**Univerzita Pavla Jozefa Šafárika v Košiciach,
Prírodovedecká fakulta**

Ústav informatiky

Diplomová práca

Indexovanie a triedenie v metrických priestoroch

Košice 2008

Martin Šumák

Vyhlásenie

Vyhlasujem, že diplomovú prácu som vypracoval samostatne a na základe vedomostí získaných štúdiom. Všetká použitá literatúra je uvedená na konci práce.

Košice, apríl 2008 Martin Šumák

Abstrakt

Súčasné algoritmy na vyhľadávanie najlepších *n* objektov nemajú podporu atribútov vyjadrujúcich vzdialenosť objektov od používateľom zvoleného bodu v metrickom priestore. Optimálne vyhľadávacie techniky vyžadujú zasielanie objektov a hodnôt ich atribútov v prúde pre každý atribút v poradí od najlepších po najhoršie. Štandardné typy vyhľadávania nad metrickými priestormi, ako sú rozsahové dopyty alebo kNN dopyty, toto neumožňujú. V tejto práci predstavíme prúdovú verziu metódy najbližších susedov a jej vylepšenie umožňujúce definovať preferenciu nie len pre blízke, ale aj pre stredne vzdialené či vzdialené objekty od zvoleného fixovaného bodu. Algoritmy pre uvedené dopyty sú navrhnuté nad silnou metrickou indexovacou štruktúrou M-strom. Druhá časť práce je venovaná vytváraniu kvalitnej indexácie pomocou M-stromu. Okrem známych spôsobov ako sú algoritmus Slim down a Multi way leaf choice predstavíme techniku virtuálnych stredov uzlov a ich dynamického prepočítavania.

Abstract

Current algorithms of searching the top *n* objects include no support for attributes that express object's distance from user defined point in metric space. Optimal search techniques require for each attribute a stream of objects ranked from the best to the worst with respect to user preferences. Standard algorithms over metric spaces for range queries or kNN queries do not offer a stream as a result. In this work we propose the stream version of nearest neighbours queries and its improvement with support of the preferences for middle distance objects as well as for far objects from the fixed anchor point. All the algorithms processing mentioned queries are designed for powerfull metric indexing structure – the M-tree. Second part of this work is dedicated to creating a high-quality M-tree index. Except the well-known techniques like algorithms Slim down and Multi way leaf choice, we propose a technique for virtual centers of tree nodes and its dynamic recomputing.

Obsah

1 Úvod

1.1 Vyhľadávanie najlepších *n* objektov

1.2 Triedenie bodov metrického priestoru podľa vzdialenosti

2 Indexovacie štruktúry pre body metrického priestoru

2.1 Prehľad existujúcich indexovacích štruktúr

2.2 M-strom

2.3 Vytváranie M-stromu

3 Dopyty nad M-stromom

3.1 Existujúce dopyty vs. požadovaný dopyt

3.2 Rozsahový dopyt – algoritmus range query

3.3 Dopyt na *k* najbližších susedov – algoritmus kNN query

3.4 Sekvenčný dopyt na najbližších susedov – algoritmus Sort query

3.5 Algoritmus Fuzzy sort query

4 Kvalita M-stromu

4.1 Význam kvality M-stromu

4.2 Meranie kvality M-stromu

4.3 Algoritmus Node sieve

4.4 Algoritmus Multi way leaf choice

4.5 Algoritmus Generalized slim down

4.6 Virtuálne stredy uzlov

4.7 Dynamické prepočítavanie stredov

5 Praktická časť – návrh programu

5.1 Požiadavky na aplikáciu

5.2 Návrh knižnice pre M-strom

5.3 Implementácia

6 Testy

6.1 Testy výkonnosti dopytov

6.2 Testy kvality M-stromu

7 Záver

Literatúra

Prílohy

A Používateľská príručka k programu MTreeViewer

B Vzorové implementácie rozhraní `MIndexable` a `FuzzyEvaluator`

C Vzorová ukážka použitia M-stromu

1 Úvod

Táto práca je venovaná indexovaniu bodov metrického priestoru pomocou M-stromu, dopytom nad M-stromom a kvalite indexácie v M-strome. Motiváciou zdokonaľovanie systému pre vyhľadávanie najlepších *n* objektov, ktorému som sa venoval v bakalárskej práci [5]. Cieľom tohto systému je nájsť z veľkého množstva objektov takých *n* objektov, ktoré najviac vyhovujú preferenciám používateľa.

Prvá časť práce je venovaná tvorbe M-stromu a dopytom nad ním. Najprv sú rozobrané známe dopyty – rozsahový dopyt (range query) a dopyt na *k* najbližších susedov (kNN query), ktoré neriešia kľúčový problém. Pre riešenie problému som navrhol nový algoritmus Sort query, ktorý definuje nový typ dopytu – tzv. sekvenčný dopyt na najbližších susedov. Následne je predstavená aj jeho fuzzy verzia, ktorá prináša riešenie aj pre veľmi špecifické preferencie používateľa vyjadrené fuzzy funkciou. V testoch je experimentálne ukázaná výborná efektivita tohto algoritmu, na ktorú nemá vplyv ani distribúcia dát, ani preferencie používateľa.

Druhá časť práce je venovaná kvalite indexácie v M-strome. Opäť sú na začiatku rozobrané existujúce prístupy, ktorými sú algoritmy Generalized slim down a Multi way leaf choice. Následne predstavujem vlastné pristupy t.j. virtuálne stredy uzlov M-stromu a ich dynamické prepočítavanie. Je nutné dodať, že experimenty ukázali, že tieto prístupy nepriniesli sľubovaný efekt, no výsledkom je skúsenosť, že pokusy o zlepšovanie kvality indexácie v M-strome sa musia uberať iným smerom.

Prílohy na konci práce a na priloženom CD obsahujú javovskú knižnicu mindex.jar poskytujúcu plnú funkcionalitu M-stromu, jej zdrojové kódy, vygenerovanú javovskú webovú dokumentáciu a tiež vzorové príklady jej použitia. Ďalej v prílohe nájdete program MTreeViewer pre vizualizáciu M-stromu a používateľskú príručku k tomuto programu.

1.1 Vyhľadávanie najlepších *n* objektov

Množstvo spracovávaných dát v počítačovej sieti neustále rastie, a preto sa pri vyhľadávaní čoraz viac využívajú dopyty, ktoré vrátia používateľovi len niekoľko preňho najlepších objektov. Takéto dopyty predstavujú problém vyhľadávania najlepších *n* objektov nad distribuovanými dátami. Z tohoto problému vychádza téma, ktorej je venovaná táto práca, preto najprv priblížim problém vyhľadávania najlepších *n* objektov.

Objektom rozumieme nejaký predmet záujmu, na ktorý kladieme konkrétne požiadavky a podľa nich chceme nájsť najlepší možný objekt z ponuky. Použijem príklad s hotelmi. Chystáme sa na dovolenku do Tatier a hľadáme vhodný hotel. Kladieme naňho konkrétne požiadavky, napríklad nízku cenu, dobrú kvalitu služieb a samozrejme vhodnú polohu. Chceme, aby bol niekde v cieľovej lokalite. Z uvedeného príkladu je zrejmé, že sa

nerozhodujeme podľa jednej vlastnosti, ale v hre je niekoľko krítérií rozhodovania. Pri širokej ponuke hotelov to nie je triviálny problém, preto vznikla potreba použitia informačného systému. Základom takého systému sú dáta a algoritmy na vyhľadávanie. Kým v bakalárskej práci som sa zaoberal algoritmami, teraz sa budem bližšie venovať dátam. Popíšeme si teda ich štruktúru, ktorá je základným predpokladom pre vyhľadávacie algoritmy.

Ako bolo spomenuté, pri vyhľadávaní najlepších *n* hotelov ide o multikriteriálne rozhodovanie o tom, ktorý hotel je lepší. V našom príklade sa obmedzíme na tri atribúty hotelov, ktoré budeme sledovať, a podľa ktorých sa budeme rozhodovať. Budú to cena, kvalita služieb a vzdialenosť od cieľovej lokality. Dáta budú tvorené troma zoznamami hotelov. V jednom zozname budú ceny hotelov, v druhom nejaké ohodnotenie kvality ich služieb (napr. počet hviezdičiek) a v treťom vzdialenosti hotelov od cieľovej lokality. Podotýkam, že sú to logické zoznamy, teda fyzicky môžu byť dáta uložené v ľubovoľnej dátovej štruktúre. Predpokladom algoritmov je, že dáta (tie tri zoznamy) sú distribuované, normované, zotriedené a poskytujú určite aspoň sekvenčný prístup, v lepšom prípade aj priamy prístup.

Najprv sa zastavme pri distribuovanosti. Dáta nemusia byť uložené na jednom počítači v jednej spoločnej databáze, prípadne v nejakej inej rovnakej spoločnej štruktúre. Jeden zoznam môže byť v databáze na lokálnom počítači, druhý na vzdialenom serveri, ku ktorému máme prístup prostredníctvom internetu a tretí podobne na inom serveri a v úplne inej štruktúre, napr. v textovom súbore.

Ďalšou podmienkou je, že dáta sú normované. Na úrovni reálnych dát je cena vyjadrená v slovenských korunách a kvalita služieb vyjadrená počtom hviezdičiek. Sú to hodnoty zrozumiteľné pre človeka, ale už menej zrozumiteľné pre počítač a algoritmy. Tie potrebujú pre nájdenie *n* najlepších hotelov jednotlivé hotely konkrétne ohodnotiť, aby vedeli pre ľubovoľné dva hotely rozhodnúť, ktorý je lepší. Pri ohodnocovaní musia zobrať do úvahy hodnoty hotelov vo všetkých sledovaných atribútoch. Nebudem sa ďalej zaoberať spôsobom ohodnocovania objektov, postačí nám vedomosť o tom, že pre korektné ohodnocovanie musia byť dáta normované. Zoznamy teda obsahujú normované hodnoty, t.j. všetky ceny, ohodnotenia kvality a vzdialenosti sú prepočítané na reálne čísla z intervalu <0;1>. Takto normovaná hodnota potom vyjadruje mieru spokojnosti používateľa, t.j. nakoľko daný objekt spĺňa jeho požiadavky v danom atribúte. Čím je normovaná hodnota vyššia, tým je objekt v danom atribúte lepší. Hodnota 1 znamená, že daný objekt má danú vlastnosť presne podľa používateľových požiadaviek, naopak 0 znamená, že objekt používateľové požiadavky v danom atribúte vôbec nespĺňa. Vráťme sa k nášmu príkladu s hotelom v Tratrách. Čím je hotel lacnejší, tým vyššiu má normovanú hodnotu ceny, čím je kvalitnejší tým vyššiu má normovanú hodnotu kvality. Analogicky je to so vzdialenosťou, čím bližsie je hotel k cieľovej lokalite, tým vyššia je jeho normovaná hodnota vzdialenosti. Toto normovanie realizuje jednoduchá lineárna funkcia. Existujú aj iné možnosti prepočtu reálnych hodnôt na normované, zatiaľ budeme uvažovať tieto.

Ďalšia algoritmami predpokladaná vlastnosť dát je zotriedenosť. Jednotlivé zoznamy sú zotriedené podľa normovaných hodnôt zostupne, t.j. od najlepšieho po najhorší. To, že dáta sú tvorené zotriedenými zoznamamy, je len logický pohľad na nich. Zotriedenosť zaručuje sekvenčný prístupu k nim. Pomocou neho algoritmy čítajú jednotlivé zoznamy v poradí od najlepšieho po najhorší podľa používateľových preferencií. Sekvenčný prístup je základný prístup, ktorý musí použitá dátová štruktúra poskytovať. Prvým volaním sekvenčného prístupu do jedného logického zoznamu získame najvačšiu normovanú hodnotu príslušného atribútu nachádzajúcu sa v danom zozname a zároveň identifikátor (napr. meno) príslušného hotela. Vo všeobecnosti *i*-tym volaním sekvenčného prístupu získame *i*-tu najväčšiu normovanú hodnotu príslušného atribútu a zároveň identifikátor príslušného hotela. Sekvenčný prístup je znázornený na obrázku 1.1 v zozname vzdialeností.

cena kvalita vzdialenosť

Hilton 0,90 ... i - ty
Royal 0,85
Plazza 0,88 ... i + 1 - vy
Ritz 0,80

Jazero 0,23

Obr. 1.1 Odpoveď sekvenčného (pri vzdialenosti) a priameho (pri cene) prístupu znázorňuje rámček.

V zozname cien je na obr. 1.1 znázornený priamy prístup, ktorý je nepovinný. Žiadna dátová štruktúra ho nemusí poskytovať. Priamym prístupom rozumieme odpoveď na otázku typu: „Akú normovanú hodnotu ceny má hotel Royal?“. Je to jednoduché priame zistenie normovanej hodnoty konkrétneho objektu v konkrétnom atribúte.

Algoritmy vyhľadávajú nad dátami práve pomocou sekvenčných a priamych prístupov. Nebudem bližšie popisovať algoritmy, postačí nám vedomosť, že každý zoznam postupne čítajú sekvenčnými prístupmi. Kedykoľvek, bez toho aby prečítali celé zoznamy môžu nájsť *n* najlepších objektov a výpočet ukončiť. Niekedy stačí, že prečítaju z nejakého zoznamu zopár prvých hodnôt, inokedy môžu prečítať celý zoznam až do konca. Pre rýchle vyhľadávanie najlepších *n* objektov treba implementovať čo najrýchlejší sekvenčný a podľa možností aj priamy prístup. V čase otázky na najlepších *n* objektov už nie je únosné jednotlivé zoznamy začať triediť. Je dobré mať tieto zoznamy už vopred zotriedené podľa normovaných hodnôt a v čase otázky len ich začať čítať. Ceny hotelov a kvalitu služieb môžeme považovať vzhľadom na otázku za fixné. Ak navyše predpokladáme, že každý hľadá čo najlepšiu cenu a čo najvyššiu kvalitu, tak si môžeme dopredu pripraviť zotriedené zoznamy pre normované hodnoty ceny a kvality služieb. Za uvedených predpokladov, tieto hodnoty nezávisia od otázky na najlepších *n* objektov. So vzdialenosťou je problém. Lokalita, v ktorej hľadáme hotel je jedným z parametrov otázky. Normované hodnoty vzdialeností hotelov sú teda známe až v čase otázky. Vzdialenosť môžeme považovať za metrický atribút a jeho zoznam nie je zotriediteľný vopred. Napriek tomu potrebujeme aj nad týmto atribútom rýchly sekvenčný prístup. Práve týmto problémom na všeobecnej úrovni sa zaoberá táto práca.

1.2 Triedenie bodov metrického priestoru podľa vzdialenosti

Abstrahujme od akýchkoľvek ďalších informácií o hoteloch, ktoré sa bezprostredne netýkajú ich polohy. Potom hotely, spomedzi ktorých vyhľadávame, sú pre nás body na

zemskom povrchu, medzi ktorými dokážeme vypočítať vzdialenosť. Také body sú určené GPS súradnicami a metrikou je potom najkratšia vzdialenosť dvoch bodov na guľovom povrchu. Štandardnejším príkladom metrického priestoru je euklidovská rovina, alebo euklidovský priestor. Vzdialenosťou medzi bodmi nemusí byť len ich fyzická odľahlosť, ale napríklad aj čas potrebný na prepravu medzi nimi. Na príklade s bodmi na zemskom povrchu, alebo v rovine, či priestore, si stále dokážeme predstaviť nejaké konkrétne polohy nejakých bodov. Preto poďme s abstrakciou ešte ďalej. Predstavme si, že naše objekty, spomedzi ktorých vyhľadávame, sú obrázky. Je zrejmé, že aj medzi obrázkami je určitá vzdialenosť, lepšie povedané podobnosť. Niektoré obrázky sa podobajú viac, iné menej. Takýmto objektom už ťažšie prisúdime nejakú pozíciu niekde v priestore. Predpokladáme však, že stále máme možnosť vypočítať vzdialenosť medzi dvoma obrázkami. Môžeme teda chcieť zotriediť skupinu obrázkov od najpodobnejšieho po najrozdielenjší od konkrétneho zvoleného obrázku.

Ak abstrahujeme od akýchkoľvek konkrétnych predstáv, dôjdeme k tomu, že jedinou znalosťou, ktorú vieme bez ujmy na všeobecnosti použiť, je vzdialenosť medzi ľubovoľnými dvoma objektami. Na takto abstraktnej úrovni potrebujeme mať objekty uložené v nejakej vhodnej indexovacej štruktúre, aby bolo možné vykonávať rýchly sekvenčný prístup. Požiadavkou je, aby sekvenčný prístup postupne vrátil objekty v poradí od najlepšieho po najhorší podľa ich vzdialeností od konkrétneho fixovaného objektu. Tento fixovaný objekt budeme nazývať kotvový bod. Aby bolo možné navrhnúť vhodnú indexovaciu štruktúru, musia medzi jednotlivými vzdialenosťami platiť určité pravidlá. Zadefinujme si preto metrický priestor (tak ako je bežne definovaný).

Def.: Nech **M** je neprázdna množina, nech funkcia d: $\mathbf{M}\times\mathbf{M} \rightarrow \mathbf{R}$, má nasledujúce vlastnosti:

$\forall x, y, z \in \mathbf{M}$ platí:

1. $d(x, y) \geq 0$ (nezápornosť)
2. $d(x, y) = 0 \leftrightarrow x = y$ (reflexivita)
3. $d(x, y) = d(y, x)$ (symetria)
4. $d(x, y) \leq d(x, z) + d(z, y)$ (trojuholníková nerovnosť)

Potom dvojicu (**M**, d) nazývame metrický priestor a funkciu d nazývame metrika. Prvky množiny **M** nazývame body metrického priestoru.

Objekty budeme reprezentovať bodmi metrického priestoru a funkcia na výpočet vzdialenosti medzi takými dvoma bodmi musí byť metrikou.

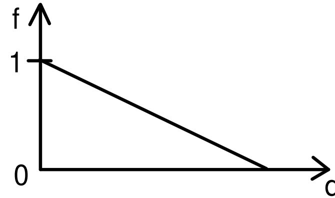
Reflexivita platí medzi bodmi metrického priestoru, nemusí však platiť medzi objektami, ktoré sú tými bodmi reprezentované. Ak dva rôzne objekty reprezentujeme dvoma rovnakými bodmi, tak tieto objekty majú nulovú vzdialenosť. Vidíme však, že ak majú dva objekty nulovú vzdialenosť, neznamená to, že je to jeden a ten istý objekt. Jednoducho sú tieto objekty reprezentované tým istým bodom a teda len z pohľadu metrického priestoru nie je medzi nimi rozdiel.

Na indexovaciu štruktúru kladieme nasledujúce požiadavky. Štruktúra musí byť nezávislá na konkrétnom metrickom priestore a uchovávaných bodoch. Predpokladáme veľké množstvo indexovaných bodov, preto požadujeme stránkovanie dát na pevný disk.

To čo môžeme pre takúto štruktúru zaručiť je, že funkcia na výpočet vzdialenosti medzi bodmi je metrika.

