greitas ir efektyvus. Jis greitai apdoroja duomenis ir juos suklasterizuoja. Tą parodė atliktas eksperimentas. Prireikė 1 milijono duomenų, kad būtų pajaustas greitaveikos skirtumas klasterizuojant paprastus duomenis ir suindeksuotus.

Atliekant analizę taip pat pastebėta, kad didelę įtaką algoritmo greitaveikai turi klasterio centrų skaičius. Kuo daugiau klasterio centrų, tuo algoritmas darosi lėtesnis. Tačiau tai visiškai normalu, nes reikia analizuoti didesnį kiekį klasterių centrų.

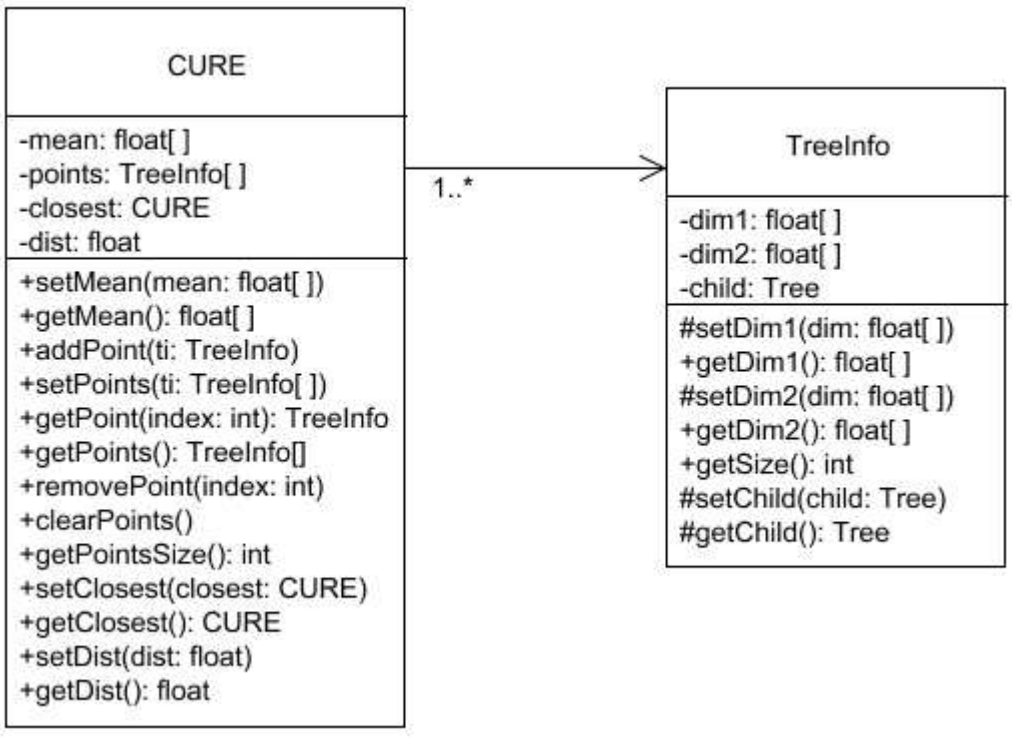
8. CURE algoritmo realizacija

Antrajai klasterizavimo algoritmo realizacijai pasirinktas CURE algoritmas. Šis algoritmas yra pakankamai naujas, jei lygintume su K-means algoritmu. Jis naudojamas didelių duomenų kiekiams apdoroti, bei labiau patikimas dirbant su duomenimis, kurie yra arti vienas kito.

8.1. Projektavimas

CURE klasterizavimo algoritmas yra pilnai aprašytas 6.2 poskyryje. Aprašytas jo veikimo principas ir pateiktas algoritmo išsamus aprašymas po žingsnį. Pasinaudojus šia informacija buvo suprogramuotas CURE algoritmas.

Programavimo darbai buvo atliekami praplečiant suprogramuoto R-medžio programą nauju funkcionalumu. Prieš programavimą pasibraižiau UML diagramą, kaip atrodys programuojama CURE klasė, kokia jos struktūra ir funkcijos.



Pav. 10: CURE UML diagramą.

CURE algoritmo klasteris aprašomas viena JAVA klase. Sukurtas toks objektas savyje saugo klasterio centro koordinates, nuorodas į aplink jį esančius R-medžio lapus, nuorodą į artimiausią kitą klasterį ir atstumą iki artimiausio klasterio.

Klasterio centro koordinatės saugomos „float“ tipo kintamuosiuose, pvz. jei klasterio centras yra dviejų dimensijų plokštumoje, tai šiam taškui aprašyti reikės dviejų kintamųjų (x, y). Kad programa būtų lanksti ir ją būtų galima pritaikyti bet kokios dimensijos duomenims, „CURE“ objekte saugomas dimensijos masyvas į kurį galima įrašyti bet kokios dimensijos taško koordinates.

Nuorodoms į lapus užpildyti sukurtos procedūros „addPoint“ vienos nuorodos į lapą priskyrimui ir „setPoints“ priskirti visas iš anksto surinktas nuorodas vienu metu.

Tik sukurtas CURE klasteris užpildomas vienu R-medžio lapu. Sekančiu žingsniu visiems šiems klasteriams suskaičiuojami artimiausi klasteriai. Toliau vykdomas rekursinis ciklas tol kol lieka tik norimas klasterių kiekis. Cikle apjungiami du klasteriai ir perskaičiuojamas artimiausias klasteris.