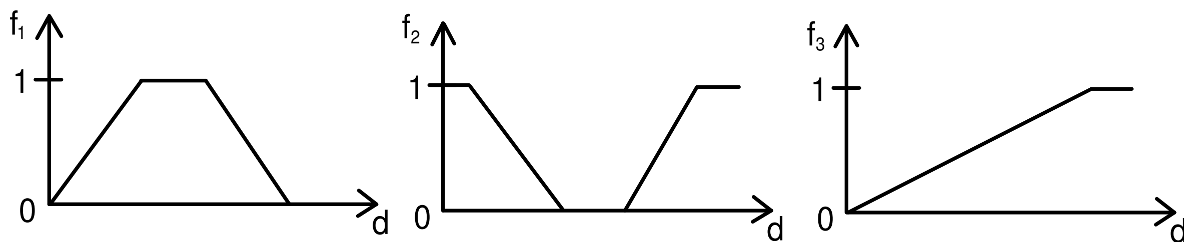
Nad indexovacou štruktúrou potrebujeme následne navrhnúť a implementovať efektívny a rýchly sekvenčný prístup. Dáta, ktoré algoritmom poskytuje sekvenčný prístup musia byť normované. Najprv uvažujme, že čím je bod bližšie ku kotvovému bodu, tým je

podľa našich požiadaviek lepší. Potom stačí, aby sekvenčný prístup vrátil body v poradí od najbližšieho po najvzdialenejší od kotvového bodu. Normovanie ich vzdialeností stačí vykonať až po získaní hodnoty sekvenčným prístupom a správnosť poradia sa zachová. Ak navyše body vzdialenejšie ako nejaká akceptovateľná vzdialenosť považujeme za úplne nevyhovujúce, normovanie môžeme vykonať fuzzy funkciou na obrázku 1.2.



Obr.1.2 Klesajúca fuzzy funkcia, na x-ovej osi je vzdialenosť od kotvového bodu.

Nechceme sa však obmedziť len na predpoklad, že používateľ bude vždy považovať bližšie body za lepšie. Chceme mu poskytnúť možnosť vyjadriť svoje preferencie voči vzdialenosti od kotvového bodu. Parmetrom otázky bude teda aj samotná fuzzy funkcia, ktorá ohodnotí vzdialenosti od kotvového bodu podľa používateľových predstáv. Typické príklady fuzzy funkcií znázorňuje obrázok 1.3.



Obr.1.3 Príklady iných fuzzy funkcií.

Napríklad fuzzy funkcia f_1 na obrázku 1.3 vyjadruje preferenciu stredných vzdialeností. Je to typický prípad, keď pracujeme v centre veľkomesta a hľadáme si byt alebo dom. Kotvovým bodom je centrum mesta, ale v centre bývať nechceme. Preferujeme periférnu obytnú oblasť mesta vzdialenú od centra približne 5 až 10 km. Vzdialenejšie byty a domy opať klesajú na zaujímavosti, lebo nechceme každý deň dochádzať z takej diaľky (a možno tam už nechodí ani MHD).

V prípade, že fuzzy funkcia je jedným z parametrov otázky, musí už samotný sekvenčný prístup poskytovať body v poradí od najlepšieho po najhorší podľa príslušnej fuzzy funkcie.

Takéto druhy dopytov nad metrickými atribútmi sú doteraz nové. V kapitole 3 uvediem nový algoritmus, ktorý umožňuje výpočet takýchto dopytov veľmi efektívne a porovnám ho s existujúcimi technikami, ktoré boli nevyhovujúce. V druhej časti práce sa budem venovať kvalite indexácie, aby sa dosiahol čo najvyšší výkon dopytov. Použitou indexovacou štruktúrou bude M-strom a uvediem niekoľko nových prístupov a vylepšení indexácie v M-strome.

2 Indexovacie štruktúry pre body metrického priestoru

V tejto časti najprv predstavíme niekoľko existujúcich indexovacích štruktúr určených pre indexvanie bodov metrického priestoru. Hlavnou štruktúrou, ktorej sa budeme venovať, je

M-strom. Podrobne popíšeme jeho vlastnosti, výhody voči ostatným štruktúram a spôsob jeho vytvárania.

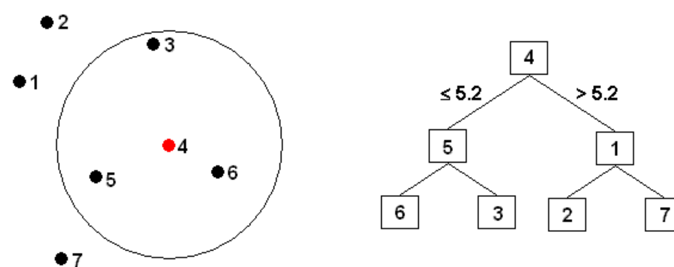
2.1 Prehľad existujúcich indexovacích štruktúr

Existuje množstvo rôznych, prevažne stromových indexovacích štruktúr. Podľa typu metriky, pre ktorý sú navrhnuté ich môžeme rozdeliť do dvoch skupín. Jednu skupinu tvoria stromy navrhnuté pre diskrétnu metriku, druhú skupinu tvoria stromy pre spojitú metriku. Stromy pre diskrétnu metriku vyžadujú, aby oborom hodnôt vzdialenostnej funkcie bola relatívne malá množina niekoľkých hodnôt. Tento predpoklad nemôžeme zaručiť, teda tieto stromy sú pre náš problém nevyhovujúce. Spomeniem len, že ide o tzv. BKT (Burkhard-Keller Tree) [9] a ďalšie typy, ktoré z nich vychádzajú. Dôležitou skupinou pre náš problém sú stromy navrhnuté pre spojitú metriku. Stromy navrhnuté pre spojitú metriku sú rovnako použiteľné aj pre diskrétnu metriku. Jednoducho nekladú na vzdialenostnú funkciu žiadne ďalšie požiadavky.

Na jednotlivých stromoch budeme v prvom rade sledovať, či spĺňajú naše požiadavky, t.j. nezávislosť od metrického priestoru a stránkovanie dát na pevný disk. Ďalšie sledované vlastnosti budú možnosť dynamického pridávania bodov a vyváženosť.

Vyvážený strom má bez ohľadu na dáta v ňom uložené vždy všetky vetvy rovnako dlhé. Za každých okolností je zaručené, že výška stromu stúpa s počtom jeho bodov logaritmicky. Nemôže sa teda v dôsledku špecificky zvolených dát stať, že sa strom vybuduje nevhodne (zopár krátkych vetiev a jedna veľmi dlhá vetva a pod.) a my prídeme o efektivitu pri vyhľadávaní.

Najjednoduchším príkladom stromu pre spojitú metriku je asi VPT (Vantage-Point Tree) [9]. Stručne si tento strom popíšeme. Predpokladajme, že chceme zaindexovať množinu bodov **U**. VPT je binárny strom vytvorený nasledujúcim spôsobom: nech P je ľubovoľný bod z množiny **U**, zvolíme ho za koreň stromu. Vypočítame priemernú vzdialenosť ostatných bodov od bodu P, označíme ju *md*. Do množiny bodov pre ľavý podstrom zaradíme tie body $X \in \mathbf{U}$, pre ktoré platí: $d(P, X) \leq md$. Do množiny bodov pre pravý podstrom zaradíme zvyšné body z množiny **U**. Na obidve množiny rekurzívne aplikujeme tento postup.

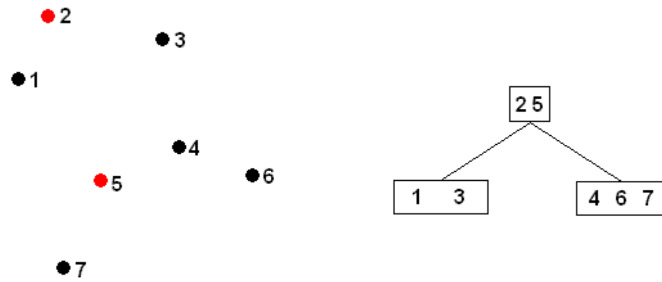


Obr.2.1 Vantage-Point Tree

Obrázok 2.1 znázorňuje príklad jednoduchého VPT. Rozšírením VPT je MVPT (Multi Vantage-Point Tree) [9], ktorý je *m*-árny. Jednoducho nevypočíta len jednu priemernú vzdialenosť, ale *m*-1 vzdialeností a body rozdelí do *m* podstromov podľa toho, do ktorého intervalu patrí ich vzdialenosť od zvoleného bodu P. Je zrejmé, že nie je možné zaručiť, aby bol takýto strom vyvážený a navyše VPT neposkytuje dynamické pridávanie bodov.

Ďalším stromom pre spojitú metriku je BST (BiSector Tree) [9]. Je to opäť binárny strom vytvorený nasledujúcim spôsobom. Chceme zaindexovať množinu bodov **U**, vyberieme z nej dva body P_1, P_2. Do množiny bodov pre ľavý podstrom dáme tie body $X \in \mathbf{U}$, pre ktoré

platí $d(X, P_1) \leq d(X, P_2)$. Do množiny bodov pre pravý podstrom zaradíme zvyšné body z **U**. Tento postup rekurzívne aplikujeme na obidve množiny.



Obr. 2.2 BiSector Tree

Na obr. 2.2 je znázornená prvá úroveň BST. Zovšeobecnením BST je GNAT (Geometric Near-neighbour Access Tree) [9], ktorý je *m*-árny. Ten zakaždým vyberie *m* bodov $P_1, \ldots, P_m$ a množinu bodov rozdelí do *m* podstromov podľa toho, ku ktorému z vybraných bodov sú najbližšie. GNAT však tiež nie je vyvážený.

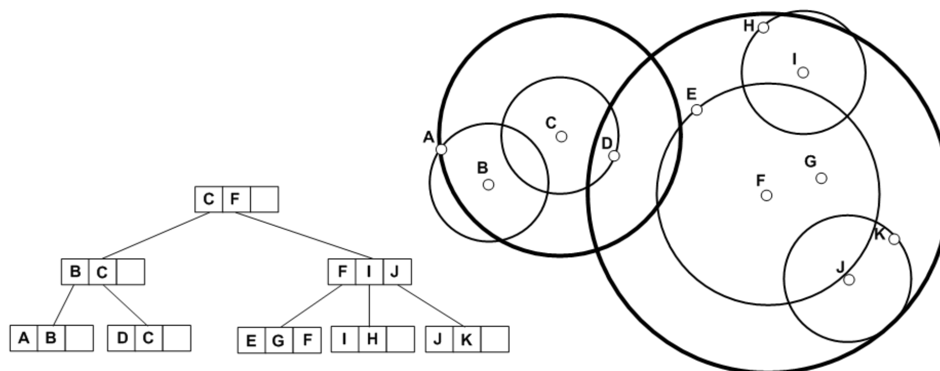
Podobne by sme mohli pokračovať opisom ďalších štruktúr ako SAT, AESA, LAESA, atď. Nakoniec by sme však museli priznať, že pre náš problém je najvhodnejší M-strom (Metric Tree) [7]. Na indexovanie bodov teda použijeme M-strom.

2.2 M-strom

M-strom je indexovacia štruktúra navrhnutá pre uchovávanie objektov, ktoré môžeme reprezentovať bodmi v metrickom priestore. M-strom poskytuje nezávislosť od metrického priestoru, ktorého body chceme v ňom uchovávať. Ak chceme použiť na indexovanie bodov M-strom, stačí ak poznáme funkciu na počítanie vzdialenosti medzi ľubovoľnými dvoma bodmi, ktorá je metrikou. Konkrétnou metrikou, ktorú M-stromu poskytneme, a typom objektov, ktoré budeme v strome uchovávať, definujeme metrický priestor.

M-strom je štruktúrou, ktorá poskytuje stránkovanie jednotlivých uzlov na pevný disk, dynamické pridávanie nových bodov a navyše je vyvážený. Teda okrem nevyhnutných požiadaviek spĺňa aj všetky ostatné sledované vlastnosti. Princíp vyváženosti a dynamického pridávania bodov je založený na princípe vytvárania B-stromu, u ktorého bol tento spôsob prvý krát použitý. Podrobne si tento proces popíšeme v časti 2.3 venovanej vytváraniu M-stromu.

Všetky indexované body sú v M-strome uložené v listoch. Ostatné (vnútorné) uzly stromu slúžia na hierarchické zoskupovanie uzlov nižšej úrovne do samostatných celkov t.j. uzlov vyššej úrovne. Každý uzol M-stromu predstavuje guľu daného metrického priestoru. Listy obsahujú body uchovávané v strome, vnútorné uzly obsahujú informácie o svojich poduzloch. Uzly majú jednotnú kapacitu a jednotné minimálne zaplnenie, ktoré musia dosahovať všetky uzly s výnimkou koreňa stromu. Príklad M-stromu znázorňuje obrázok 2.3.



Obr. 2.3 Príklad M-stromu a grafické znázornenie bodov a uzlov.

Ako bolo spomenuté, M-strom poskytuje stránkovanie uzlov na pevný disk. Presnejšie to znamená, že všetky uzly sú uložené na pevnom disku a tvoria elementárne celky, ktoré z disku čítame (alebo na disk zapísujeme). Jedným prístupom na disk budeme rozumieť prečítanie jedného uzla z disku (alebo zapísanie jedného uzla na disk).

Obsah uzlov M-stromu je navrhnutý, tak aby bolo pri vyhľadávaní možné minimalizovať nie len počet prístupov na disk, ale aj počet výpočtov vzdialeností. Pri netriviálnom výpočte vzdialenosti rastie dôležitosť minimalizácie ich počtu. Formálne môžeme uzly popísať nasledujúcim spôsobom:

$$\mathbf{N} = \langle C(\mathbf{N}), r(\mathbf{N}), p(\boldsymbol{P}(\mathbf{N})), \boldsymbol{D}(\mathbf{N}) \rangle \qquad \text{(uzol M-stromu)}$$

$$\boldsymbol{D}(\mathbf{N}) = \{ \langle \mathrm{O}, d(\mathrm{O}, C(\mathbf{N})) \rangle : \mathrm{O} \text{ je bod listu } \mathbf{N} \} \qquad \text{(ak uzol } \mathbf{N} \text{ je list)}$$

$$\boldsymbol{D}(\mathbf{N}) = \{ \langle C(\mathbf{M}), d(C(\mathbf{M}), C(\mathbf{N})), r(\mathbf{M}), p(\mathbf{M}) \rangle : \boldsymbol{P}(\mathbf{M}) = \mathbf{N} \} \qquad \text{(ak uzol } \mathbf{N} \text{ je vnútorný uzol)}$$

N je uzol, $C(\mathbf{N})$ je stred uzla **N** vybraný pre list spomedzi jeho bodov O a pre vnútorný uzol spomedzi stredov jeho poduzlov $C(\mathbf{M})$. $\boldsymbol{P}(\mathbf{N})$ je rodičovský uzol uzla **N** a $p(\mathbf{N})$ je smerník na uzol **N**. Funkcia d je metrika (vzdialenostná funkcia medzi bodmi) a $r(\mathbf{N})$ je polomer uzla **N**. Pre list **N** je $\boldsymbol{D}(\mathbf{N})$ množina záznamov o bodoch uložených v liste. Každý záznam nesie informáciu o jednom bode. Pre vnútorný uzol **N** je $\boldsymbol{D}(\mathbf{N})$ množina záznamov o poduzloch uzla **N**. Každý záznam nesie informáciu o jednom poduzle.

Z uvedenej formalizácie uzlov M-stromu je zrejmé, že každý list udržiava ku všetkým svojim bodom aj ich vzdialenosť od svojho stredu. Body, ktoré udržiava vnútorný uzol sú stredy jeho poduzlov a prene eviduje tiež ich vzdialenosť od svojho stredu. Vnútorné uzly udržiavajú pre každý svoj poduzol ešte navyše smerník na daný poduzol a jeho polomer. Všetky uzly ešte evidujú svoj vlastný stred, polomer a smerník na svojho rodiča.

Koreň stromu však nemá ani stred, ani polomer ani rodiča. V jeho záznamoch navyše chceme evidovať vzdialenosti jeho bodov (resp. stredov jeho poduzlov) od jeho stredu. Kvôli neskoršej potrebe v algoritmoch dodefinujme hodnoty koreňa stromu nasledovne:

$$\mathbf{R} = \langle null, null, null, \boldsymbol{D}(\mathbf{R}) \rangle \qquad \text{(koreň M-stromu)}$$

$$\boldsymbol{D}(\mathbf{R}) = \{ \langle \mathrm{O}, 0 \rangle : \mathrm{O} \text{ je bod listu } \mathbf{R} \} \qquad \text{(ak koreň } \mathbf{R} \text{ je list)}$$

$$\boldsymbol{D}(\mathbf{R}) = \{ \langle C(\mathbf{M}), 0, r(\mathbf{M}), p(\mathbf{M}) \rangle : \boldsymbol{P}(\mathbf{M}) = \mathbf{R} \} \qquad \text{(ak koreň } \mathbf{R} \text{ je vnútorný uzol)}$$

Stred koreňa stromu, jeho polomer a smerník na jeho rodiča naozaj nepotrebujeme, preto majú prázdnu hodnotu *null*. V záznamoch o bodoch (resp. poduzloch) koreňa stromu je predpočítaná vzdialenosť od jeho stredu definovaná na nulu.

Smerníkom je najčastejšie nejaký offset, ukazujúci na začiatok dát daného uzla v súbore. Polomer uzla **N** vyjadruje, že žiaden bod podstromu uzla **N** nemá vzdialenosť od stredu uzla **N** väčšiu ako je ten polomer uzla **N**. Iba vtedy môžeme polomer uzla **N** považovať za korektný. Guľa uzla teda nemusí nutne obsahovať kompletne celé gule svojich poduzlov. Túto situáciu znázorňuje opäť obrázok 2.3, kde až v dvoch prípadoch potomok vytŕča zo svojho rodiča. Tu je ale nutné poznamenať, že to vytŕčanie je korektné, iba ak časť uzla, ktorá vytŕča zo svojho rodičovského uzla je prázdna, t.j. neobsahuje žiadny bod.

Pre kvalitu inexácie v M-strome je lepšie, aby boli polomery uzlov čo najmenšie a tiež čo najmenšie prieniky uzlov na danej úrovni. Toto sa snažíme zabezpečiť už pri pridávaní bodov do stromu, t.j. už pri vytváraní stromu.

2.3 Vytváranie M-stromu

Vytváranie M-stromu spočíva v spôsobe pridávania objektov do stromu. Ten je prevzatý z B-stromov a prispôsobený pre body metrického priestoru. Konkrétnymi algoritmami si popíšeme pridanie jedného bodu tak, aby bol strom vyvážený, aby boli body rozdelené do uzlov čo najvhodnejšie a aby uzly obsahovali všetky informácie uvedené v predchádzajúcej časti.

Keďže všetky body v M-strome sa nachádzajú v listoch, aj nový bod musíme pridať do nejakého listu. Prvou fázou pridania bodu do stromu je teda nájdenie najvhodnejšieho listu, do ktorého potom bod pridáme. Na tento účel slúži rekurzívna metóda *findLeaf* idúca cez výšku stromu zhora nadol. Metóda *findLeaf* má dva vstupy: nový bod O pridávaný do stromu a uzol **N**, na ktorý aplikujeme hľadanie. Výstupom je list **L**. Metóda vyberie čo najvhodnejšieho potomka uzla **N**, pre ktorého rekurzívne zavolá vyhľadávanie, až kým nenarazí na list, ten sa stáva vysledkom hľadania. Hľadanie samozrejme štartujeme od koreňa stromu volaním ***findLeaf***(O, **R**), kde **R** je koreň stromu.

L ***findLeaf***(O, **N**) {
1. Ak **N** je list {
    1. Vráť list **N** a skonči.
    }
2. Ak **N** je vnútorný uzol {
    1. Nech $\mathbf{B} = \{ \mathbf{M} : \boldsymbol{P}(\mathbf{M}) = \mathbf{N} \;\&\; d(O, C(\mathbf{M})) \leq r(\mathbf{M}) \}$.
    2. Ak $\mathbf{B} \neq \emptyset$ {
        1. Nech **V** je taký uzol z množiny **B**, že platí:
           $d(O, C(\mathbf{V})) = min\{ d(O, C(\mathbf{M})) : \mathbf{M} \in \mathbf{B} \}$.
        2. Vráť výsledok volania ***findLeaf***(O, **V**) a skonči.
        }
    3. Ak $\mathbf{B} = \emptyset$ {
        1. Nech **V** je taký poduzol uzla **N**, že platí:
           $d(O, C(\mathbf{V})) - r(\mathbf{V}) = min\{ d(O, C(\mathbf{M})) - r(\mathbf{M}) : \boldsymbol{P}(\mathbf{M}) = \mathbf{N} \}$.
        2. Zväčši polomer uzla **V** na hodnotu $d(O, C(\mathbf{V}))$.
        3. Vráť výsledok volania ***findLeaf***(O, **V**) a skonči.
        }
    }
}

Všimnime si, že metóda *findLeaf* sa v prvom rade snaží vybrať taký poduzol uzla **N**, aby sa nemusel zväčšovať polomer toho poduzla. Zo všetkých takých poduzlov potom

vyberie ten, pre ktorý je pridávaný bod najbližšie k jeho stredu. Ak také poduzly neexistujú, tak vyberie taký poduzol, pre ktorý je pridávaný bod najbližšie k jeho guli, ktorú predstavuje. Vtedy však treba zväčšiť polomer vybraného poduzla, aby po pridaní bodu ostal jeho polomer korektný.

Jedno vyhľadanie listu metódou *findLeaf* spôsobí prechod stromom zhora nadol pozdĺž jednej vetvy. Na každej úrovni stromu sa zlezie do práve jedného poduzla. To znamená, že sa vykoná toľko prístupov na disk, aká je výška stromu.

Do nájdeného listu potom jednoducho ten nový bod pridáme. Uzly však majú obmedzenú kapacitu. Ak nájdený list ešte nebol plný, tak je všetko v priadku a pridanie je úspešne ukončené. Inak musí dôjsť k rozdeleniu listu na dva nové listy. Formálne popíšeme pridanie bodu do listu metódou *addToLeaf*, ktorá má na vstupe pridávaný bod O a nájdený list **L** a nemá žiaden výstup.

void ***addToLeaf***(O, **L**) {
1. Pridaj do množiny ***D***(**L**) záznam <O, *d*(O, *C*(**L**))>.
2. Ak je |***D***(**L**)| väčšia ako povolená kapacita {
    1. Vytvor nový list **L’**.
    2. Rozdeľ záznamy množiny ***D***(**L**) čo najlepšie medzi množiny ***D***(**L**) a ***D***(**L’**), nájdi nové stredy *C*(**L**) a *C*(**L’**), nájdi nové polomery *r*(**L**) a *r*(**L’**), prepočítaj v záznamoch vzdialenosti od nových stredov.
    3. Ak uzol **L** nie je koreň stromu {
        1. Zavolaj ***addToInnerNode***(**L’**, ***P***(**L**)).
        }
    4. Ak uzol **L** je koreň stromu {
        1. Vytvor nový vnútorný uzol **R**.
        2. Zavolaj ***addToInnerNode***(**L**, **R**).
        3. Zavolaj ***addToInnerNode***(**L’**, **R**).
        4. Zmeň koreň stromu na uzol **R**.
        }
    }
}

Kľúčovým problémom pridávania bodu do listu je rozdelenie listu na dva listy v prípade, že došlo k jeho preplneniu. Jeden list vytvoríme nový a ten existujúci preplnený len zmeníme tak, že časť jeho bodov vložíme do nového listu. Pri tomto rozdeľovaní treba pre obidva listy nájsť nové, čo najvhodnejšie stredy. Podotýkam, že stredy listov sa vyberajú len spomedzi bodov, ktoré sú v nich uložené. Následne zvyšné body rozdelíme medzi tie dva listy podľa toho, ku ktorému zo stredov sú bližšie. Uzly však majú predpísanú nie len kapacitu, ale aj minimálne zaplnenie. Ak ho pri takto vykonanom rozdelení jeden z listov nedosahuje, musíme do neho preložiť minimálny nutný počet najbližších bodov z toho druhého listu. Polomer listu je potom vzdialenosť jeho najvzdialenejšieho bodu od jeho stredu. Snažíme sa pre listy nájsť také stredy, aby bol väčší z ich polomerov čo najmenší možný. Aby sme to dosiahli, musíme vyskúšať zvoliť za stredy každú dvojicu bodov z pôvodneho preplneného listu. Ak je kapacita listu m, preplnený list potom obsahuje $m + 1$ bodov, a tak treba vyskúšať $m(m + 1)/2$ možností.

Pre stručnosť neuvádzam presné detaily o tom, čo všetko je potrebné v dotknutých uzloch zmeniť. Sú to však jasne viditeľné zmeny potrebné pre udržanie konzistencie dát v uzloch (smerníky, polomery, stredy, vzdialenosti od stredov).

Vytvorili sme teda list **L’** ako nového suseda listu **L**. Ak list **L** nebol koreňom stromu, tak nový list **L’** musíme pridať rodičovi listu **L**. Ak bol list **L** koreňom stromu, tak musíme

vytvoriť nový vnútorný uzol **R**, do ktorého pridáme aj list **L** aj list **L'**. Uzol **R** sa stane novým koreňom stromu a výška stromu stúpne o jednu úroveň. Na operáciu pridávania nového uzla do jeho nastávajúceho rodičovského uzla slúži metóda *addToInnerNode*. Jej prvým parametrom je pridávaný uzol **M** a druhým parametrom je uzol **N**, do ktorého sa pridáva.

void ***addToInnerNode***(**M**, **N**) {

1. Pridaj do množiny $\boldsymbol{D}(\mathbf{N})$ záznam $<C(\mathbf{M}), d(C(\mathbf{M}), C(\mathbf{N})), r(\mathbf{M}), p(\mathbf{M})>$.
2. Ak je $|\boldsymbol{D}(\mathbf{N})|$ väčšia ako povolená kapacita {
    1. Vytvor nový vnútorný uzol **N'**.
    2. Rozdeľ záznamy množiny $\boldsymbol{D}(\mathbf{N})$ čo najlepšie medzi množiny $\boldsymbol{D}(\mathbf{N})$ a $\boldsymbol{D}(\mathbf{N'})$, nájdi nové stredy $C(\mathbf{N})$ a $C(\mathbf{N'})$, nájdi nové polomery $r(\mathbf{N})$ a $r(\mathbf{N'})$, prepočítaj v záznamoch vzdialenosti od nových stredov a zmeň uzlom preloženým do uzla **N'** smerník na ich nového rodiča.
    3. Ak uzol **N** nie je koreň stromu {
        1. Zavolaj ***addToInnerNode***(**N'**, ***P***(**N**)).

        }
    4. Ak uzol **N** je koreň stromu {
        1. Vytvor nový vnútorný uzol **R**.
        2. Zavolaj ***addToInnerNode***(**N**, **R**).
        3. Zavolaj ***addToInnerNode***(**N'**, **R**).
        4. Zmeň koreň stromu na uzol **R**.

        }

    }

}

Kľúčovou operáciou tohoto pridávania je opäť delenie uzla. Tentokrát ide o delenie vnútorného uzla. Stredy vnútorných uzlov sa vyberajú spomedzi stredov ich poduzlov. Aj tu je potrebné vyskúšať všetky dvojice potenciálnych stredov, aby sme pre uzly **N** a **N'** našli také stredy, že po rozdelení jednotlivých poduzlov medzi tie dva uzly **N** a **N'** dosiahneme väčší z ich polomerov čo najmenší možný.

Ideálny polomer vnútorného uzla je vzdialenosť najvzdialenejšieho bodu jeho podstromu od jeho stredu. Je ale neúnosné, aby sa pri zisťovaní polomeru zakaždým prehľadával celý podstrom daného uzla. Preto polomer vnútorného uzla **N** vypočítame ako $max\{ d(C(\mathbf{M}), C(\mathbf{N})) + r(\mathbf{M}) : \boldsymbol{P}(\mathbf{M}) = \mathbf{N} \}$.

Po rozdelení uzla musíme opäť rodičovi pridať nový poduzol metódou *addToInnerNode*. Ak bol aktuálne rozdeľovaný uzol **N** koreňom stromu, vytvoríme nový koreň stromu **R** s dvoma poduzlami **N** a **N'**, čím stúpne výška stromu o jednu úroveň.

3 Dopyty nad M-stromom

Táto kapitola je venovaná dopytom na M-stromom. Najprv rozoberieme existujúce dopyty a vysvetlíme si, prečo nám nestačia. Následne predstavíme nový algoritmus Sort query a jeho fuzzy verziu pre nový druh dopytu.

3.1 Existujúce dopyty vs. požadovaný dopyt

Nad M-stromom sú všeobecne známe dva typy dopytov. Sú to rozsahový dopyt (range query) a dopyt na *k* najbližších susedov (*k*-nn query). Príkladom rozsahového dopytu je otázka: „Ktoré hotely sú od centra Košíc vzdialené najviac 20 km ?“. Odpoveďou je potom množina hotelov v okruhu najviac 20 km od centra Košíc. Príkladom dopytu na *k* najbližších susedov môže byť požiadavka: „Nájdi mi 15 najbližších hotelov pri Štrbskom plese!“. Odpoveďou je zoznam obsahujúci 15 hotelov, ktoré sú spomedzi všetkých hotelov najbližšie ku Štrbskému plesu. Tento zoznam je navyše zotriedený od najbližšieho po najvzdialenejší.

Všimnime si, že rozsahový dopyt potrebuje dopredu vedieť, aký veľký okruh čo sa týka vzdialenosti od vybraného miesta nás zaujíma. Počet hotelov, ktoré v danom okruhu nájde je však preňho vopred neznámy. Dopyt na *k* najbližších susedov zase potrebuje dopredu vedieť, koľko hotelov chceme nájsť. Avšak preňho je zase vopred neznáma veľkosť okruhu, v akom stačí hotely hľadať.

Dopyt, ktorý potrebujeme pre implementáciu sekvenčného prístupu nad M-stromom vykonať, by sme mohli na príklade uviesť asi takto: „Hovor mi postupne hotely v poradí od najbližšieho po najvzdialenejší od centra Bratislavy, kým nepoviem stop!“. Rozsahový dopyt je pre tento problém jasne absolútne nepoužiteľný. Nevieme dopredu určiť, v akom veľkom okruhu od centra Bratislavy stačí hľadať. Dopyt na *k* najbližších susedov je nášmu problému podobnejší. Ba dokonca príslušných *k* bodov vráti v poradí od najbližšieho po navzdialenejší. Vyžaduje však dopredu vedieť číslo *k*, teda počet udávajúci, koľko hotelov chceme nájsť. Dopytu pre sekvenčný prístup nedokážeme vopred povedať, koľko najbližších hotelov budeme potrebovať. Dopyt na *k* najbližších susedov je teda tiež nepoužiteľný.

Pre sekvenčný prístup je neznámym aj vzdialenostný rozsah aj počet požadovaných hotelov. Preto musíme byť potenciálne schpní vrátiť všetky body stromu v správnom poradí. Riešením by bolo zadať na vstupe dopytu na *k* najbližších susedov vždy číslo *k* rovné počtu bodov v strome, ale to je ekvivalentné zotriedeniu všetkých bodov v čase otázky.

Skôr ako uvediem dopyt, pomocou ktorého dokážeme implementovať sekvenčný prístup, popíšeme si presnejšie známe dopyty, t.j. rozsahový dopyt a dopyt na *k* najbližších susedov, uvedené [6]. Keďže pre algoritmy jednotlivých dopytov budeme prísne sledovať počet prístupov na disk, tak v ich popise vždy explicitne uvediem, kde presne treba prečítať uzol z disku a vykonať tak jeden prístup na disk.

3.2 Rozsahový dopyt – algoritmus range query

Rozsahový dopyt má parametre bod a polomer, čo určuje guľu v danom metrickom priestore. Výstupom je množina všetkých bodov stromu, ktoré patria do gule definovanej parametrami.

Hľadanie sa štartuje od koreňa stromu a postupne zostupuje k listom. Na každej úrovni sa prehľadajú len tie uzly, ktorých guľa má neprázdny prienik s guľou definovanou parametrami dopytu. Táto podmienka zaručuje, že pri vyhľadávaní sa vykoná len minimálny nutný počet prístupov na disk. V každom navštívenom liste sa pre každý jeho bod overí, či patrí do gule daného rozsahového dopytu, teda, či ho treba zahrnúť do výsledku.

Formálne si rozsahový dopyt popíšeme rekurzívnou metódou *rangeQuery*, ktorá má štyri parametre na vstupe. Prvé dva parametre určujú guľu dopytu, teda sú to bod Q a polomer *q*. Tretím parametrom je uzol **N**, na ktorý sa aplikuje hľadanie a štvrtým parametrom je číslo $v = d(\mathrm{Q}, C(\mathbf{N}))$. Číslo *v* je teda vzdialenosť bodu Q od stredu uzla **N** a tá je vždy pred volaním metódy *rangeQuery* na uzol **N** už vypočítaná. Je to preto, že volanie metódy *rangeQuery* na uzol **N** je podmienené neprázdnym prienikom gule uzla **N** s guľou dopytu. Prienik gule uzla **N** s guľou rozsahového dopytu je neprázdny, ak $d(\mathrm{Q}, C(\mathbf{N})) \leq q + r(\mathbf{N})$. Teda zistenie, že ich

prienik je neprázdny, vyžaduje výpočet tejto vzdialenosti. Výstupom metódy je výsledný zoznam nájdených bodov Rq (result queue).

Číslo v na vstupe a uložené vzdialenosti bodov od stredu uzla, v ktorom sa nachádzajú, sú využité na minimalizáciu počtu výpočtov vzdialenosti.

Rq ***rangeQuery***(Q, q, **N**, $v = d(Q, C(\mathbf{N}))$) {
1. Nech zoznam Rq je prázdny.
2. Ak **N** je list {
    1. Nech $\mathbf{B} = \{ O : |d(O, C(\mathbf{N})) - v| \leq q$ & O je bod listu **N** }.
    2. Pre všetky body $O \in \mathbf{B}$ vypočítaj $d(O, Q)$.
    3. Nech $\mathbf{B^*} = \{ O : O \in \mathbf{B} \ \& \ d(O, Q) \leq q \}$.
    4. Vlož do zoznamu Rq všetky body $O \in \mathbf{B^*}$.

    }
3. Ak **N** je vnútorný uzol {
    1. Nech $\mathbf{B} = \{ \mathbf{M} : |d(C(\mathbf{M}), C(\mathbf{N})) - v| \leq q + r(\mathbf{M}) \ \& \ \boldsymbol{P}(\mathbf{M}) = \mathbf{N} \}$.
    2. Pre všetky uzly $\mathbf{M} \in \mathbf{B}$ vypočítaj $d(Q, C(\mathbf{M}))$.
    3. Nech $\mathbf{B^*} = \{ \mathbf{M} : \mathbf{M} \in \mathbf{B} \ \& \ d(Q, C(\mathbf{M})) \leq q + r(\mathbf{M}) \}$.
    4. Pre každý uzol $\mathbf{M} \in \mathbf{B^*}$ vykonaj {
        1. Prečítaj uzol **M** z disku.
        2. Nech Hq = ***rangeQuery***(Q, q, **M**, $d(Q, C(\mathbf{M}))$).
        3. Pridaj do zoznamu Rq všetky body zoznamu Hq.

        }

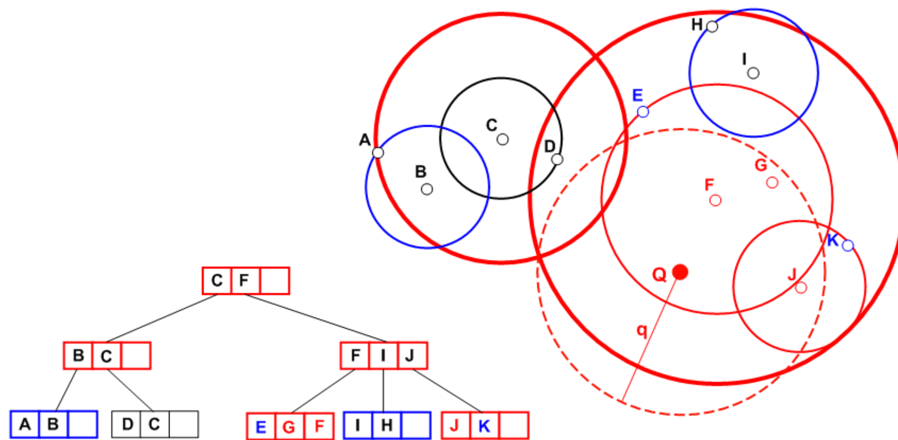
    }
4. Vráť zoznam Rq a skonči.

}

Hľadanie štartujeme od koreňa stromu. Koreň stromu nemá stred a nevieme preňho vypočítať vstupnú hodnotu v. Preto hodnotu v na začiatku nastavíme na nulu. Ak **R** je koreň stromu, tak rozsahový dopyt spustíme volaním ***rangeQuery***(Q, q, **R**, 0).

Všimnime si ešte, kde sa použije číslo v a uložené vzdialenosti bodov od stredu ich uzla na redukciu počtu výpočtov vzdialenosti. Uvažujme prípad, že **N** je vnútorný uzol. Naším cieľom je vybrať tie poduzly uzla **N**, ktoré majú neprázdny prienik s guľou dopytu. V kroku 2.1 (resp. 3.1) najprv definujeme **B** ako množinu vybraných poduzlov uzla **N** a ďalej pracujeme už len s poduzlami v množine **B**. Do množiny **B** sa nedostanú tie poduzly uzla **N**, pre ktoré platí $|d(C(\mathbf{M}), C(\mathbf{N})) - d(Q, C(\mathbf{N}))| > q + r(\mathbf{M})$. Na základe tejto nerovnosti totiž môžeme rovno vyhlásiť, že tieto poduzly majú prázdny prienik s guľou dopytu. Rozhodnutie o tom, či poduzol **M** patrí do množiny **B** si nevyžaduje žiaden výpočet vzdialenosti. Hodnota v je vstupným parametrom metódy a hodnota $d(C(\mathbf{M}), C(\mathbf{N}))$ je uložená v uzle **N**, konkrétne v zázname o jeho poduzle **M**. Takto dokážeme zamietnuť niektoré poduzly uzla **N** bez toho, aby sme počítali ich vzdialenosť od bodu Q. Analogicky to funguje aj v prípade, keď **N** je list. Rovnako aj vtedy dokážeme niektoré body listu **N** zamietnuť bez toho, aby sme počítali ich vzdialenosť od bodu Q.