8.2. Įvertinimas

Lyginant CURE klasterizavimo algoritmą ir K-means iškarto pastebima, kad CURE algoritmas yra sudėtingesnis. Šis iš pažiūros naujas algoritmas atlieka daugiau skaičiavimų ir funkcijų. Suprogramuoti jį nebuvo sunku nors ir jis yra sudėtingesnis. Vykdoma daug, tačiau nesudėtingų skaičiavimų.

CURE klasterizavimo algoritmas yra giriamas dėl gaunamų puikių rezultatų. Kaip teigiama [GRS01] jis puikiai klasterizuoja įvairių tipų duomenis. Tačiau, nors jis ir yra giriamas, šis algoritmas turi vieną pagrindinį minusą. Atlikti eksperimentai parodė kad jis yra lėtas. Tik suprogramavus ir paleidus keletą kartų pastebėta, kad jo vykdymo trukmė žymiai ilgesnė nei K-means klasterizavimo algoritmo.

Iš testų pastebėta, kad kaip ir R-medis, taip ir šis CURE algoritmas vyksta tolygiai. Iš pastebėjimų galima pasakyti, kad algoritmo greitis, paleidus kelis kartus ir sulyginus, skiriasi maždaug 1,5%. Toks skirtumas yra normalus ir priimtinas. Dėl šio teigiamo CURE klasterizavimo algoritmo bruožo galima lengviau prognozuoti kiek laiko jis bus vykdomas.

9. EEG įrašymas ir kaupimas

Duomenys, kuriems bus pritaikyti aukščiau aprašyti duomenų gavybos (angl. Data Mining) algoritmai, yra gauti iš elektroencefalogramų (EEG). Šiems duomenis surinkti yra išskirta atskira mokslo šaka kuri vadinasi elektroencefalografija.

Elektroencefalografija - galvos paviršiuje esančio elektrinio lauko pasikeitimo matavimas ir užrašymas. Elektrinį lauką keičia tūkstančių neuronų sinchroniška veikla - sinapsiniai aktyvavimai [NL05]. 1924 m. Hansas Bergeris užrašė žmogaus elektroencefalogramą, o savo rezultatus publikavo 1929 m. Jis taip pat atrado Alfa bangų blokus. Buvo pastebėti žymūs EEG pakitimai, kai bandomasis atsimerkdavo arba buvo skatinamas išlaikyti protinės veiklos aktyvumą. Nors tokie tyrimai atrodė naujiena, tačiau su gyvūnais jie buvo vykdomi jau nuo 1870 metų [NL06]. Elektroencefalografija yra naudojama smegenų veiklai tyrinėti, padeda pažinti vykstančius procesus, nustatyti negalavimus, smegenų veiklos anomalijas. EEG naudojama klinikiniams, neurologiniams tyrimams atlikti.

Klinikiniuose tyrimuose EEG yra naudojamas [THB+07]:

- Epilepsijos diagnozavimui, klasifikavimui ir teisingo gydymo kelio nustatymui;
- Diagnozuoti kitus, ne epileptinius smegenų veiklos sutrikimus;
- Nustatyti smegenų būseną (miego stadija, mieguistumas) bei smegenų mirties faktui nustatyti;
- Diagnozuoti miego sutrikimus, narkolepsiją;
- Lokalizuoti smegenų vėžį, epilepsijos šaltinį;
- Stebėti anestetikų veikimą;
- Stebėti komos būklės pacientus.

Elektroencefalograma (EEG) - tai žmogaus smegenų neuronų elektrinių impulsų įrašas – smegenų būklė. Nagrinėdami EEG galime nustatys žmogaus dabartinę būklę, o kartais netgi ir atliekamus veiksmus. Todėl yra nenuostabu, jog šis biomedicininis signalas yra labai palčiai naudojamas, tiek siekiant suprasti žmogaus smegenų veiklą, tiek diagnozuojant bei tyrinėjant įvairias neurologines ligas.

EEG susideda iš kanalų, parodančių smegenų veiklą skirtinguose regionuose. Kanale esantis signalas yra skirstomas į alfa, beta, gama, delta, teta bangas pagal dažnį. Šios bangos yra siejamos su tam tikra žmogaus veikla, būsena.

9.1. EEG analizė

Biomedicininių signalų analizė labai svarbi žmogaus sveikatos būsenai nustatyti. Vienas iš pagrindinių signalų šaltinių - neuronais keliaujantys nerviniai impulsai, aplink save sukuriantys mažą, bet pastebimą elektrinio lauko pakitimą. Tipinę elektroencefalogramą sudaro nuo 12 iki 128 kanalų signalas, užrašomas 128Hz ar didesniu dažniu.

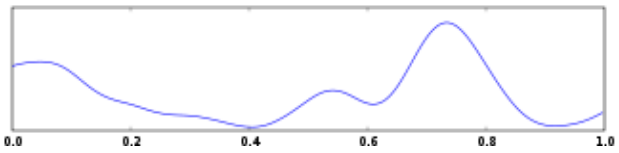
Žmogaus nervų sistemoje nuolatos vyksta įvairūs procesai, kai kurie iš jų palieka labai matomas žymes elektroencefalogramoje. Vieni procesai yra ritmingi, sinusoidės formos, jie klasifikuojami pagal dažnių juostas (ritmą, bangas). Kiti procesai yra ne ritmingi (periodiškai nesikartoja) - tai miego kompleksai, epilepsiniai ir neepilepsiniai pikai, akies ir kitų raumenų judesių palikti artefaktai [THB+07].