Pri dopyte však sledujeme hlavne počet prístupov na disk. Rozsahový dopyt vykoná toľko prístpov na disk, koľko krát sa uskutoční volanie metódy *rangeQuery*. Inými slovami, len pred volaním metódy *rangeQuery* na uzol **N** (a nikde inde) musíme uzol **N** prečítať z disku. Z disku sa teda prečítajú len tie uzly, ktoré majú neprázdny prienik s guľou dopytu a preto je algoritmus range query z hľadiska počtu prístupov na disk optimálny.



Obr. 3.1 Rozsahový dopyt so stredom Q a polomerom *q*.

Príklad rozsahového dopytu znázorňuje obrázok 3.1. Červenou farbou sú zvýraznené tie uzly a body, ktoré sú relevantné voči dopytu. Červené body sú teda výsledkom dopytu a červené uzly su tie, ktoré sme museli prečítať z disku. Všimnime si, že sú to práve tie uzly, ktoré majú neprázdny prienik s guľou dopytu. Ostatné uzly a body nie sú voči dopytu relevantné, preto boli zamietnuté. Modrou fabrou sú zvýraznené tie uzly (resp. body), ktoré sa dostali v kroku 2.1 (resp. 3.1) do množiny **B**, ale v kroku 2.3 (resp. 3.3) sa už nedostali do množiny **B***. V kroku 2.2 (resp. 3.2) bola pre tieto body (resp. uzly) vypočítaná ich vzdialenosť od bodu Q. Až na základe tejto vzdialenosti bolo možné rozhodnúť, že tieto body (resp. uzly) nie sú relevantné voči dopytu, a tak boli zamietnuté. Napokon nezvýraznené ostali tie body (resp. uzly), ktoré sa nedostali ani do množiny **B**. Tieto body (resp. uzly) boli zamietnuté bez toho, aby sa počítala ich vzdialenosť od bodu Q. Na ich zamietnutie stačili v ich záznamoch uložené vzdialenosti od stredu rodičovského uzla a číslo *v* na vstupe metódy *rangeQuery*.

3.3 Dopyt na *k* najbližších susedov – algoritmus kNN query

Parametrami tohoto dopytu sú bod Q (nazývaný aj kotvový bod) a číslo *k* udávajúce koľko najbližších bodov od bodu Q chceme nájsť. Výstupom je zoznam *k* najbližších bodov zoradený od najbližšieho po najvzdialenejší od bodu Q. Súčasťou výstupu sú aj ich vzdialenosti od bodu Q.

Algoritmus si popíšeme metódou *knnQuery*, ktorá má dva parametre. Prvým je kotvový bod Q a druhým číslo *k*. Výstupom metódy je výsledný zoznam nájdených bodov a ich vzdialeností od Q označený Rq. Metóda používa dva zoznamy: spomínaný výsledný zoznam Rq a pracovný zoznam Wq (work queue). Výsledný zoznam má pevne stanovený počet prvkov na *k* a udržiava prvky dvojakého typu:

bodový prvok: <O, *d*(O, Q)>
uzlový prvok: <*p*(**N**), *d*(*C*(**N**), Q) + *r*(**N**)>

O je bod v strome, **N** je uzol stromu a Q je kotvový bod dopytu. Bodový prvok je usporiadaná dvojica skladajúca sa z bodu a jeho vzdialenosti od Q. Uzlový prvok je usporiadaná dvojica, ktorej prvá zložka je smerník na uzol **N** a druhá zložka je maximálna vzdialenosť od Q, ktorú môže nejaký bod podstromu uzla **N** dosiahnuť.

Pracovný zoznam Wq obsahuje prvky jedného typu:

uzlový prvok: $<p(\mathbf{N}), max\{d(C(\mathbf{N}), Q) - r(\mathbf{N}), 0\}>$

N je uzol stromu a Q je kotvový bod. Prvkom pracovného zoznamu je usporiadaná dvojica: smerník na uzol **N** a minimálna vzdialenosť od Q, ktorú môže nejaký bod podstromu uzla **N** dosiahnuť.

Obidva zoznamy budú v priebehu výpočtu udržiavané zotriedené vzostupne podľa druhej zložky svojich prvkov.

```
Rq knnQuery(Q, k) {
1.   Inicializuj zoznam Rq k rovnakými prvkami <null, ∞>.
2.   Nech R je koreň stromu.
3.   Vlož do prázdneho zoznamu Wq prvok <p(R), ∞>.
4.   Pokiaľ zoznam Wq nie je prázdny opakuj {
     1.   Nech <x, y> je prvý prvok zoznamu Wq, odober z neho tento prvok.
     2.   Ak zoznam Rq obsahuje prvok <x, z>, kde z je ľubovoľné číslo {
          1.   Odober prvok <x, z> zo zoznamu Rq.
          2.   Pridaj do zoznamu Rq prvok <null, ∞> na poslednú (k-tu) pozíciu.
          }
     3.   Nech p(N) = x, prečítaj uzol N z disku.
     4.   Ak N je list {
          1.   Pre každý bod O listu N vykonaj {
               1.   Nech <v, w> je práve k-ty prvok zoznamu Rq.
               2.   Ak |d(O, C(N)) – d(Q, C(N))| ≤ w {
                    1.   Vypočítaj d(O, Q).
                    2.   Ak d(O, Q) ≤ w {
                         1.   Odober prvok <v, w> zo zoznamu Rq.
                         2.   Vlož do zoznamu Rq prvok <O, d(O, Q)> tak,
                              aby bol zoznam Rq zotriedený.
                         3.   Nech <v, w> je práve k-ty prvok zoznamu Rq.
                         4.   Odober zo zoznamu Wq všetky prvky <a, b>, pre
                              ktoré platí: b > w.
                         }
                    }
               }
          }
     5.   Ak N je vnútorný uzol {
          1.   Pre každý poduzol M uzla N vykonaj {
               1.   Nech <v, w> je práve k-ty prvok zoznamu Rq.
               2.   Ak |d(C(M), C(N)) – d(Q, C(N))| ≤ w + r(M) {
                    1.   Vypočítaj d(C(M), Q).
                    2.   Ak max{d(C(M), Q) – r(M), 0} ≤ w {
                         1.   Vlož do zoznamu Wq prvok
                              <p(M), max{d(C(M), Q) – r(M), 0}> tak, aby bol
                              zoznam Wq zotriedený.
                         2.   Ak d(C(M), Q) + r(M) ≤ w {
                              1.   Odober prvok <v, w> zo zoznamu Rq.
                              2.   Vlož do zoznamu Rq prvok
                                   < p(M), d(C(M), Q) + r(M)> tak, aby bol
                                   zoznam Rq zotriedený.
```

3. Nech <*v*, *w*> je práve *k*-ty prvok zoznamu Rq.
4. Odober zo zoznamu Wq všetky prvky <*a*, *b*>, pre ktoré platí: *b* > *w*.

}

}

}

}

}

}

5. Vráť zoznam Rq a skonči.

}

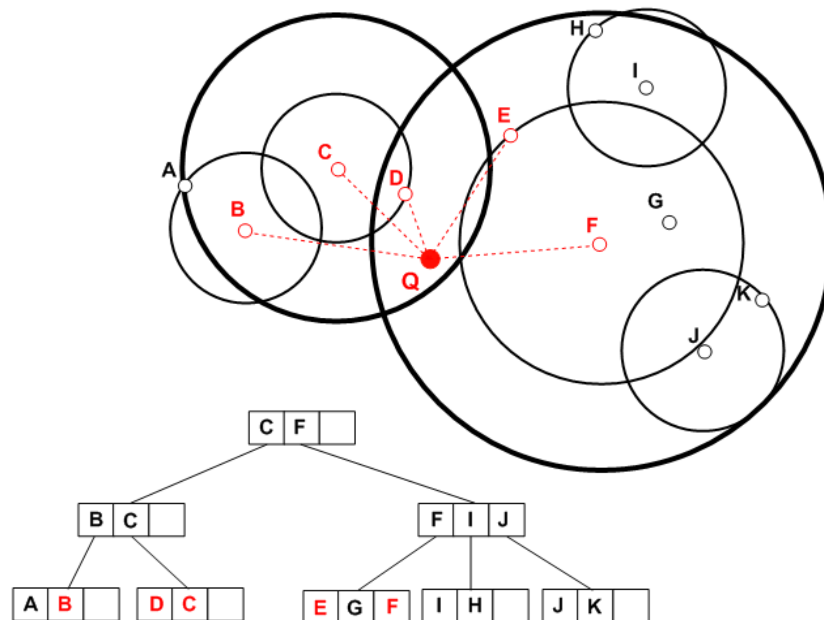
Výsledný zoznam predstavuje počas výpočtu aktuálnych *k* doteraz najbližších nájdených bodov a uzlov dokopy. Vzdialenosť uložená v *k*-tom prvku výsledného zoznamu určuje aktívny polomer dopytu. Na začiatku výpočtu je tento aktívny polomer neohraničený. Behom výpočtu sa do výsledného zoznamu dostávajú stále menej a menej vzdialené body a uzly, čím vytláčajú zo zoznamu tie vzdialenejšie a aktívny polomer sa zmenšuje. Pracovný zoznam obsahuje tie uzly stromu, ktoré ešte neboli prečítané z disku a sú relevantné vzhľadom na aktívny polomer dopytu. To znamená, že ešte stále sa môžu v podstromoch týchto uzlov nachádzať body, ktoré patria medzi *k* najbližších ku kotvovému bodu Q. Preto na začiatku výpočtu obsahuje pracovný zoznam len koreň stromu. Uzly z pracovného zoznamu sú potom v poradí od najbližšieho po najvzdialenejší postupne odoberané a rozbíjajú sa na svoje poduzly (listy sa rozbíjajú na svoje body). Tieto poduzly sú potom zaraďované do jednotlivých zoznamov na správne pozície podľa poradia. Týmto dochádza ku zmenšovaniu aktívneho polomeru. Keďže sa aktívny polomer počas výpočtu zmenšuje, z pracovného zoznamu sú priebežne vyhadzované uzly, ktoré prestávajú byť relevantné. Výpočet sa ukončí, keď dôjde k vyprázdneniu pracovného zoznamu. Vtedy už výsledný zoznam obsahuje len samé body. Je to preto, lebo výsledný zoznam nikdy neobsahuje uzly, ktoré nie sú v pracovnom zozname. Výsledný zoznam je výsledkom hľadania.

Aj tento algoritmus je navrhnutý optimálne vzhľadom na počet prístupov na disk. Prístup na disk sa vykonáva len v kroku 4.3, kedy potrebujeme zobrať prvý uzol pracovného zoznamu a rozbiť ho na jeho poduzly (resp. body, ak je to list). Vždy sa teda prečíta najbližší uzol ku bodu Q čakajúci v pracovnom zozname, pričom pracovný zoznam vždy obsahuje len relevantné uzly. Algoritmus kNN query takto vykoná rovnaký počet prístupov na disk ako algoritmus range query s tým istým kotvovovým bodom Q a polomerom rovným vzdialenosti *k*-teho najbližšieho bodu od Q.

Podobne ako u rozsahového dopytu, aj tu sú uložené vzdialenosti bodov od stredu ich uzla použité na minimalizáciu počtu výpočtov vzdialenosti. Rovnaká nerovnosť, akou boli poduzly (resp. body) zaraďované do množiny **B** u rozsahového dopytu, sa tu nachádza v krokoch 4.5.1.2 (resp. 4.4.1.2). Jedinou zmenou je, že nemáme pevný polomer dopytu *q*, ale postupne sa zmenšujúci aktívny polomer *w*. Treba si ešte uvedomiť, že vzdialenosť *d*(Q, *C*(**N**)), ktorú u rozsahového dopytu predstavuje číslo *v* na vstupe, máme aj v tomto prípade už vypočítanú. Všimnime si, že prv než vložíme nejaký uzol do pracovného zoznamu v kroku 4.5.1.2.2.1, vypočítame v kroku 4.5.1.2.1 vzdialenosť jeho stredu od bodu Q. Teda aj tu sa testovanie tejto nerovnosti zaobíde bez výpočtu vzdialenosti. Analogicky to funguje v prípade, keď uzol **N** je list.

V závere popisu tohto algoritmu chcem poukázať na jednu chybu, ktorá sa v súvislosti s týmto algoritmom objavila v článkoch a prácach, z ktorých som čerpal [6]. Krok č. 4.2. (následne aj 4.2.1 a 4.2.2) nebol uvedený, jednoducho chýbal. V dôsledku toho dochádzalo

k tomu, že vo výslednom zozname sa nachádzali uzly, ktoré sa nenachádzali v pracovnom zozname. Po vyprázdnení pracovného zoznamu sa potom môže stať, že vo výslednom zozname nebudú len samé body, ale aj nejaké uzly. Výsledný zoznam, ktorý sa potom stáva výstupom, nezodpovedá požadovanej odpovedi dopytu na *k* najbližších susedov.



Obr. 3.2 Príklad dopytu na 5 najbližších susedov bodu Q. Výsledok tvoria body D, C, E, F, B v tomto poradí.

3.4 Sekvenčný dopyt na najbližších susedov – algoritmus Sort query

Sekvenčný prístup nad M-stromom zrealizujeme pomocou sekvenčného dopytu na najbližších susedov. Jeho jediným parametrom je kotvový bod, na základe ktorého dokáže vrátiť postupne body M-stromu v poradí od najbližšieho po najvzdialenejší od kotvového bodu.

Sekvenčný dopyt na najbližších susedov si popíšeme dvoma metódami. Prvou je metóda *sortQueryInitialization*, ktorá je inicializačná, nevytvára výstup a jej jediným vstupom je kotvový bod Q. Druhá metóda je *sortQueryNext*, je iteračná, nepotrebuje už žiaden vstup a jej výstupom je jeden bod z M-stromu a jeho vzdialenosť od bodu Q t.j. <O, *d*(O, Q)>. Výpočet odštartujeme volaním ***sortQueryInitialization***(Q) a následnými volaniami ***sortQueryNext***() získavame body M-stromu v poradí od najbližšieho po najvzdialenejší od bodu Q. Metódy používajú jeden pracovný zoznam Wq, ktorý obsahuje prvky dvojakého typu:

bodový prvok: <O, *d*(O, Q), *d*(O, Q)>
uzlový prvok: <*p*(**N**), *max*{*d*(*C*(**N**), Q) – *r*(**N**), 0}, *d*(*C*(**N**), Q) + *r*(**N**)>

O je bod stromu, **N** je uzol a Q je kotvový bod dopytu. Bodový prvok je usporiadaná trojica obsahujúca bod stromu a jeho minimálnu a maximálnu vzdialenosť od bodu Q. Pre bod sú tieto vzdialenosti rovnaké. Uzlovým prvkom je usporiadaná trojica obsahujúca smerník na uzol **N** a jeho minimálnu a maximálnu vzdialenosť od bodu Q. Minimálna vzdialenosť uzla **N**

od bodu Q je najmenšia vzdialenosť, ktorú môže nejaký bod podstromu uzla **N** od bodu Q dosiahnuť. Maximálna vzdialenosť je zase najväčšia vzdialenosť, ktorú môže nejaký bod podstromu uzla **N** od bodu Q dosiahnuť.

Tento zoznam bude počas výpočtu udržiavaný vždy zotriedený. Prvky zoznamu budú teda usporiadané od najbližšieho po najvzdialenejší podľa minimálnej vzdialenosti od bodu Q. Prvky s rovnakou minimálnou vzdialenosťou budú ešte medzi sebou usporiadané od najbližšieho po najvzdialenejší podľa maximálnej vzdialenosti od bodu Q. Pre špeciálny prípad nulového polomeru uzla dodávam, že usporiadanie má byť navyše také, aby bodové prvky boli umiestnené v zozname vždy pred uzlovými prvkami s rovnakou minimálnou vzdialenosťou.

void ***sortQueryInitialization*** (Q) {

1. Nech uzol **R** je koreň stromu.
2. Vlož do prázdneho zoznamu Wq prvok <*p*(**R**), 0, 0>
3. Zapamätaj si bod Q.

}

<O, *d*(O, Q)> ***sortQueryNext***() {

1. Nech <*x*, *y*, *z*> je prvý prvok zoznamu Wq, odober z neho tento prvok.
2. Ak *x* je bod {
   1. Vráť <*x*, *y*> a skonči.

   }
3. Ak *x* je smerník na uzol {
   1. Nech *p*(**N**) = *x*, prečítaj uzol **N** z disku.
   2. Ak **N** je list {
      1. Pre každý bod O listu **N** pridaj do zoznamu Wq prvok <O, *d*(O, Q), *d*(O, Q)> tak, aby bol zoznam Wq zotriedený.

      }
   3. Ak **N** je vnútorný uzol {
      1. Pre každý poduzol **M** uzla **N** pridaj do zoznamu Wq prvok <*p*(**M**), *max*{*d*(*C*(**M**), Q) – *r*(**M**), 0}, *d*(*C*(**M**), Q) + *r*(**M**)> tak, aby bol zoznam Wq zotriedený.

      }
   4. Pokračuj krokom č. 1.

   }

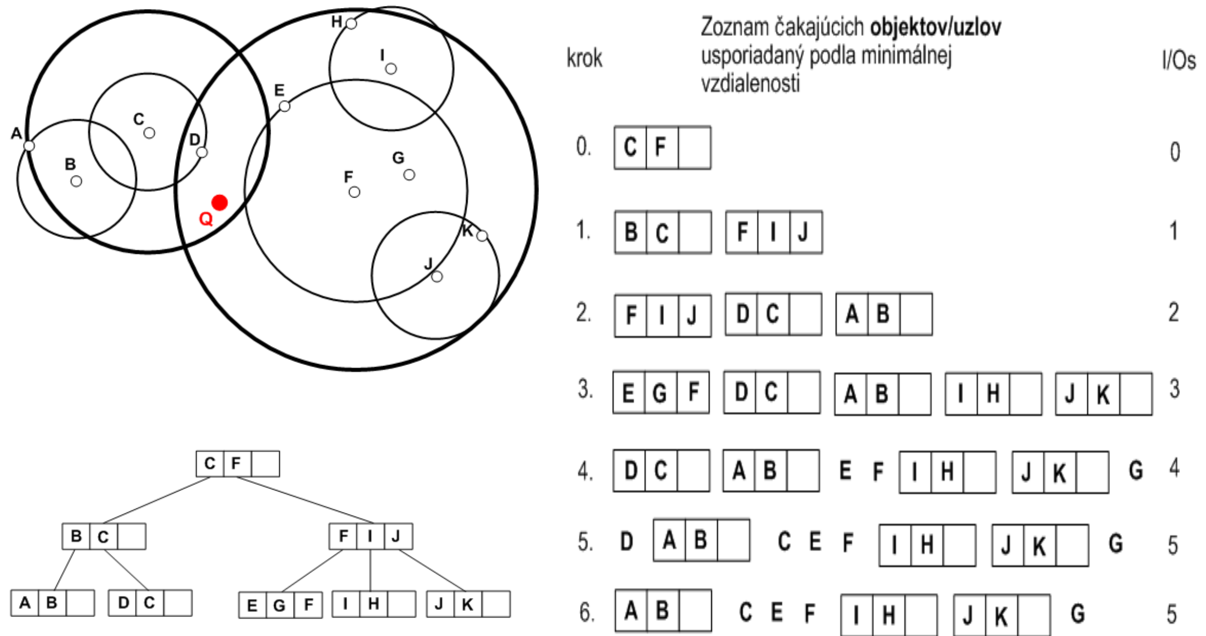
}

Veta: Algoritmus Sort query korektne vráti body stromu v poradí od najbližšieho po najvzdialenejší od kotvového bodu.

Dôkaz: Korektnosť algoritmu Sort query je zaručená spôsobom práce s pracovným zoznamom Wq. Inicializačná fáza algoritmu zaručí, že pracovný zoznam pokrýva na začiatku všetky body v strome. V iteračnej fáze, ak dôjde k odobratiu uzlového prvku z pracovného zoznamu, krok 3.2.1 (resp 3.3.1) iteračnej fázy zaručí, že pracovný zoznam opäť pokrýva všetky body v strome, ktoré ešte neboli vrátené na výstup krokom 2.1 iteračnej fázy. Pracovný zoznam môžeme týmto považovať za úplný. Úplnosť pracovného zoznamu a poradie jeho prvkov zase zaručí, že bod vrátený v kroku 2.1. iteračnej fázy je spomedzi všetkých bodov stromu, ktoré ešte neboli vrátené, najbližšie ku kotvovému bodu Q.

□

Ukážme ešte, že algoritmus Sort query vykoná po *k* volaniach metódy *sortQueryNext*, rovnako ako algoritmus kNN query, optimálny (teda minimálny nutný) počet prístupov na disk. Uzly čakajúce v pracovnom zozname sú v obidvoch algoritmoch zotriedené podľa ich minimálnej vzdialenosti od bodu Q. Uzly sú teda čítané z disku a následne rozbíjané na svoje poduzly (listy na svoje body) v rovankom poradí. Algoritmus kNN query skončí, ak je jeho pracovný zoznam prázdny, t.j. všetky relevantné uzly už boli prečítané. Algoritmus Sort query vráti *k*-ty najbližší bod, ak sa nachádza na prvom mieste v pracovnom zozname. Predtým však museli byť prečítané všetky uzly, ktoré, boli v zozname pred ním. Stačí si už len uvedomiť, že sú to práve tie relevantné uzly, ktoré prečítal aj algoritmus kNN query.



Obr. 3.3 Sekvenčný dopyt s kotvovým bodom Q.

Na obr. 3.3 je ukážka fungovania algoritmu Sort query. Známy obrázok vľavo predstavuje M-strom a konkrétny dopyt určený kotvovým bodom Q. Obrázok vpravo znázorňuje obsah pracovného zoznamu v priebehu výpočtu od inicializačnej fázy (0.-ty krok) až po návrat prvého bodu v kroku 6. V krokoch 1 až 6 je zachytený jeden beh iteračnej fázy, ktorej výsledkom je bod D, najbližší bod k bodu Q. Iteračná fáza vykonala celkom 5 krokov, kým sa na prvom mieste v pracovnom zozname objavil bod. V kroku 6 bol bod D vrátený na výstup a stav pracovného zoznamu v kroku 6 je východiskový stav pre ďalší beh iteračnej fázy.

3.5 Algoritmus Fuzzy sort query

Algoritmus Sort query dokázal vrátiť body len v poradí od najbližšieho po najvzdialenejší. V časti 1.2 sme uviedli, že je to použiteľné len pre špeciálny prípad, keď sú používateľove preferencie vyjadrené klesajúcou fuzzy funkciou. My však chceme dať používateľovi možnosť vyjadriť aj iné preferencie voči vzdialenosti. Toto dosiahneme pomerne malou úpravou algoritmu sort query, výsledkom je nový algoritmus Fuzzy sort query.

Tento algoritmus predstavuje preferenčný sekvenčný dopyt na najbližších susedov. Jeho parametrami sú kotvový bod Q a fuzzy funkcia *f*. Funkcia *f* musí každú vzdialenosť ohodnotiť reálnym číslom z intervalu $<0; 1>$. Algoritmus na ňu nekladie žiadne ďalšie požiadavky. Výstupom sú postupne body stromu v poradí od najvyššej po najnižšiu fuzzy hodnotu ich vzdialenosti od bodu Q. Takéto poradie bodov je vzhľadom na používateľove preferencie vzdialeností vyjadrené funkciou *f*, poradím od najlepšieho po najhorší.

Algoritmus Fuzzy sort query si tiež popíšeme dvoma metódami, jednou inicializačnou a druhou iteračnou. Inicializačná metóda *fuzzySortQueryInitialization* má dva parametre: kotvový bod Q a fuzzy funkciu *f*, a nevytvára žiaden výstup. Iteračná metóda *fuzzySortQueryNext* je bez parametrov a vráti jeden bod stromu a fuzzy hodnotu jeho vzdialenosti od bodu Q t.j. $<O, f(d(O, Q))>$. Postupným volaním metódy *fuzzySortQueryNext* získavame body stromu v poradí od najlepšieho po najhorší podľa funkcie *f* a bodu Q na vstupe inicializačnej metódy.

Algoritmus Fuzzy sort query sa od pôvodneho Sort query líši len pracovným zoznamom Wq. Ten bude teraz obsahovať nasledujúce prvky:

bodový prvok: $<O, f(d(O, Q)), f(d(O, Q))>$
uzlový prvok: $<p(\mathbf{N}), f_{best}(d(C(\mathbf{N}), Q) - r(\mathbf{N}), d(C(\mathbf{N}), Q) + r(\mathbf{N})),$
$f_{worst}(d(C(\mathbf{N}), Q) - r(\mathbf{N}), d(C(\mathbf{N}), Q) + r(\mathbf{N}))>$

O je bod stromu, **N** je uzol, Q je kotvový bod a *f* je fuzzy funkcia. Funkcie f_{best} a f_{worst} sú definované nasledovne: $f_{best}(a, b) = max\{ f(x) : a \le x \le b \}$, $f_{worst}(a, b) = min\{ f(x) : a \le x \le b \}$, kde *a*, *b* sú reálne čísla. Bodovým prvkom je trojica: bod a najlepšia a najhoršia fuzzy hodnota jeho vzdialenosti od bodu Q. Pre bod sú tieto fuzzy hodnoty rovnaké. Uzlovým prvkom je trojica: smerník na uzol **N** a najlepšia a najhoršia fuzzy hodnota vzdialenosti od Q, ktorú môže nejaký bod podstromu uzla **N** dosiahnuť.

Pracovný zoznam bude opäť zotriedený, teraz však zostupne podľa najlepšej fuzzy hodnoty svojich prvkov. Prvky s rovnakou najlepšou fuzzy hodnotou budú ešte usporiadané zostupne podľa najhoršej fuzzy hodnoty. Usporiadanie má byť navyše také, aby bodové prvky boli v zozname umiestnené vždy pred uzlovými prvkami s rovnakou najlepšou fuzzy hodnotou.

void ***fuzzySortQueryInitialization***(Q, *f*) {
1. Nech uzol **R** je koreň stromu.
2. Vlož do prázdneho zoznamu Wq prvok $<p(\mathbf{R}), 0, 0>$
3. Zapamätaj si bod Q a funkciu *f*.

}

$<O, f(d(O, Q))>$ ***fuzzySortQueryNext***() {
1. Nech $<x, y, z>$ je prvý prvok zoznamu Wq, odober z neho tento prvok.
2. Ak *x* je bod {
    1. Vráť $<x, y>$ a skonči.

    }
3. Ak *x* je smerník na uzol {
    1. Nech $p(\mathbf{N}) = x$, prečítaj uzol **N** z disku.
    2. Ak **N** je list {
        1. Pre každý bod O listu **N** pridaj do zoznamu Wq prvok $<O, f(d(O, Q)), f(d(O, Q))>$ tak, aby bol zoznam Wq zotriedený.

        }
    3. Ak **N** je vnútorný uzol {

1. Pre každý poduzol **M** uzla **N** pridaj do zoznamu Wq prvok $<p(\mathbf{M}), f_{best}(d(C(\mathbf{M}), Q) - r(\mathbf{M}), d(C(\mathbf{M}), Q) + r(\mathbf{M})), f_{worst}(d(C(\mathbf{M}), Q) - r(\mathbf{M}), d(C(\mathbf{M}), Q) + r(\mathbf{M}))>$ tak, aby bol zoznam Wq zotriedený.

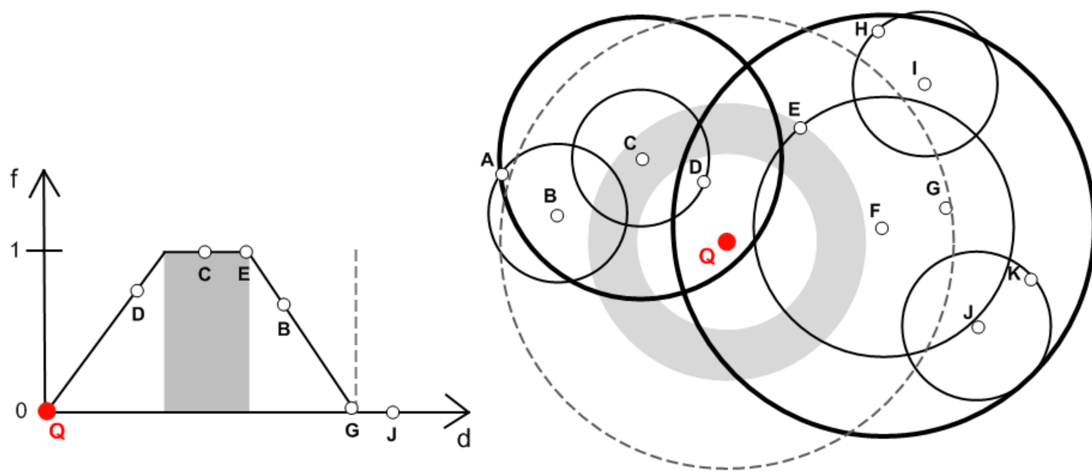
   }

4. Pokračuj krokom č. 1.

}

}

Je vidieť, že algoritmus Fuzzy sort query sa od Sort query líši len v tom, že vzdialenosti prepočítava na ich fuzzy hodnoty. Tie potom udržiava v prvkoch pracovného zoznamu, ktorý je podľa týchto hodnôt zotriedený od najlepšieho po najhorší. Korektnosť a optimalita v počte prístupov na disk je zachovaná vďaka poradiu prvkov v pracovnom v zozname, presnejšie použitím funkcií f_{best} a f_{worst}.



Obr. 3.4 Príklad sekvenčného dopytu na najbližších susedov od bodu Q s preferenciou vzdialeností podľa funkcie *f*.

4 Kvalita M-stromu

V tejto kapitole rozoberieme spôsoby vytvárania M-stromu s vyššou kvalitou indexácie. Popíšeme si význam tejto kvality, spôsob jej hodnotenia a algoritmy, ktoré ju zlepšujú. Najprv predstavím existujúce algoritmy Generalized slim down a Mulity way leaf choice, potom uvediem vlastné prístupy, ktorými sú virtuálne stredy uzlov a ich dynamické prepočítavanie.

4.1 Význam kvality M-stromu

Kvalita M-stromu má veľký vplyv na výkon dopytov. Ak je kvalita M-stromu nízka, dopyty nad takým stromom musia prečítať veľa uzlov (vykonať veľa prístpov na disk). Naopak kvalitný M-strom sa vyznačuje nízkym počtom prístupov na disk, ktorý musia dopyty

nad ním vykonať. Počty týchto prístupov na disk priamo súvisia s veľkosťami polomerov uzlov a s veľkosťami ich prienikov.

Nad jednou množinou bodov sa dá vytvoriť veľa rôznych M-stromov líšiacich sa rozdelením bodov do listov a rozdelením uzlov do svojich rodičovských uzlov. Povedzme, že budeme nad jednou množinou bodov osobitne vytvárať dva stromy algoritmami uvedenými v časti 2.3. Potom stačí, ak budeme do jedného stromu pridávať body v inom poradí, ako do druhého stromu. Takto vytvorené dva stromy sa môžu líšiť svojou kvalitou a tie isté dopyty nad jedným stromom môžu vykonať iný počet prístupov na disk ako nad druhým stromom. Výstupy týchto dopytov však budú rovnaké, lebo obidva stromy obsahujú tú istú množinu bodov. Samotná optimalita algoritmov (dopytov) v počte prístupov na disk teda ešte nezaručí, že výpočet pobeží rýchlo a nedôjde k prehľadaniu celého stromu. Algoritmy dopytov sú teda optimálne v počte prístupov na disk len vzhľadom na konkrétne hierarchicke usporiadanie bodov a uzlov konkrétneho M-stromu. Aby sme čo najviac využili túto ich optimalitu musíme nad danou množinou bodov vytvoriť čo najkvalitnejší M-strom.

V definícii M-stromu sa nehovorí o tom, ako majú byť body rozdelené do listov a uzly do svojich rodičovských uzlov. Z definície M-stromu vyplývajú len vlastnosti samotného stromu a jeho uzlov, ktoré musia byť pri vytváraní stromu (pridávaní bodov) zachované. Vytváranie stromu sme síce popísali v časti 2.3 konkrétnymi algoritmami, ale to neznamená, že strom nemôžeme vytvárať aj inými postupmi.

Základným cieľom pri vytváraní kvalitného M-stromu je rozdeliť body do listov a uzly do rodičovských uzlov tak, aby vznikli medzi uzlami tej istej úrovne čo najmensie prieniky. Všimnime si, že presne o to sa snažia už algoritmy uvedené v časti 2.3. Pri rozdeľovaní uzla sme vyberali také stredy nových uzlov a tak prerozdeľovali body, aby sme dosiahli čo najmenšie polomery nových uzlov. Podobne pri pridávaní bodu do stromu sme hľadali vhodný list tak, že sme prešli stromom od koreňa smerom k listu pozdĺž jednej vetvy cez také uzly, u ktorých po pridaní bodu nemusí dôjsť k zväčšeniu ich polomeru (prípadne dôjde k najmenšiemu zväčšeniu), aby ostali zachované všetky vlastnosti stromu. Použil sa teda základný predpoklad, že čím budú menšie polomery uzlov, tým budú menšie ich prieniky. Takto vytvorený strom však dosahuje len veľmi priemernú kvalitu. Nasledujúce postupy a algoritmy ukážu, ako vytvoriť kvalitnejší M-strom, prípadne ako zlepšiť kvalitu už existujúceho stromu.

4.2 Meranie kvality M-stromu

Skôr, ako sa pustíme do samotného zlepšovania kvality M-stromu, povedzme si, ako budeme kvalitu M-stromu merať. Aby sme mohli objektívne posúdiť vplyvy jednotlivých postupov na kvalitu M-stromu, potrebujeme nejaké ukazovatele, pomocou ktorých môžeme kvalitu stromu posudzovať. Takýmto ukazovateľom je faktor tučnosti stromu [6].

Faktor tučnosti, označme ho φ, je reálne číslo z intervalu <0; 1> a objektívne vypovedá o kvalite stromu bez ohľadu na špecifické vlastnosti stromu (napr. použitý metrický priestor). Čím je faktor menší, tým je strom štíhlejší a z pohľadu kvality lepší. Pre presnú predstavu rozoberme vzorec, podľa ktorého sa faktor tučnosti počíta:

$$\varphi = \frac{I - hn}{n(m-h)} \tag{4.1}$$

Číslo h je počet úrovní stromu (výška stromu), n je počet bodov uložených v strome, m je počet uzlov stromu a I je celkový počet prístupov na disk vykonaných dokopy za všetky

bodové rozsahové dopyty pre body uložené v strome. Hodnotu I teda získame tak, že pre každý bod O uložený v strome vykonáme rozsahový dopyt so stredom O a nulovým polomerom a spočítame prístupy na disk spolu za všetky takto vykonané dopyty.

Ideálny faktor 0 sa dosiahne vtedy, keď $I = hn$, čo je najmenšia možná dosiahnuteľná hodnota I. To je možné len tak, že každý z n vykonaných dopytov vykoná presne h prístupov na disk (prečíta h uzlov stromu pozdĺž jedinej vetvy). V takomto strome buď uzly rovnakých úrovní nemajú žiadne prieniky, alebo ich prieniky neobsahujú indexované body.

Najhorší faktor 1 sa dosiahne vtedy, keď $I = mn$, čo je najväčšia možná dosiahnuteľná hodnota I. Tento prípad nastane, keď každý z n vykonaných dopytov prečíta všetky uzly stromu. V takomto strome každý indexovaný bod patrí do gule každého uzla stromu.

4.3 Algoritmus Node sieve

Skôr ako uvediem algoritmy Multi way leaf choice a Generalized slim down (uvedené tiež v [6]), potrebujeme sa oboznámiť s algoritmom Node sieve, ktorý je nimi používaný. Tento algoritmus je tiež opísaný v [6], avšak v tomto opise bude trochu vylepšený a upresnený.

Už názov algoritmu Node sieve naznačuje, že pôjde o princíp podobný situ. Jednotlivé úrovne stromu používa ako sitá, ktorých diery sú uzly danej úrovne. Hlavným vstupom algoritmu je guľa (v prípade nulového polomeru bod) a výstupom sú uzly požadovanej úrovne, do ktorých guľa na vstupe dokázala postupne prepadnúť cez vyššie úrovne stromu ako cez sitá. Je tu vysoká podobnosť s algoritmom Range query, ktorý mal tiež na vstupe guľu. Algoritmus Range query na každej úrovni prečítal a rekurzívne sa vnoril do tých uzlov, ktoré mali neprázdny prienik s guľou na vstupe (s guľou dopytu). Algoritmus Node sieve na každej úrovni prečíta a rekurzívne sa vnorí do tých uzlov, ktorých prienik s guľou na vstupe je celá samotná guľa na vstupe. Na každej úrovni teda prečíta a rekurzívne sa vnorí do tých uzlov, cez ktoré ako cez diery v site guľa na vstupe bezozbytku prepadne. Algoritmus Range query zliezol vždy až na listovú úroveň a vrátil body patriace do gule dopytu. Algoritmus Node sieve sa zastaví na ľubovoľnej požadovanej úrovni (s vínimkou prvej úrovne, kde je len koreň stromu) a vráti tie uzly stromu, cez ktoré by guľa na vstupe dokázala ďalej ako cez diery v site bezozbytku prepadnúť. Výstupom algoritmu sú teda uzly jednej konkrétnej úrovne, ktoré obsahujú celú guľu na vstupe, a ktorých všetky rodičovské uzly až po koreň stromu tiež obsahujú celú guľu na vstupe.

Algoritmus Node sieve si popíšeme metódou *nodeSieve*, ktorá má 6 parametrov na vstupe. Su to: bod Q a číslo q určujúce stred a polomer gule, vnútorný uzol **N**, na ktorý je metóda aplikovaná, číslo $v = d(\mathrm{Q}, C(\mathbf{N}))$, ktoré má ten istý význam a využitie ako v metóde *rangeQuery*, a napokon dve celé čísla *currentLevel* a *targetLevel* určujúce aktuálnu úroveň uzla **N** a cieľovú úroveň, ktorej uzly hľadáme. Výstupom metódy je zoznam smerníkov na nájdené uzly, a vzdieleností ich stredov od bodu Q, t.j. zoznam obsahujúci prvky typu: $<p(\mathbf{M}), d(\mathrm{Q}, C(\mathbf{M}))>$. Budeme ho ako obyčajne nazývať výsledný zoznam a označovať Rq.