9.1.1 Smegenų bangos

EEG signalą sudaro keleto smegenų bangų (angl. brain wave) suma, epilepsiniai pikai bei kiti artefaktai - akių, raumenų judesiai. Kiekviena banga turi savo biologinę prasmę ir interpretaciją. Jų dažninė spektrograma leidžia geriau suprasti fiziologinę (smegenų) būseną - miego stadiją, mieguistumą ir nuovargį, todėl dažnai naudojama medicinoje ligai ar negalavimui diagnozuoti, smegenų veiklos stebėjimui operacijos metu ir tt. EEG yra labai efektyvus įrankis epilepsijai diagnozuoti ir klasifikuoti, teisingam gydymo keliui nustatyti.

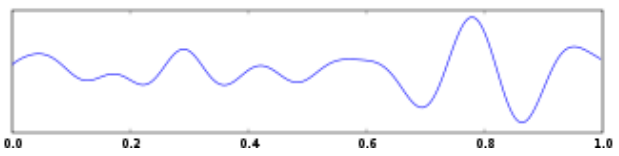
Pagal bangų dažnį yra išskiriamos penkios pagrindinės smegenų bangų grupės [NL06].

Delta bangos (0,1 – 3,9 Hz). Fiksuojama kada veikiama nesąmoningai arba giliai miegant. Esant šiam dažniui miego metu nesapnuojama. Jos siejamos su gydymo procesu, imuninės sistemos atnaujinimu, organizmo veiklos harmonizavimu [NL06].



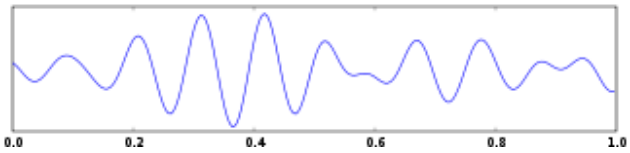
Pav. 11: Delta bangos.

Teta bangos (4 – 7,9 Hz). Dažniausiai pastebimos vaikų bei suaugusių mieguistumo ar susijaudinimo stadijose. Taip pat pastebima meditacijos metu [NL06].



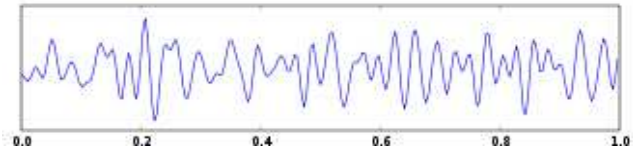
Pav. 12: Teta bangos.

Alfa bangos (8 – 13,9 Hz). Transo būsenos ir negilaus miego bangos. Jos atsiranda atsipalaidavus, esant užmerktoms akims bei nusiraminus. Alfa bangos atitinka budrios sąmonės, esant atsipalaidavus, būseną [NL06].



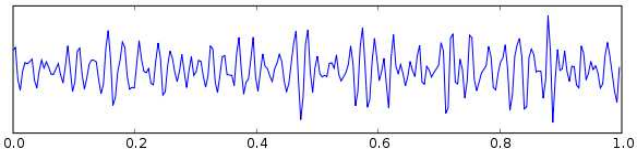
Pav. 13: Alfa bangos.

Beta bangos (14 – 30 Hz). Beta bangos dominuoja būdravimo būsenoje, esant atmerktoms akims, dėmesį fokusuojant į išorinį pasaulį arba sprendžiant konkrečias problemas. Beta bangos siejamos su koncentracija bei kognicija (pažinimas, suvokimas), stresu, karštligiška būsena, stresu bei susirūpinimu, nerimu [NL06].



Pav. 14: Beta bangos.

Gama bangos (30 – 128 Hz). Šių bangų įtaka žmogui nėra tiksliai nustatyta [NL06].

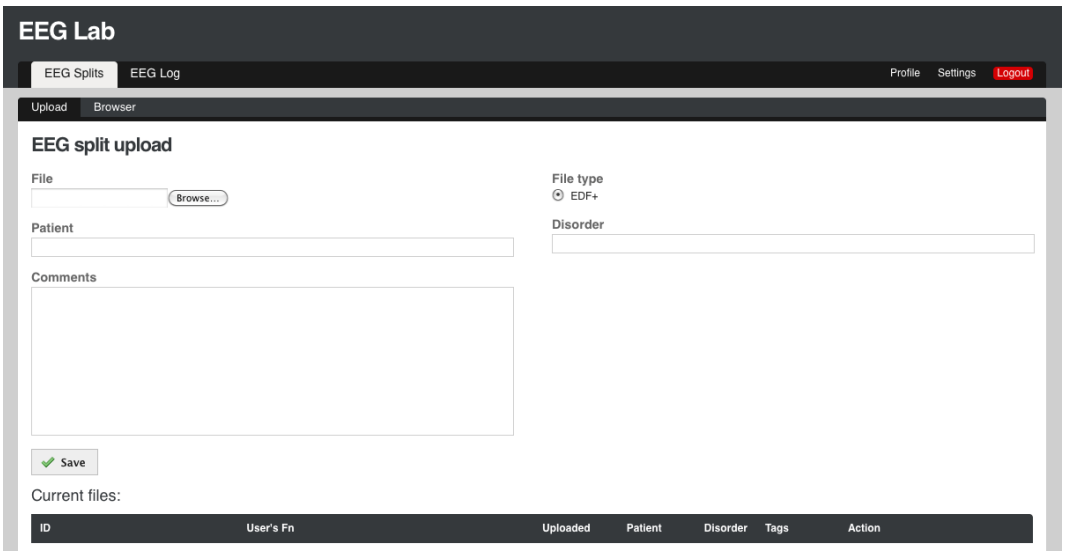


Pav. 15: Gama bangos.

Žinant elektroencefalogramas sudarančias smegenų bangas galima atlikti EEG analizę. Ši EEG analizė aprašyta sekančiame „EEG parametrizavimas“ skyriuje. Po analizės gaunama labai daug daugiamačių duomenų, kuriems galima pritaikyti duomenų gavybos algoritmus. Pritaikius šiuos algoritmus galima lengviau diagnozuoti epilepsiją bei kitus sveikatos sutrikimus.