Rq ***nodeSieve***(Q, q, **N**, $v = d(\mathrm{Q}, C(\mathbf{N}))$, *currentLevel*, *targetLevel*) {

1. Nech zoznam Rq je prázdny.
2. Nech $\mathbf{B} = \{ \mathbf{M} : |d(C(\mathbf{M}), C(\mathbf{N})) - v| + q \le r(\mathbf{M})$ & $\boldsymbol{P}(\mathbf{M}) = \mathbf{N} \}$.
3. Pre všetky uzly $\mathbf{M} \in \mathbf{B}$ vypočítaj $d(\mathrm{Q}, C(\mathbf{M}))$.
4. Nech $\mathbf{B^*} = \{ \mathbf{M} : \mathbf{M} \in \mathbf{B}$ & $d(\mathrm{Q}, C(\mathbf{M})) + q \le r(\mathbf{M}) \}$.
5. Ak *currentLevel* + 1 = *targetLevel* {
    1. Pre každý uzol $\mathbf{M} \in \mathbf{B^*}$ pridaj do zoznamu Rq prvok $<p(\mathbf{M}), d(\mathrm{Q}, C(\mathbf{M}))>$.

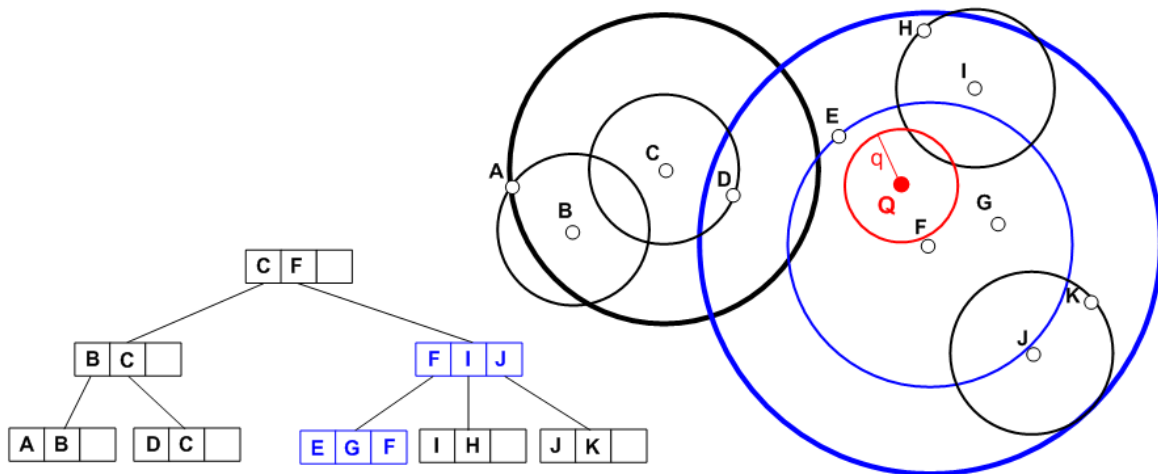
    }

6. Ak *currentLevel* + 1 < *targetLevel* {
   1. Pre každý uzol $\mathbf{M} \in \mathbf{B}^*$ vykonaj {
      1. Prečítaj uzol **M** z disku.
      2. Nech Hq = ***nodeSieve***(Q, *q*, **M**, *d*(Q, *C*(**M**)), *currentLevel* + 1, *targetLevel*).
      3. Pridaj do zoznamu Rq všetky prvky zoznamu Hq.

   }
7. Vráť zoznam Rq a skonči.

}

Výpočet algoritmu Node sieve odštartujeme volaním ***nodeSieve***(Q, *q*, **R**, 0, 1, *targetLevel*), kde **R** je koreň stromu. Je zrejmé, že metódu *nodeSieve* nemôžeme aplikovať na list, lebo ten nemá žiadne poduzly. Algoritmus node sieve môžeme teda aplikovať len na strom výšky minimálne 2. Ak *h* je výška stromu, tak pre cieľovú úroveň, ktorej uzly hľadáme musí platiť: $1 < targetLevel \leq h$.



Obr. 4.1 Algoritmus Node sieve

Na obrázku 4.1 je znázornená situácia, kde vstup pre algoritmus Node sieve tvorí bod Q, polomer *q* a *targetLevel* nastavený na hodnotu 3, t.j. na listovú úroveň. Modrou farbou sú zvýraznené všetky uzly, ktoré sa dostali do množiny **B*** (vrámci svojej úrovne). Sú to uzly, ktoré bezozbytku obsahujú celú guľu so stredom Q a polomerom *q*. Výstupom algoritmu sú teda uzly 3. úrovne (listy) zvýraznené modrou farbou. V našom prípade výsledný zoznam Rq bude obsahovať len jeden list obsahujúci body E, G, F.

4.4 Algoritmus Multi way leaf choice

Pridanie bodu do stromu si vyžaduje nájdenie vhodného listu. Metóda *findLeaf* uvedená v kapitole 2 nájde list tak, že prejde stromom od koreňa po list pozdĺž jednej vetvy. Jej rozhodovanie je na každej úrovni len lokálne a neberie do úvahy usporiadanie uzlov na nižších úrovniach. Optimálny list sa teda vyberie len zriedkavo. Metóda *findLeaf* predstavuje algoritmus Single way leaf choice. Algoritmus Multi way leaf choice [6] sa snaží vybrať naozaj optimálny list.

Algoritmus si popíšeme metódou *multiWayLeafChoice*, ktorej jediným vstupom je pridávaný bod O a výstupom je list **L**.

L ***multiWayLeafChoice***(O) {

1. Nech **R** je koreň stromu a h je výška stromu.
2. Nech Rq = ***nodeSieve***(O, 0, **R**, 0, 1, h).
3. Ak zoznam Rq je prázdny {
    1. Vráť výsledok volania ***findLeaf***(O, **R**) a skonči.

    }
4. Ak zoznam Rq nie je prázdny {
    1. Nech **L** je taký list, že platí: $<p(\mathbf{L}), d(O, C(\mathbf{L}))> \in Rq$ &
       $d(O, C(\mathbf{L})) = min\{ d(O, C(\mathbf{N})) : <p(\mathbf{N}), d(O, C(\mathbf{N}))> \in Rq \}$.
    2. Vráť list **L** a skonči.

    }

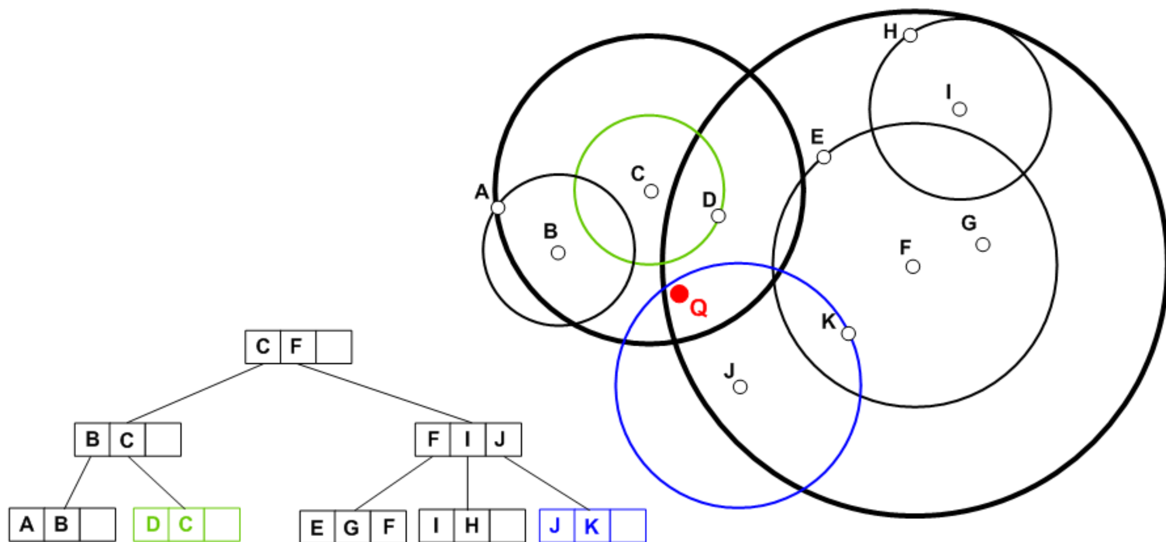
}

Algoritmus Multi way leaf choice sa najprv snaží nájsť list algoritmom Node sieve. Ak takto nájdený list ešte nie je plný, tak po pridaní bodu sa nemusí rozdeliť a celé pridanie bodu nespôsobí zväčšenie polomeru žiadneho uzla. Ak však algoritmom Node sieve nenájdeme žiadny vyhovujúci list, musíme nájsť vhodný list pôvodným spôsobom a to algoritmom Single way leaf choice (metódou *findLeaf*).

Všimnime si, že vzdialenosti uložené v prvkoch zoznamu Rq sú využité a výber listu **L** v kroku 4.1. sa zaobíde bez jediného výpočtu vzdialenosti (a samozrejme tiež bez jediného prístupu na disk). Voliteľne však môžeme krok 4.1 nahradiť nasledujúcim krokom:

4.1. Nech **L** je taký list, že platí:
$<p(\mathbf{L}), d(O, C(\mathbf{L}))> \in Rq$ &
$|\boldsymbol{D}(\mathbf{L})|$ je menšia ako povolená kapacita &
$d(O, C(\mathbf{L})) = min\{ d(O, C(\mathbf{N})) : <p(\mathbf{N}), d(O, C(\mathbf{N}))> \in Rq$ & $|\boldsymbol{D}(\mathbf{N})|$ je menšia ako povolená kapacita }.

V takto zmenenom kroku 4.1. sa výber listu **L** opäť zaobíde bez výpočtu vzdialenosti, ale nezaobíde sa bez prístupu na disk. Uzly neobsahujú žiadne informácie o zaplnení svojich poduzlov. Nie je teda možné aby informáciu o zaplnení uzlov poskytol výstup algoritmu Node sieve tak, ako poskytol už raz vypočítané vzdialenosti bodu O od stredov nájdených uzlov. Resp. nie je to možné bez toho, aby sám algoritmus Node sieve neprečítal príslušné uzly z disku. Tento výber uzla **L** teda vyžaduje v najlepšom prípade 1, v najhoršom prípade až |Rq| prístupov na disk. V najhoršom prípade sa dokonca môže stať, že všetky nájdené listy budú plné, vtedy musíme vybrať list **L** pôvodnym krokom 4.1. a rozdeleniu listu sa nevyhneme. Úprava kroku 4.1. vedie k väčšiemu (lepšiemu) priemernému zaplneniu listov, vyžaduje však viac prístupov na disk.



Obr. 4.2 Porovnanie výsledkov hľadania listu metódou *multiWayLeafChoice* a *findLeaf*.

Obrázok 4.2 znázorňuje typický prípad, kedy algoritmom Multi way leaf choice nájdeme optimálny list pre pridanie bodu Q (zvýraznený modro), zatiaľ čo algoritmom Single way leaf choice nájdeme iný list (zvýraznený zeleno), ktorému by sa po pridaní bodu Q musel zväčšiť polomer.

4.5 Algoritmus Generalized slim down

Vytváranie M-stromu postupným pridávaním bodov je navrhnuté tak, aby sme dosiahli čo najlepšiu kvalitu indexu. Takto vytvorený strom však stále nie je ani zďaleka optimálny. Jeho kvalitu dokážeme zlepšiť poprehadzovaním niektorých bodov (na listovej úrovni) a uzlov (na vyšších úrovniach) do vhodnejších rodičovských uzlov. Tým dôjde k zmenšeniu polomeru u niektorých uzlov. Na to slúži algoritmus Generalized slim down (skrátene Slim down) [6].

Tento algoritmus hľadá postupne pre všetky body vo všetkých listoch iný vhodnejší list, do ktorého by mohol daný bod prehodiť. Pre list, z ktorého bod odoberie, prepočíta nový polomer, pričom nanajvýš môže dôjsť k jeho zmenšeniu. List, do ktorého bod pridá, je vybraný tak, že nedochádza u neho k žiadnej ďalšej zmene. Tento postup je aplikovaný postupne na všetky úrovne stromu (s výnimkou prvej úrovne, kde je len koreň) v poradí od listovej úrovne nahor. Pre uzly vyšších úrovní hľadá vhodnejšie rodičovské uzly a nanajvýš môže dôjsť k zmenšeniu polomeru uzla, ktorému bol odobraný nejaký jeho poduzol.

Algoritmus Generalized slim down si popíšeme metódou *slimDown*, ktorá nemá žiadne vstupy ani výstupy.

void ***slimDown***() {
1. Nech **R** je koreň stromu, h je výška stromu a nech $k = h$.
2. Pokiaľ $k > 1$ opakuj {
    1. Ak $k = h$ {
        1. Pre každý list **L** stromu vykonaj {
            1. Pre každý bod O listu **L** vykonaj {
                1. Nech Rq = ***nodeSieve***(O, 0, **R**, 0, 1, h).
                2. Nech **L’** je taký list, že platí:

$\langle p(\mathbf{L'}), d(O, C(\mathbf{L'}))\rangle \in Rq$ &
$|\boldsymbol{D}(\mathbf{L'})|$ je menšia ako povolená kapacita &
$d(O, C(\mathbf{L'})) = \mathit{min}\{ d(O, C(\mathbf{N})) :$
$\langle p(\mathbf{N}), d(O, C(\mathbf{N}))\rangle \in Rq$ & $|\boldsymbol{D}(\mathbf{N})|$ je menšia ako povolená kapacita }.

3. Ak taký **L'** existuje a $\mathbf{L'} \neq \mathbf{L}$ {

1. Odober záznam $\langle O, d(O, C(\mathbf{L}))\rangle$ z množiny $\boldsymbol{D}(\mathbf{L})$ a pridaj do množiny $\boldsymbol{D}(\mathbf{L'})$ záznam $\langle O, d(O, C(\mathbf{L'}))\rangle$.
2. Ak je to možné, zmenši čo najviac polomer listu **L** a následne zdola nahor polomery všetkých jeho naduzlov.

}

}

}

}

2. Ak $k < h$ {

1. Pre každý uzol **U** *k*-tej úrovne vykonaj {

1. Pre každý poduzol **V** uzla **U** vykonaj {

1. Nech $Rq =$ ***nodeSieve***$(C(\mathbf{V}), r(\mathbf{V}), \mathbf{R}, 0, 1, k)$.
2. Nech **U'** je taký uzol, že platí:
$\langle p(\mathbf{U'}), d(C(\mathbf{V}), C(\mathbf{U'}))\rangle \in Rq$ &
$|\boldsymbol{D}(\mathbf{U'})|$ je menšia ako povolená kapacita &
$d(C(\mathbf{V}), C(\mathbf{U'})) + r(\mathbf{V}) = \mathit{min}\{ d(C(\mathbf{V}), C(\mathbf{N})) + r(\mathbf{V}) :$
$\langle p(\mathbf{N}), d(C(\mathbf{V}), C(\mathbf{N}))\rangle \in Rq$ & $|\boldsymbol{D}(\mathbf{N})|$ je menšia ako povolená kapacita }.
3. Ak taký **U'** existuje a $\mathbf{U'} \neq \mathbf{U}$ {

1. Odober záznam $\langle C(\mathbf{V}), d(C(\mathbf{V}), C(\mathbf{U})), r(\mathbf{V}), p(\mathbf{V})\rangle$ z množiny $\boldsymbol{D}(\mathbf{U})$ a pridaj do množiny $\boldsymbol{D}(\mathbf{U'})$ záznam $\langle C(\mathbf{V}), d(C(\mathbf{V}), C(\mathbf{U'})), r(\mathbf{V}), p(\mathbf{V})\rangle$.
2. Zmeň uzlu **V** smerník na rodiča z pôvodného $p(\mathbf{U})$ na $p(\mathbf{U'})$.
3. Ak je to možné, zmenši čo najviac polomer uzla **U** a následne zdola nahor polomery všetkých jeho naduzlov.

}

}

}

}

3. Nech $k = k - 1$.

}

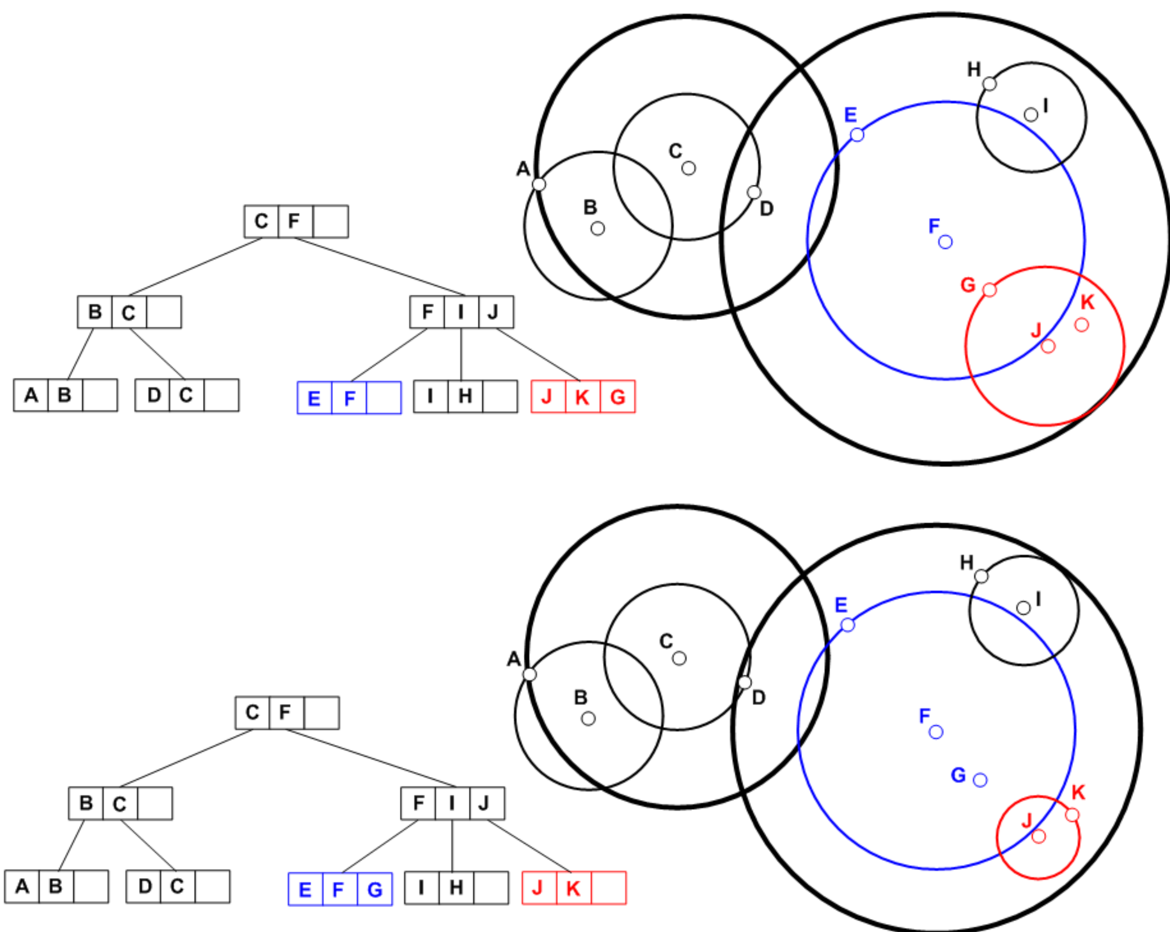
}

Krok 2.1.1.1.3.1 odoberie bod O listu **L** a pridá ho listu **L'** a krok 2.2.1.1.3.1 zase odoberie poduzol **V** uzlu **U** a pridá ho uzlu **U'**. Dôležité je to, že tento presun bodu (alebo poduzla) do iného uzla nevyžaduje žiadne zmeny stredov a polomerov dotknutých uzlov. Je to zaručené tým, že uzol, do ktorého bod (alebo poduzol) presúvame sme našli algoritmom Node sieve. Je len v našom záujme, že sa po presune bodu (alebo poduzla) snažíme zmenšiť polomer uzla, z ktorého sme bod (alebo poduzol) odobrali. Ak tomu uzlu nájdeme menší polomer, má

význam skúsiť prepočítať polomer aj jeho rodičovského uzla. Takto postupujeme zdola nahor a prepočítavame nové polomery (pra)rodičovských uzlov, až kým neprídeme na prvú uroveň alebo kým nenastane to, že nový polomer nie je menší ako pôvodný.

Algoritmus Generalized slim down vo svojej základnej verzii prejde takto každú úroveň stromu práve raz. Voliteľne však môžeme proces na jednotlivých úrovniach opakovať dovtedy, pokiaľ dochádza k presunom. Ak už k presunu na danej úrovni nedôjde, až vtedy ideme na vyššiu úroveň.

Tento algoritmus môže na veľmi košatom strome bežať netriviálne dlho. Treba ale poznamenať, že tento algoritmus môže bežať ako samostatný proces na pozadí v čase nečinnosti medzi výpočtami jednotlivých dopytov. Až na niekoľko kritických sekcií (t.j. len samotné presuny) ho môžeme kedykoľvek prerušiť a opätovne spustiť od miesta posledného prerušenia. Všetky vlastnosti stromu sú počas behu algoritmu Generalized slim down (aj po ňom) zachované. K zmene dochádza len v hierarchickom usporiadaní uzlov (a bodov) a v polomeroch uzlov, ktoré sa môžu nanajvýš iba ak zmenšiť.



Obr. 4.3 Príklad aplikácie algoritmu Slim down.

V hornej časti obrázku 4.3 je M-strom, na ktorý bol aplikovaný algoritmus Slim down, v dolnej časti obrázku 4.3 je tento strom už po zbehnutí algoritmu. Na listovej úrovni jedine bod G je bližšie k stredu susedného uzla (k bodu F) ako k stredu svojho vlastného uzla (bodu J), preto na listovej úrovni došlo k jednému presunu. Na vyšších úrovniach už v tomto triviálnom príklade k presunom nedošlo. V dôsledku presunu bodu G došlo k zmenšeniu polomeru listu so stredom J (zvýraznený červene), vďaka čomu sa mohol zmenšiť polomer aj jeho rodičovského uzla so stredom F. Zmenšovanie polomerov rodičovských uzlov je

podrobnejšie rozobrané v časti 4.7 venovanej dynamickému prepočítavaniu stredov a polomerov uzlov.

4.6 Virtuálne stredy uzlov

M-strom je pôvodne navrhnutý tak, že stred listu je jeden z bodov uložených v liste a stred vnútorného uzla je stred jedného z jeho potomkov. Tento prístup vedie k tomu, že len výnimočne sa nájde optimálny stred a polomer uzla.

V definícii uzla stromu v kapitole 2 by bolo možné stred uzla **N** určiť len odkazom na jeden z uložených bodov v záznamoch množiny ***D***(**N**). Stred uzla je však definovaný nezávisle od ostatného obsahu uzla. Vďaka tomu nie sme (ani z logického hľadiska) nijako obmedzení v určovaní stredu uzla. Pokúsime sa teda pre uzly nájsť optimálne alebo aspoň skoro optimálne stredy a polomery.

4.6.1 Algoritmus Submininimal bounding hyperball

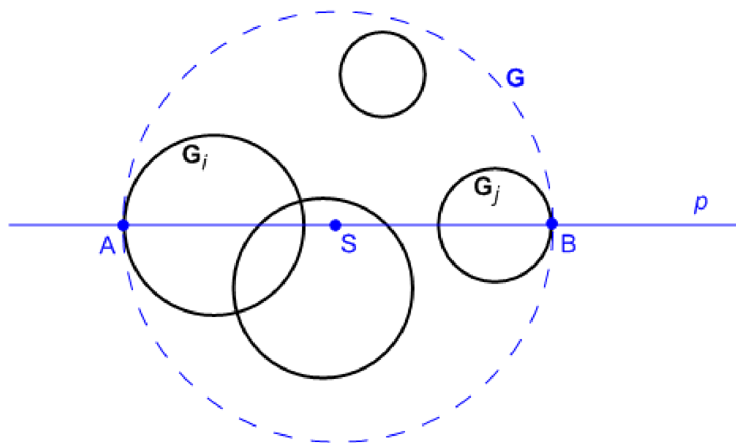
Vo všeobecnosti stojíme pred problémom nájsť minimálnu obaľujúcu n-rozmernú guľu pre konečnú množinu n-rozmerných gúľ v n-rozmernom metrickom priestore. Špeciálne pre listy stromu treba nájsť minimálnu obaľujúcu n-rozmernú guľu pre konečnú množinu bodov. Je zrejmé, že tento problém treba riešiť pre rôzne metrické priestory samostatne. Ďalej sa preto budeme zaoberať len euklidovským priestorom. Nájsť minimálnu obaľujúcu guľu pre množinu bodov v euklidovskej rovine nie je problém. V časti 4.6.2 uvediem algoritmus, ktorý ju nájde. Nájsť takú guľu pre viacrozmerný euklidovský priestor už problém je. Ak však upustíme od požiadavky, že hľadaná guľa musí byť minimálna, tak dokážeme nájsť aspoň skoro-minimálnu obaľujúcu guľu a to nie len pre množinu bodov v n-rozmernom euklidovskom priestore, ale aj pre množinu n-rozmerných gúľ v n-rozmernom euklidovskom priestore. Nájdenie takej skoro-minimálnej obaľujúcej gule si popíšeme novým algoritmom Submininimal bounding hyperball, ktorý má na vstupe číslo n vyjadrujúce dimenzionalitu priestoru, a množinu n-rozmerných gúľ. Výstupom algoritmu je stred a polomer nájdenej gule. Guľu **G** budeme zapisovať usporiadanou dvojicou $\mathbf{G} = \langle C(\mathbf{G}), r(\mathbf{G})\rangle$, kde $C(\mathbf{G})$ je stred gule **G** a $r(\mathbf{G})$ je jej polomer.

G ***subminimalBoundingHyperball***$(n, \{\mathbf{G}_1, \dots, \mathbf{G}_m\})$ {

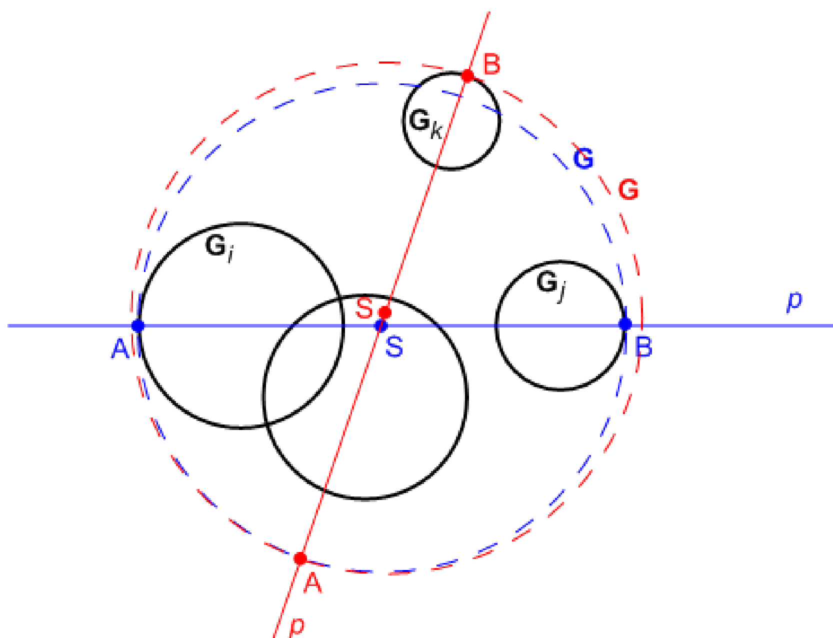
1. Nájdi i, j také, že $d(C(\mathbf{G}_i), C(\mathbf{G}_j)) + r(\mathbf{G}_i) + r(\mathbf{G}_j) = max\{d(C(\mathbf{G}_k), C(\mathbf{G}_l)) + r(\mathbf{G}_k) + r(\mathbf{G}_l) : k, l \in \{1, \dots, m\}\}$.
2. Nech p je priamka určená bodmi $C(\mathbf{G}_i)$ a $C(\mathbf{G}_j)$.
3. Nech body A, B sú také, že $A \in p \cap \mathbf{G}_i$, $B \in p \cap \mathbf{G}_j$ a $d(A, B) = d(C(\mathbf{G}_i), C(\mathbf{G}_j)) + r(\mathbf{G}_i) + r(\mathbf{G}_j)$.
4. Nech bod S je stred úsečky AB.
5. Nech guľa $\mathbf{G} = \langle S, |AB| / 2\rangle$.
6. Pre každé $k \in \{1, \dots, m\} - \{i, j\}$ vykonaj {
    1. Ak $d(C(\mathbf{G}), C(\mathbf{G}_k)) + r(\mathbf{G}_k) > r(\mathbf{G})$ {
        1. Nech p je priamka určená bodmi $C(\mathbf{G})$ a $C(\mathbf{G}_k)$.
        2. Nech body A, B sú také, že $A \in p \cap \mathbf{G}$, $B \in p \cap \mathbf{G}_k$ a $d(A, B) = d(C(\mathbf{G}), C(\mathbf{G}_k)) + r(\mathbf{G}) + r(\mathbf{G}_k)$.
        3. Nech bod S je stred úsečky AB.
        4. Nech guľa $\mathbf{G} = \langle S, |AB| / 2\rangle$.

        }

}
7. Vráť na výstup guľu **G** a skonči.
}



Obr. 4.4 prípad nájdenia minimálnej obaľujúcej gule (kroky 1 až 5)



Obr. 4.5 prípad nájdenia skoro-minimálnej obaľujúcej gule (modrá farba znázorňuje kroky 1 až 5, červená farba znázorňuje kroky 6.1.1 až 6.1.4)

Algoritmus najprv nájde dve najvzdialenejšie gule $\mathbf{G}_i$ a $\mathbf{G}_j$ a vypočíta prene minimálnu obaľujúcu guľu **G**. Pre prípad množiny vstupných gúľ na obrázku 4.4 sa táto guľa stáva riešením a kroky 6.1.1 až 6.1.4 sa nevykonajú. Nájdená guľa je minimálna. Môže ale nastať prípad na obrázku 4.5, ktorý vyriešime tak, že nájdeme novú priamku *p*, novú úsečku AB a nakoniec novú guľu **G**. Takto prídeme o minimalitu gule, ale novú guľu môžeme považovať sa skoro-minimálnu.

Algoritmus Submininimal bounding hyperball vieme aplikovať na *n*-rozmerný euklidovský priestor, lebo jednotlivé súradnice bodov A, B, S vieme vypočítať navzájom nezávisle, t.j. nezávisle od dimenzionality priestoru. Výpočet súradníc teda môžeme počítať rovnako bez ohľadu na hodnotu *n*. Napríklad *x*-ovú súradnicu bodu A (označme A.*x*) vypočítame v kroku 3 vzťahom:

$$A.x = C(\mathbf{G}_i).x + r(\mathbf{G}_i) * (C(\mathbf{G}_i).x - C(\mathbf{G}_j).x) / d(C(\mathbf{G}_i), C(\mathbf{G}_j)) \quad (4.2)$$

Zámenou všetkých x za y vo vzťahu 4.2 dostaneme vzťat pre y-ovú súradnicu bodu A a rovnako by sme postupovali pri výpočte zvyšných $n - 2$ súradníc bodu A. Na druhej strane zámenou všetkých i za j a opačne vo vzťahu 4.2 dostaneme vzťah pre výpočet x-ovej súradnice bodu B. Pre úplnosť ešte dodajme, že súradnice bodu S počítame aritmetickým priemerom príslušných súradníc bodov A a B.

4.6.2 Algoritmus Minidisk

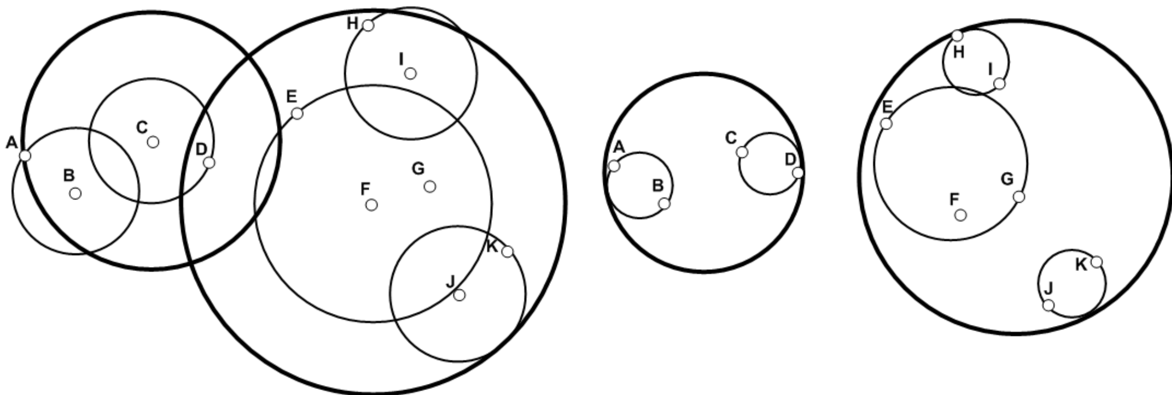
Špeciálnym prípadom rozoberaného problému je hľadanie minimálnej obaľujúcej kružnice pre body (listu stromu) v euklidovskej rovine. Pre tento účel predstavím pravdepodobnostný algoritmus Minidisk, ktorý podľa [10] nájde jedinú existujúcu minimálnu obaľujúcu kružnicu pre n bodov v rovine, a to priemerne v čase $O(n)$. Algoritmus Minidisk má na vstupe dve množiny bodov **P** a **R**. Množina **P** obsahuje body, pre ktoré hľadáme minimálnu obaľujúcu kružnicu a množina **R** obsahuje body, o ktorých vieme, že ležia na hľadanej kružnici. Výstupom algoritmu je kružnica K, ktorú podobne ako guľu budeme reprezentovať usporiadanou dvojicou <$C(K)$, $r(K)$> obsahujúcou jej stred a polomer.

```
K minidisk(P, R) {
1.      Ak P = Ø {
        1.      Ak R = Ø vráť null a skonči.
        2.      Ak R = {A} vráť K = <A, 0> a skonči.
        3.      Ak R = {A, B} {
                1.      Nech S je stred úsečky AB.
                2.      Vráť K = <S, |AB| / 2> a skonči.
                }
        4.      Ak R = {A, B, C} {
                1.      Nech K je kružnica opísaná trojuholníku ABC.
                2.      Vráť K a skonči.
                }
        }
2.      Ak P ≠ Ø {
        1.      Ak R = {A, B, C} {
                1.      Nech K je kružnica opísaná trojuholníku ABC.
                2.      Vráť K a skonči.
                }
        2.      Ak |R| < 3 {
                1.      Zvoľ náhodne bod T z množiny P.
                2.      Nech K = minidisk(P – {T}, R).
                3.      Ak K = null alebo d(T, C(K)) > r(K) {
                        1.      Nech K = minidisk(P – {T}, R ∪ {T}).
                        }
                4.      Vráť K a skonči.
                }
        }
}
```

Hľadanú kružnicu pre množinu bodov **P** získame volaním ***minidisk***(**P**, Ø). Algoritmus Minidisk je rekurzívny a vnorenie v kroku 2.2.2 využíva redukciu problému z n bodov na n – 1 bodov. Ak náhodne zvolený bod T leží v kruhu kružnice K pre zvyšných n – 1 bodov, tak kružnica K je riešením aj pre pôvodnu množinu n bodov (vrátane T). Vnorenie v kroku 2.2.3.1 zase využíva fakt, že ak náhodne zvolený bod T neleží v kruhu kružnice pre zvyšných n – 1 bodov, tak potom bod T určite leží na hľadanej kružnici pre všetkých n bodov. Týmto sme sfixovali jeden stupeň voľnosti nášho problému. Takto pribúdajú body do množiny **R** a ak množina **R** obsahuje tri body, hľadaná kružnica pre všetkých n bodov je určená práve bodmi v množine **R**.

4.6.3 Rozdiely vyplývajúce z virtuálnych stredov

Stredy uzlov sa počítajú pri delení uzla. Kým sme nepočítali virtuálne stredy, tak sme skúšali každú dvojicu možných stredov z množiny existujúcich bodov. Pre zvolenú dvojicu stredov sme rozdelili zvyšné body do dvoch množín a vypočítali polomery nových uzlov. Týmto postupom sme vybrali najlepšiu dvojicu nových stredov. S virtuálnymi stredmi je situácia iná. Body musíme najprv rozdeliť a až potom pre obidve množiny bodov vypočítať virtuálne stredy. Rozdelenie bodov môžeme vykonať tak, že nájdeme dva najvzdialenejsie body A a B a ostatné body pridelíme do jednej alebo druhej množiny podľa toho, ku ktorému z bodov A, B sú bližšie. Nezmeneným postupom vyriešime situáciu, kedy veľkosť jednej z množín nedosahuje minimálne zaplnenie uzla.



Obr. 4.6 Porovnanie M-stromu s klasickými a virtuálnymi stredmi

Obrázok 4.6 znázorňuje dva M-stromy obsahujúce tie isté body, pričom strom vpravo používa virtuálne stredy uzlov. Obrázok budí dojem, že virtuálne stredy uzlov prinášajú drammatické zlepšenie kvality indexácie. Pri väčších kapacitách uzla však toto zlepšenie kvality stromu nie je také výrazné, resp nie je žiadne. Konkrétne závery o zlepšení kvality indexácie obsahuje časť 6.2 venovaná testom kvality M-stromu.

4.7 Dynamické prepočítavanie stredov

Pri vytváraní stromu sme stredy uzlov doteraz počítali len pri delení uzlov. Bezprostredne po rozdelení uzla sme získali dva uzly, o ktorých sme mohli tvrdiť, že majú najlepšie stredy a polomery, aké sme dokázali popísanými postupmi nájsť. Ďalšie pridávanie bodov do stromu môže tento fakt porušiť. Preto je vhodné pokúsiť sa prepočítať nový stred a

polomer uzla vždy, keď sa zmení jeho obsah t.j. keď pre uzol **N** dôjde k zmene v množine ***D***(**N**).