9.2. EEG parametrizavimas

EEG duomenų analizei buvo pasinaudota Python programavimo kalba parašyta EEG Lab programa. Ši programa įvilkta į paprastą bei intuityvią vartotojo sąsają. Šios vartotojo sąsajos pagalba pasirenkamas *.edf formato failas, kuriame yra elektroencefalograma. Papildomi meta duomenys apie pacientą ir ligą yra taip pat užpildomi per programą. Failas ir meta duomenys išsaugomi duomenų bazėje.



Pav. 16: Programos vartotojo sąsaja.

Duomenų bazė ir skaičiavimai orientuojasi ties maža 20 sekundžių atkarpa. Ta iškarpa yra iš pilno EEG įrašo, į ją eina visi kanalai bei bangos. 20 sekundžių atkarpa nesunkiai telpa į kompiuterio ekraną - dėl to vizualiai ją lengva analizuoti ir jos „ištemptumas“ yra artimas tam, kuris naudojamas daugelyje vadovėlių.

Kiekviena atkarpa reprezentuoja savarankišką ir nepriklausomą EEG įrašo intervalą, jam yra nustatomi pikai, KDS (angl. Karolinska Drowsiness Score) ir kiti parametrai. Atkarpa yra atvaizduojama unikaliu dokumentu duomenų bazėje. Šis dokumentas yra JSON (angl. Javascript Object Notation) objektas. Jam priskiriamas unikalus identifikatorius (ID).

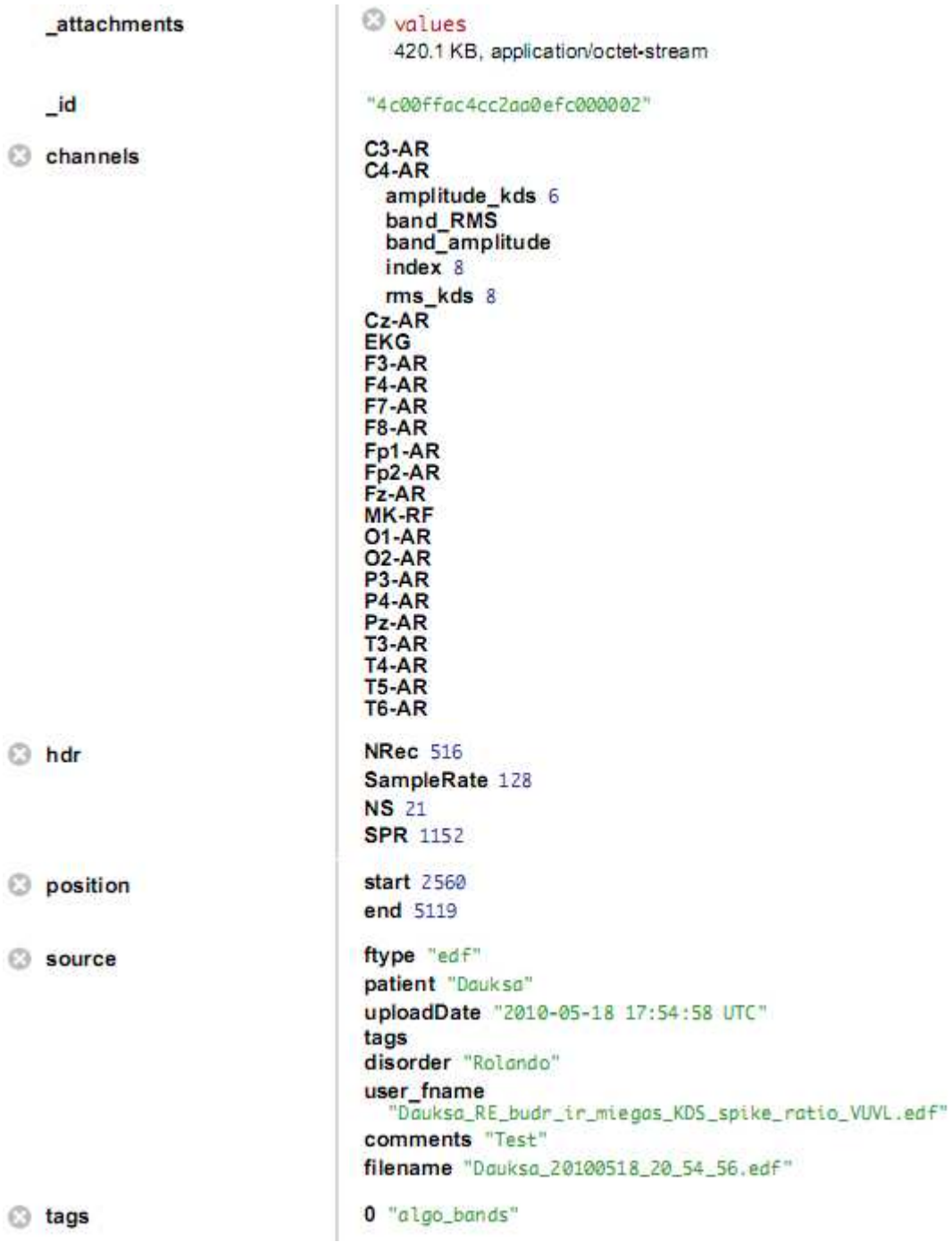
Pav. 17: Programa atvaizduoja EEG 20 sekundžių atkarpą.

9.2.1 EEG dokumento struktūra

Pagrindiniai elementai sudarantys duomenų bazės JSON objektą (dokumentą):

- **Attachments/values** – EEG duomenys išsaugoti dvimačiame numpy.ndarray masyve. Kiekviena eilutė tai kitas kanalas. Stulpeliai – signalo amplitudė tam tikru laiko momentu.
- **Hdr** - EEG duomenų failo antraštė, nukopijuota iš failo. Saugoma svarbi informacija apie duomenis, pvz. gavimo data, rinkimo dažnis (angl. SampleRate) bei kita.
- **Position** - 20 sekundžių atkarpos absoliuti pozicija visame EEG duomenų faile.
- **Source** - EEG duomenų šaltinis - paciento identifikacijos numeris, sutrikimo tipas, failo vardas, laikas.
- **Tags** – Atkarpos žymės. Nurodo kokie algoritmai buvo naudoti šiai atkarpai ir jų versijos, duomenų kokybę (ar nėra artefaktų, galinčių sugadinti analizės rezultatus).

- **Channels** - Informacija apie kanalus. Kiekvieno kanalo indeksas duomenų faile. Kanalo suskaičiuoti parametrai.



Pav. 18: EEG dokumento pavyzdys.

9.2.2 EEG duomenų gavyba

Išanalizuoti EEG duomenys patalpinami duomenų bazėje. Šie duomenys yra daugiamačiai ir jų yra labai daug, todėl reikalingas greitas duomenų išrinkimas. Tam tikslui, kad duomenis būtų

galima greitai surasti, bus pritaikytas R-medžio algoritmas. Duomenys bus pakrauti į R-medį ir tokiu principu panaudotas R-medis bus kaip indeksas duomenims. Turint R-medį galima atlikti duomenų klasterizavimą. Daugiamačiai duomenys skaitomi iš R-medžio ir klasterizuojami K-Means metodu. Gautas duomenų grupes galima analizuoti ir lyginti su kitomis. Pritaikius šiuos algoritmus galima lengviau diagnozuoti epilepsiją bei kitus sveikatos sutrikimus.

10. Atlikti darbai ir eksperimentai

Šiame darbe rašoma apie hierarchinę duomenų paieškos struktūrą R-medį, kuris skirtas daugiamačių duomenų indeksavimui, taip pat apie K-means ir CURE klasterizavimo algoritmus, bei apie elektroencefalogramas (EEG) jų kaupimą, analizę ir duomenis.

Pasitelkus šias žinias sukurta programa, kurioje realizuotos visos R-medžio funkcijos bei K-means ir CURE klasterizavimo algoritmai, ir pritaikyta EEG daugiamačių duomenų indeksavimui ir klasterizavimui. Norint geriau suprasti ir išanalizuoti minėtus algoritmus, programa parašyta nuo pradžių mano paties. Be to nuo pradžių parašyta programa lengviau pritaikoma iškeltiems tikslams ir poreikiams.

Prieš pradedant programuoti paruošti duomenys, kurie naudojami atliekamame eksperimente, tai yra išanalizuoti ir parametrizuoti elektroencefalogramų daugiamačiai duomenys. Šie duomenys buvo apdorojami „EEG Lab“ programa, kuri suskaičiuoja daug įvairių atributų ir parametrų. Tokie apdoroti duomenys yra pasiskirstę 6-matėje erdvėje. Toliau šie duomenys bus naudojami atliekant testus ir analizuojant R-medžio, K-means ir CURE algoritmų greitaveiką.

Programa pradėta realizuoti nuo R-medžio (R-medžios realizavimas aprašytas 4 skyriuje). Programa ištirta greitinimo požiūriu ir ne kartą buvo tobulinama, kad algoritmas vyktų ne tik greičiau ar efektyviau bet taip pat kad išgauti kuo geresnius rezultatus. Patobulinus R-medžio programą gauti tikrai puikūs spartos rezultatai. Pasiektas 50% ir daugiau spartesnis medžio užpildymas.

R-medžio tyrimai parodė, kad suprogramuotas medis puikiai tinka daugiamačių duomenų saugojimui. Daugiamačiai duomenys buvo sparčiai apdoroti ir suindeksuoti.

R-medžio privalumai būtų tokie:

- R-medžio indeksas yra labai mažas, jis užima mažai disko vietos ir sukūrimo bei palaikymo laikas yra mažas;
- Kilimas medžiu aukštyn yra optimalus greičio požiūriu;
- Greitai vykdomas medžio atnaujinimo algoritmas;
- R-medis laiko informaciją apie duomenų objektą lapo taške.

Sekančiu etapu programa papildyta K-means klasterizavimo algoritmu (K-means klasterizavimo algoritmo realizacija aprašyta 7 skyriuje). Suprogramavus K-means algoritmą ir atlikus analizę prieita išvados, kad šio algoritmo greitis yra labai priklausomas nuo klasterio

centrų išsidėstymo. To pasekoje sunku prognozuoti kiek laiko bus vykdomas algoritmas. Vieną kartą paleidus algoritmą rezultatai gali būti labai geri, o jau sekantį kartą - vykdymo laikas žymiai pailgėja. Taip pat bandymu metu gauti rezultatai parodė, kad didelę įtaką algoritmo greitaveikai turi klasterio centrų skaičius. Kuo daugiau klasterio centrų, tuo algoritmas darosi lėtesnis.

Nepaisant nenuspėjamo vykdymo laiko K-means klasterizavimo algoritmas išties yra greitas ir efektyvus. Jis greitai apdoroja duomenis ir juos suklasterizuoja.

Toliau trečiu etapu programa papildyta CURE klasterizavimo algoritmu (CURE klasterizavimo algoritmo realizacija aprašyta 8 skyriuje). Taigi suprogramavus šį algoritmą ir atlikus pirminius testus pastebėta, kad jis yra lėtesnis už K-means klasterizavimo algoritmą. Kadangi CURE algoritmas yra sudėtingesnis ir atlieka daugiau funkcijų bei patikrinimų, todėl jis ir yra lėtesnis.

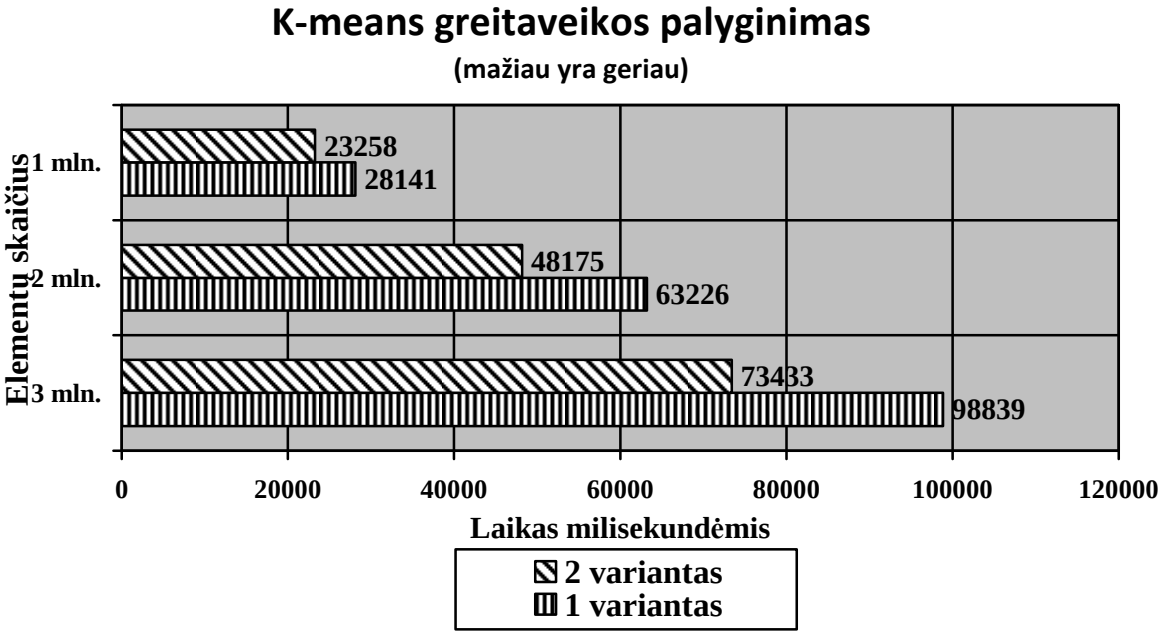
Vienas iš teigiamų CURE klasterizavimo algoritmo bruožų būtų tai kad jis tolygiai apdoroja duomenis. Taip pat šis algoritmas yra giriamas dėl gaunamų puikių rezultatų. Kaip teigiama [GRS01] jis puikiai klasterizuoja įvairių tipų duomenis.

Po programos sukūrimo buvo atlikti testai. Domino algoritmų greičiai ir kaip sparčiai jie vyksta. Taip pat norėta išsiaiškinti, kaip greičiau bus suklasterizuoti duomenys:

1. Ar tuomet kai duomenys skaitomi visi iš eilės;
2. Ar kai tie patys duomenys suindeksuoti R-medžio pagalba.

Visų pirma buvo atliekami K-means algoritmo testai. Atlikus analizę paaiškėjo, kad maži duomenų kiekiai klasterizuojami greičiau kai duomenys apdorojami visi iš eilės. Taip yra todėl, kad klasterizuojant pirmuoju variantu duomenys iš karto pradedami analizuoti ir klasterizuoti. Antruoju variantu visų pirma duomenis reikia suindeksuoti, o tik po to galima juos pradėti klasterizuoti. Taigi susumavus indeksavimo ir klasterizavimo laikus, antrasis variantas greičio atžvilgiu gaunasi lėtesnis, kai duomenų yra sąlyginai mažai (duomenų kiekis neviršija 1 milijono). Tačiau duomenų skaičius, su kuriais yra dirbama, tikrai nėra mažas ir gauti klasterizavimo rezultatai su mažu duomenų kiekiu nėra reikšmingi.

Toliau vykdant K-means algoritmo greičio testus paaiškėjo, kad antrasis variantas, t. y. visu pirma suindeksavus duomenis o tik po to juos klasterizuojant, su didesniais duomenų kiekiais yra greitesnis.



2 var. Greičiai / Elementų skaičius	Greitis lyginant su 1 var.	Greičiau už 1 var.	Vidutiniškai
1 mln.	82,64 %	17,36 %	22,29 %
2 mln.	76,19 %	23,81 %	
3 mln.	74,29 %	25,71 %	

Lentelė 9. Suindeksuotų duomenų procentinis greičio palyginimas.

Atlikus eksperimentą gauti rezultatai įrodė, kad dirbant su dideliais duomenų kiekiais prieš klasterizuojant duomenis K-means algoritmu juos reikia suindeksuoti. Tuomet duomenų apdorojimas vyksta 22% greičiau.

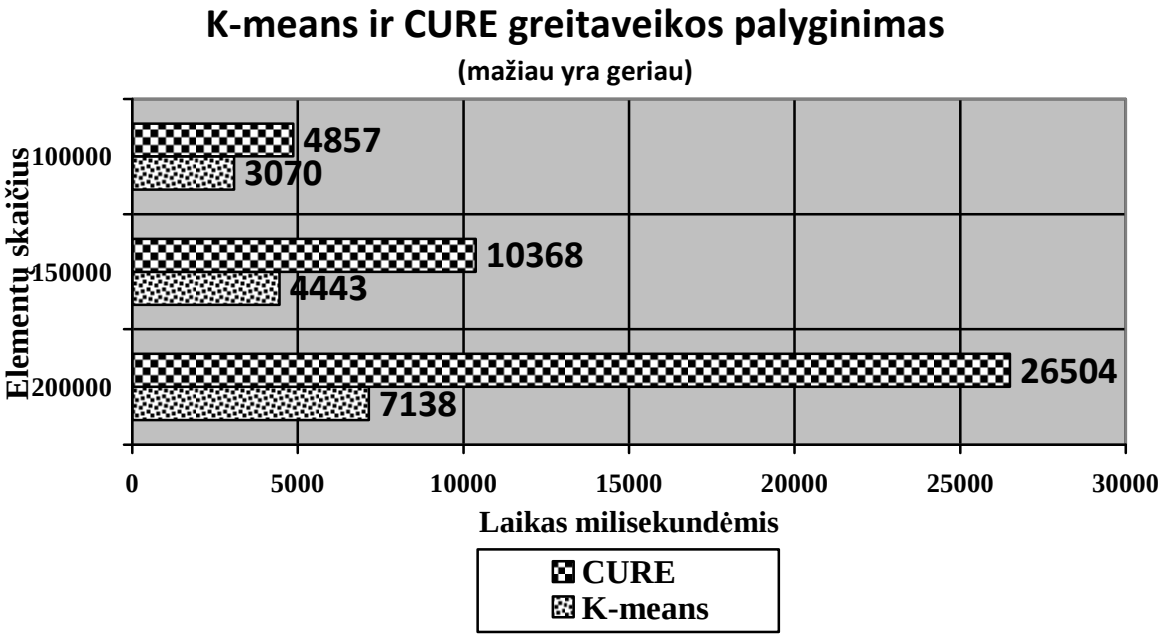
Toliau lygiai tokie pat eksperimentai buvo atlikti ir su CURE klasterizavimo algoritmu. Klasterizuojant duomenis pirmuoju ir antruoju variantais gautas didžiulis skirtumas. Galima sakyti gauti visiškai priešingi eksperimento rezultatai, nei prieš tai vykdyto su K-means algoritmu.

Elementų skaičius / Variantas	5 000	10 000	20 000
1 variantas	16390 ms	74688 ms	234797 ms
2 variantas	78 ms	312 ms	1203 ms

Lentelė 10. CURE klasterizavimo algoritmo greičiai.

Iš pateiktų duomenų 10 lentelėje matyti, kad paprastai duomenis iš eilės klasterizuojant algoritmas vykdomas labai lėtai. Tai tikrai didelis šio algoritmo minusas. Tačiau CURE klasterizavimo algoritmas tikrai gerai klasterizuoja duomenis (kiek tai buvo galima nustatyti žiūrint į skaičius, nes atvaizduoti duomenis sudėtinga kadangi jie pasiskirstę 6-matėje erdvėje). Iš tikro abu tiek K-means, tiek CURE algoritmai pateikia gerus rezultatus, bet truputi geresni yra CURE rezultatai. Ne veltui jis už tai yra giriamas.

Toliau nuspręsta palyginti koks gi iš tiesių greitaveikos skirtumas yra tarp K-means ir CURE algoritmų. Kadangi iškarto matyti, kad CURE algoritmas yra lėtesni, tai bus lyginama kuris algoritmas greičiau suklasterizuos jau suindeksuotus duomenis. Abu algoritmai indeksuotus duomenis apdoroja greitai. Taigi žemiau pateiktoje diagramoje atvaizduojama per kiek laiko buvo suklasterizuoti 100000, 150000 ir 200000 duomenų.



Iš pateiktos diagramos matyti, kad CURE algoritmas nusileidžia savo sparta K-means algoritmui. Maži duomenų kiekiai apdorojami sąlyginai vienodai, tačiau kuomet duomenų yra 100000 ir daugiau tai K-means algoritmas veikia greičiau. Taip pat reiktų paminėti kad didėjant duomenų kiekiui CURE algoritmo greitis sparčiai didėja.

Išvados ir rekomendacijos

Šiame darbe išanalizuoti hierarchinės duomenų paieškos R-medžio, K-means ir CURE klasterizavimo algoritmai, bei elektroencefalogramos (EEG) jų kaupimas, analizė ir duomenys.

Šio darbo rezultatas sukurta programa, kurioje realizuotos visos R-medžio funkcijos bei K-means ir CURE klasterizavimo algoritmai, ir pritaikyta EEG daugiamačių duomenų indeksavimui ir klasterizavimui. Suprogramuotas R-medis ištirtas, tyrimai parodė, kad R-medis puikiai tinka daugiamačių duomenų saugojimui. Duomenys buvo sparčiai apdoroti ir suindeksuoti.

R-medžio privalumai būtų tokie:

- R-medžio indeksas yra labai mažas, jis užima mažai disko vietos ir sukūrimo bei palaikymo laikas yra mažas;
- Kilimas medžiu aukštyn yra optimalus greičio požiūriu;
- Greitai vykdomas medžio atnaujinimo algoritmas;
- R-medis laiko informaciją apie duomenų objektą lapo taške.

Suprogramavus K-means algoritmą ir atlikus analizę prieita išvados, kad šio algoritmo greitis yra labai priklausomas nuo klasterio centrų išsidėstymo. Taip pat bandymu metu gauti rezultatai parodė, kad didelę įtaką algoritmo greitaveikai turi klasterio centrų skaičius. Kuo daugiau klasterio centrų, tuo algoritmas darosi lėtesnis. Nepaisant K-means klasterizavimo algoritmo minusų jis išties yra greitas ir efektyvus, greitai apdoroja duomenis ir juos suklasterizuoja.

Toliau suprogramuotas CURE algoritmas. Šis algoritmas yra giriamas dėl gaunamų puikių rezultatų. Jis puikiai klasterizuoja įvairių tipų duomenis. Tačiau, nors jis ir yra giriamas, šis algoritmas turi vieną pagrindinį minusą. Atlikti eksperimentai parodė kad jis yra lėtas.

Po programos sukūrimo buvo atlikti testai. Domino algoritmų greičiai ir kaip sparčiai jie vyksta. Atlikus analizę paaiškėjo, kad maži duomenų kiekiai K-means algoritmu klasterizuojami greičiau kai duomenys apdorojami visi iš eilės. Taip yra todėl, kad duomenys iš karto pradedami analizuoti ir klasterizuoti. Kitu atveju visų pirma duomenis reikia suindeksuoti, o tik po to galima juos pradėti klasterizuoti. Taigi susumavus indeksavimo ir klasterizavimo laikus, antrasis variantas greičio atžvilgiu gaunasi lėtesnis, kai duomenų yra mažai (duomenų kiekis neviršija 1 milijono).

Toliau vykdant K-means greičio testus paaiškėjo, kad visu pirma suindeksavus duomenis o tik po to juos klasterizuojant, su didesniais duomenų kiekiais algoritmas yra greitesni. Suindeksuotų duomenų klasterizavimas vyksta 22% greičiau.

Vykdant CURE algoritmo testus paaiškėjo, kad jis yra labai lėtas. Tačiau pritaikius R-medžio indeksavimą rezultatai labai pagerėjo, nors to nepakako kad būtų aplenktas K-means algoritmas. Visgi K-means algoritmas yra greitesnis nei CURE, tačiau CURE algoritmas pateikia truputi geresnius klasterizavimo rezultatus.

Atlikta analizė ir testai įrodė, kad R-medis puikiai tinka daugiamačių duomenų indeksavimui, o K-means klasterizavimo algoritmas yra efektyvus ir spartus. Įvykdžius testus gauta išvada, kad dirbant su dideliais duomenų kiekiais prieš klasterizuojant duomenis geriau juos suindeksuoti. Tačiau jei duomenų nėra daug, tai indeksavimo neprireiks.

Literatūros sąrašas

[BKS+90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, “The R*-tree: An Efficient and Robust Access Method for Points and Rectangles,”. 1990.

[Ben75] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching, 1975.

[GRS97] Sudipto Guha, Rajeev Rastogi, Kyuseok Shim. CURE: A clustering algorithm for large databases, 1997, pp. 73-84.

[GRS01] Sudipto Guha, Rajeev Rastogi, Kyuseok Shim. CURE: An Efficient Clustering Algorithm for Large Databases, 2001, pp. 35–58

[Gun86] O.Gunther. The cell tree: An index for geometric data. University of California, 1986, Berkeley.

[Gut84] A.Guttman. R-trees: A dynamic index structure for spatial searching. ACM SIG-MOD, 1984, pp.47-57.

[HM04] S. Har-Peled, S. Mazumdar. Coresets for k-means and k-median clustering and their applications. 2004, pp. 291-300.

[HS93] E.G.Hoel and H.Samet. Data-parallel R-tree algorithms. 23rd Int’l. Conf. on Parallel Processing, 1993.

[HJR97] Y-W. Huang, N. Jing, E. A. Rundensteiner. Spatial Joins Using R-Trees: Breadth-First Traversal with Global Optimizations. 1997.

[Jag90] H. V. Jagadish. Linear Clustering of Objects with Multiple Attributes. May 1990, Atlantic City.

[Juo07] A. Juozapavičius. Duomenų struktūros ir efektyvūs algoritmai. 2007.

[KF92] I.Kamel and C.Faloutsos. Parallel R-tree. University of Maryland, 1992.

[KF93] I.Kamel and C.Faloutsos. On packing R-tree. 2nd Int'l. Conf. on Information and Knowledge Management, 1993, pp.490-499.

[KMN+02] T. Kanungo, D. M. Mount, N. S. Netanyahu, C, D, Piatko, R. Silverman, A. Y. Wu. A local search approximation algorithm for k-means clustering. 2002, pp. 8-18.

[Mac67] J. MacQueen. Some Methods for classification and Analysis of Multivariate Observations. University of California, 1967.

[NL05] Ernst Niedermeyer and Fernando Lopes da Silva. Electroencephalography: Basic Principles, Clinical Applications, and Related Fields. 2005.

[NL06] Ernst Niedermeyer, Fernando Lopes da Silva. Electroencephalography. 5th Edition, 2006.

[NH94] R. Ng and J. Han. Efficient and Effective Clustering Method for Spatial Data Mining. 1994.

[Pol81] D. Pollard. Strong consistency of k-means clustering. 1981, pp. 135-140.

[RL85] N.Roussopoulos and D.Leifker. Direct spatial search on pictorial databases using packed R-tree. ACM SIGMOD, 1985.

[SRF87] T. Sellis, N. Roussopoulos, and C.Faloutsos. The R+-Tree: A Dynamic Index For Multi-Dimensional Objects. September 1987, Brighton.

[THB+07] William O. Tatum, Aatif M. Husain, Selim R. Benbadis, and Peter W. Kaplan. Handbook of EEG interpretation. Demos Medical Publishing, 2007.