S prepočítavaním stredov a polomerov uzlov sme sa už stretli v algoritme Generalized slim down a to konkrétne v krokoch 2.1.1.1.3.2. a 2.2.1.1.3.3. metódy *slimDown*. Presne tento istý postup prepočítavania stredov a polomerov môžeme použiť aj v ľubovoľnom inom prípade kedy dôjde k zmene v obsahu uzla a jeho pôvodny stred a polomer už nie sú najlepšie, aké dokážeme pre ten uzol nájsť. Najprv si podrobne popíšeme činnosť v spomínaných krokoch metódou *recomputeCenter* a potom si ukážeme využitie tejto metódy pri pridávaní bodov do stromu.

Metóda *recomputeCenter* má jediný vstup, ktorým je uzol, ktorého stred a polomer má byť prepočítaný, výstup nemá žiadny.

void ***recomputeCenter***(**N**) {
1. Nájdi pre množinu ***D***(**N**) optimálny stred S a polomer *s*.
2. Ak *s* < *r*(**N**) {
    1. Nastav aktuálny polomer *r*(**N**) na hodnotu *s*.
    2. Ak S ≠ *C*(**N**) {
        1. Nastav aktuálny stred *C*(**N**) na bod S.
        2. V záznamoch množiny ***D***(**N**) prepočítaj uložené vzdialenosti voči novému stredu S.

        }
    3. Zaeviduj zmenu polomeru a prípadnú zmenu stredu uzla **N** jeho rodičovskému uzlu, t.j. uzlu ***P***(**N**).
    4. Ak uzol ***P***(**N**) nie je koreň stromu {
        1. Zavolaj ***recomputeCenter***(***P***(**N**)).

        }

    }

}

Ak sa ešte vrátime k metóde *slimDown*, tak jej kroky 2.1.1.1.3.2. a 2.2.1.1.3.3. môžeme teraz formálnejšie prepísať takto:

...
2.1.1.1.3.2. Zavolaj ***recomputeCenter***(**L**).
...
2.2.1.1.3.3. Zavolaj ***recomputeCenter***(**U**).
...

Metódu *recomputeCenter* môžeme využiť pri pridávaní bodov do stromu, teda pri vytváraní stromu. Metóda *addToLeaf* (analogicky aj *addToInnerNode*) by s využitím metódy *recomputeCenter* vyzerala nasledovne:

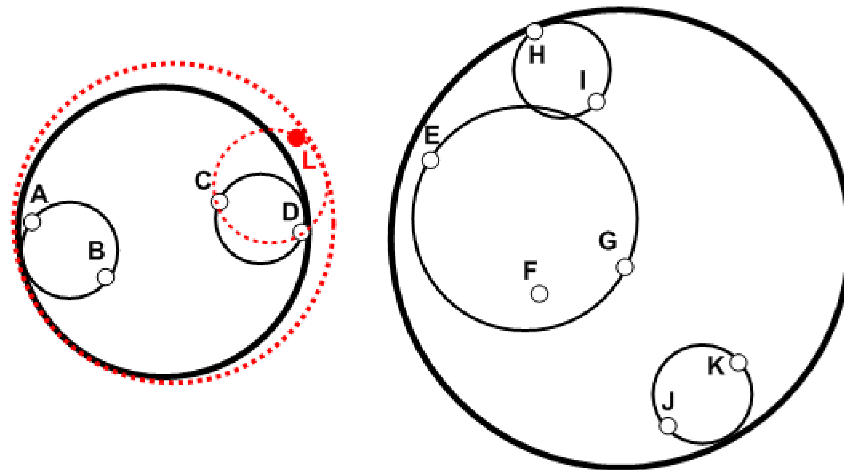
void ***addToLeaf***(O, **L**) {
1. Pridaj do množiny ***D***(**L**) záznam <O, *d*(O, *C*(**L**))>.
2. Ak |***D***(**L**)| nie je väčšia ako povolená kapacita {
    1. Zavolaj ***recomputeCenter***(**L**).

    }
3. Ak je |***D***(**L**)| väčšia ako povolená kapacita {

    ... bez zmeny ...

}

Do metódy *addToLeaf* pribudol nový krok 2., ktorý v prípade, že nedôjde k prekročeniu kapacity uzla, zavolá metódu *recomputeCenter*. Vtedy totiž dochádza k zmene v obsahu uzla a jeho pôvodny stred a polomer už nemusia byť vzhľadom na jeho nový obsah optimálne. Navyše pri hľadaní vhodného listu pre pridanie bodu do stromu metódou *findLeaf* môže dochádzať k zväčšovaniu polomerov uzlov pozdĺž vetvy, ktorou hľadanie prechádza (konkrétne v kroku 2.3.2 pôvodnej metódy *findLeaf*). Tieto zväčšovania polomerov spôsobujú, že polomery a stredy uzlov už nie sú optimálne a bolo by vhodné ich prepočítať. Tento prepočet odštartuje uzol, ktorému pribudne záznam, ale nedôjde ešte k jeho deleniu. Prepočet nových stredov a polomerov postupuje pozdĺž vetvy zdola nahor kým nepríde ku koreňu stromu alebo kým nenastane to, že nový polomer nie je lepší ako pôvodny.



Obr. 4.7 Príklad pridania nového bodu s použitím metódy *recomputeCenter*.

Obrázok 4.7 predstavuje M-strom s virtuálnymi stredmi uzlov a následne jeho zmenu po pridaní nového bodu L. Môžeme pozorovať, že dvom uzlom sa zmenil stred, vďaka čomu je nárast ich polomerov minimálny nutný a nespôsobuje zbytočné zväčšenie uzla do všetkých strán.

5 Praktická časť – návrh programu

Základným cieľom programátorskej časti práce je naprogramovať indexovaciu štruktúru M-strom so všetkými možnosťami, vlastnosťami a algoritmami popísanými v teoretickej časti. Táto kapitola popisuje návrh a mplementáciu M-stromu podľa presne definovaných požiadaviek.

5.1 Požiadavky na aplikáciu

Cieľom je vytvoriť knižnicu, ktorá ponúkne plnú funkcionalitu M-stromu každému programátorovi, ktorý sa rozhodne použiť M-strom vo svojom programe. Druhoradým a nepovinným cieľom, ktorý som si stanovil sám, je vytvoriť grafickú aplikáciu, pomocou ktorej je možné funkcionalitu vytvorenej knižnice demonštrovať názorne a graficky. Návrhu tohto programu sa venovať nebudeme, nakoľko ide o aplikáciu, ktorá má len demonštračný charakter. Používateľskú príručkú k tomuto programu sa nájdete na konci práce v prílohe A.

Požiadavky pre knižnicu poskytujúcu funkcionalitu M-stromu:

1. Použiť moderný rozšírený objektový progamovací jazyk
2. Zachovanie všetkých deklarovaných vlastností M-stromu:
   a. Použiteľnosť M-stromu pre indexovanie bodov ľubovoľného metrického priestoru.
   b. Vyváženosť stromu.
   c. Možnosť dynamického pridávania bodov.
   d. Pre uloženie uzlov stromu použiť pevný disk.
3. Používanie cache pamäte pre naposledy prečítané uzly a s tým spojená minimalizácia počtu prístupov na disk.
4. Možnosť uložiť celý strom do súboru a zachovať tak jeho znovupoužiteľnosť medzi dvoma behmi aplikácie, ktorá ho využíva.
5. M-strom musí byť plne parametrizovateľný a prispôsobiteľný požiadavkám konkrétneho programátora t.j.:
   a. Možnosť zadať veľkosť uzlov v bajtoch pre uloženie uzlov v súbore a umožniť tak efektívne využitie pevného disku.
   b. Možnosť zadanie veľkosti cache pamäte pre naposledy prečítané uzly.
   c. Možnosť nastaviť hraničný pomer rozdelenia bodov do nových uzlov pri preplnení uzla.
   d. Možnosť voľby použitia virtuálnych stredov uzlov.
   e. Možnosť voľby použitia dynamického prepočítavania stredov uzlov.
   f. Možnosť voľby použitia algoritmu Multi way leaf choice pre pridávanie nových bodov.
6. Poskytnúť plnú funkcionalitu pre vyhľadávanie:
   a. Možnosť vyhľadávať pomocou rozsahového dopytu.
   b. Možnosť vyhľadávať pomocou dopytu na *k* najbližších susedov.
   c. Možnosť vyhľadávať pomocou sekvenčného dopytu na najbližších susedov.
   d. Možnosť použitia ľubovoľnej fuzzy funkcie pre sekvenčný dopyt na najbližších susedov.
7. Poskytnúť plnú funkcionalitu pre správu kvality indexácie:
   a. Možnosť použiť algoritmus Multi way leaf choice pre pridávanie novch bodov.
   b. Možnosť zlepšiť kvalitu M-stromu aplikáciou algoritmu Generalized slim down
   c. Možnosť použiť virtuálne stredy uzlov.
   d. Možnosť použiť dynamické prepočítavanie stredov a polomerov uzlov.
   e. Možnosť merať kvalitu faktorom tučnosti stromu.

5.2 Návrh knižnice pre M-strom

Diagram tried na obrázku 5.1 predstavuje návrh tried a rozhraní, ktoré sú zoskupené do jedného balíka `mindex`. Balík `mindex` predstavuje navrhovanú knižnicu poskytujúcu funkcionalitu M-stromu. Tento balík môže byť distribuovaný samostatne alebo spolu s balíkom `implementations` (obr. 5.2), ktorý obsahuje základné a ukážkové implementácie rozhraní `MIndexable` a `FuzzyEvaluator`. Diagramy zobrazujú len stručné základné informácie, aby boli čo najprehľadnejšie.

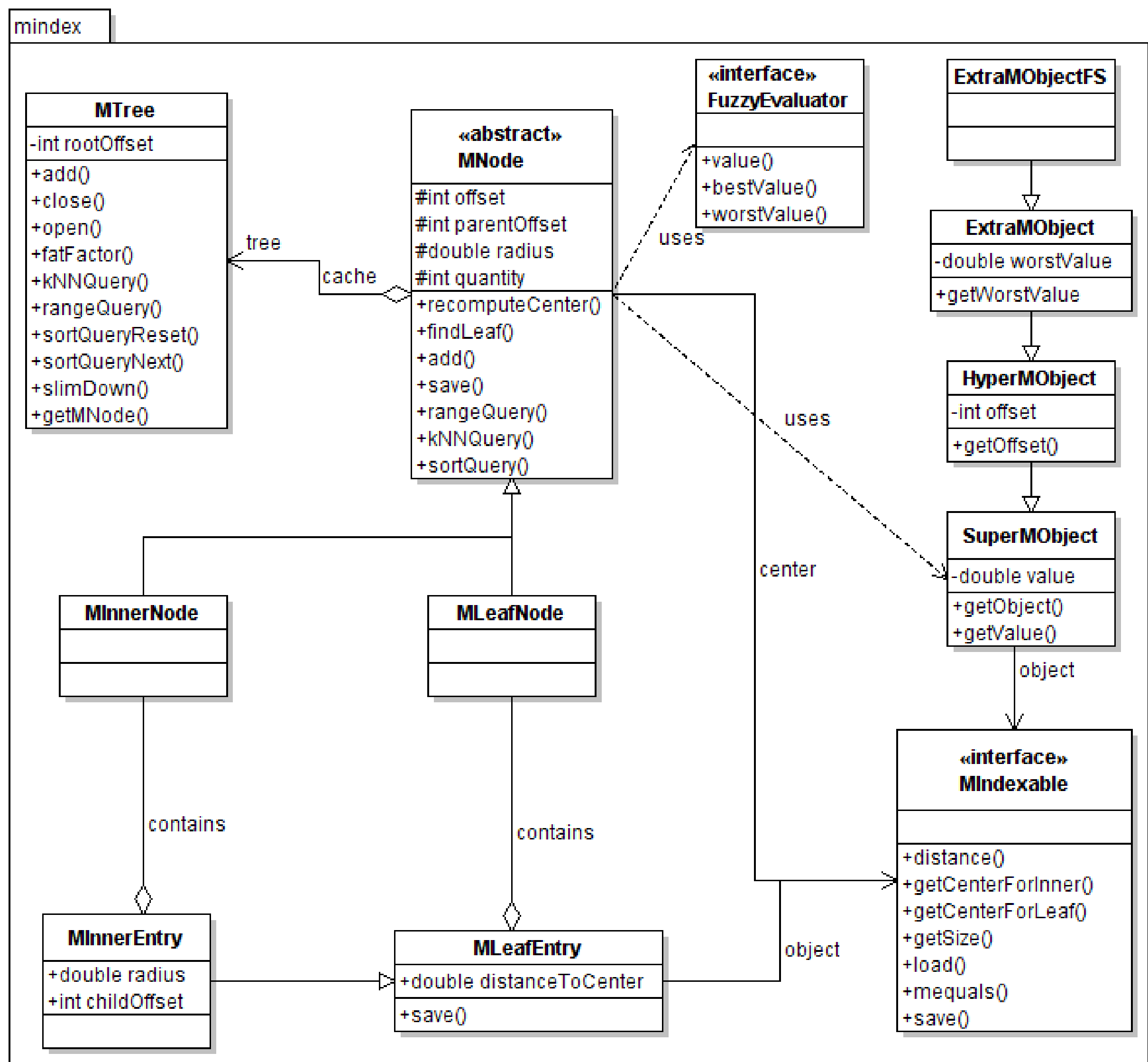
Základnou triedou pre prácu s M-stromom je `MTree`. Objekt tejto triedy predstavuje samotný strom (index), do ktorého je možné predávať body (objekty). Na pridávanie bodov do stromu slúži metóda `add()`. Keďže strom pracuje so súborom, v ktorom má uložené svoje uzly, metódy `open()` a `close()` slúžia na riadenie prístupu k tomuto súboru. Metóda `fatFactor()` vypočíta a vráti faktor tučnosti stromu. Metódy `kNNQuery()` a `rangeQuery()` poskytujú funkcionalitu pre dopt na *k* najbližších susedov a rozsahový dopyt. Metódy `sortQueryReset()` a `sortQueryNext()` zase predstavujú inicializačnú a iteračnú fázu sekvenčného dopytu na najbližších susedov. Pomocou metódy `slimDown()` spustíme nad stromom algoritmus Generalized slim down, ktorý zlepší kvalitu indexu. Metóda `getMNode()` poskytuje centrálny prístup k uzlom stromu. Každý uzol stromu obsahuje referenciu na objekt M-stromu a pomocou jeho metódy `getMNode()` získava objekty svojich poduzlov uložených v súbore rovnako ako aj objekt svojho rodičovského uzla. Táto metóda spravuje cache pamät pre naposledy prečítané uzly a pristupuje k súboru len v prípade, že požadovaný uzol nemá v pamäti.

Uzly stromu sú rozdelené na listy a vnútorné uzly, ktorých triedy zjednocuje ich rodičovská abstraktná trieda `MNode`. Listy a vnútorné uzly sa líšia implementáciou metód zdedených z rodičovskej triedy a obsahom svojich záznamov. Kým listy obsahujú zoznam objektov triedy `MLeafEntry` (t.j. objekt a jeho vzdialenosť od stredu uzla), vnútorné uzly obsahujú zoznam objektov triedy `MInnerEntry`, ktoré obsahujú navyše polomer svojho poduzla a jeho offset v súbore.

Nezávislosť triedy `MTree` od použitého metrického priestoru je zabezpečená použitím rozhania `MIndexable`. Indexované body predstavujú inštancie triedy implementujúcej toto rozhranie. Vďaka tomu si môže ktokoľvek vytvoriť vlastnú implementáciu tohto rozhrania, ktorou definuje vlastný metrický priestor. Takto je možné indexovať ľubovoľné objekty s ľubovoľným spôsobom výpočtu vzdialenosti (ale musí to byť metrika). Metódy `getCenterForInner()` a `getCenterForLeaf()` je potrebné implementovať len v prípade, že chceme použiť virtuálne stredy uzlov. Ich počítanie závisí od metrického priestoru a je teda nutné nechať ich výpočet na impementácii rozhrania `MIndexable`.

Podobná filozofia umožňujúca použiť rôzne implementácie je použitá aj pre fuzzy ohodnocovanie vzdialenosti. To znamená, že spomínané fuzzy funkcie v teoretickej časti (pre algoritmus Fuzzy sort query) je možné implementovať rôzne, ale musí byť splnená špecifikácia daná rozhraním `FuzzyEvaluator`.

Trieda `SuperMObject` predstavuje dvojicu objekt (bod) a hodnota. Jej podtriedy pridávajú ďalšie atribúty ako `offset` a `worstValue` a implementujú špecifický spôsob porovnania pre špecifické usporiadanie. Objekty týchto tried slúžia na reprezentáciu prvkov pracovného a výsledného zoznamu algoritmov kNN query, Sort query a Fuzzy sort query.



Obr. 5.1 Diagram tried pre implementáciu M-stromu

5.3 Implementácia

Pre implementáciu predstaveného návrhu som sa rozhodol použiť jazyk Java. Je to moderný rozšírený objektovo orientovaný programovací jazyk, v Jave disponujem bohatými zručnosťami a skúsenosťami a výsledná knižnica by mala byť jednoducho použiteľná v projekte NAZOU, ktorého nástroje sú vyvýjané práve v Jave. Projekt NAZOU sa zaoberá získavaním, udržovaním a organizovaním znalostí v prostredí heterogénnych informačných zdrojov. Viac o tomto projekte nájdete na jeho domovskej stránke *http://nazou.fiit.stuba.sk*.

Cieľom tohto popisu implementácie nie je vyčerpávajúco zdokumentovať API (application programming interface) vytvorenej knižnice (*mindex.jar*), ale vysvetliť použité spôsoby riešenia. Podrobná dokumentácia verejných metód tried knižnice sa nachádza na priloženom CD. Je to štandardná javovská dokumentácia k API balíka `mindex` vygenerovaná aplikáciou *javadoc*. Je prezentovaná pmocou webových stránok a je určená programátorom, ktorí chcú použiť M-strom vo svojich vlastných javovských programoch.

Najprv si popíšme balík `implementations`, ktorý sa nachádza na CD aj v podobe použiteľnej knižnice *implementations.jar* aj v podobe zdrojových kódov. Diagram tried tohto balíka je znázornený na obrázku 5.2. Triedy tohto balíka predstavujú ukážkové implementácie rozhraní `MIndexable` a `FuzzyEvaluator`.

Trieda `FuzzySet` implementuje rozhranie `FuzzyEvaluator` pomocou charakteristickej funkcie fuzzy množiny. Dokáže reprezentovať ľubovoľnú spojitú po častiach lineárnu charakteristickú funkciu. Reprezentuje teda funkcie, ktorých grafom je lomená čiara.

Trieda `Point2D` predstavuje implementáciu rozhrania `MIndexable` v podobe bodov v rovine koordinovaných dvoma reálnymi súradnicami. Jej podtriedy sa líšia metrikou, ktorú používajú na výpočet vzdialenosti. Trieda `Point2DL1` používa súčtovú metriku označovanú tiež ako manhattan. Výpočet vzdialenosti medzi bodmi A a B pomocou tejto metriky vyzerá takto:

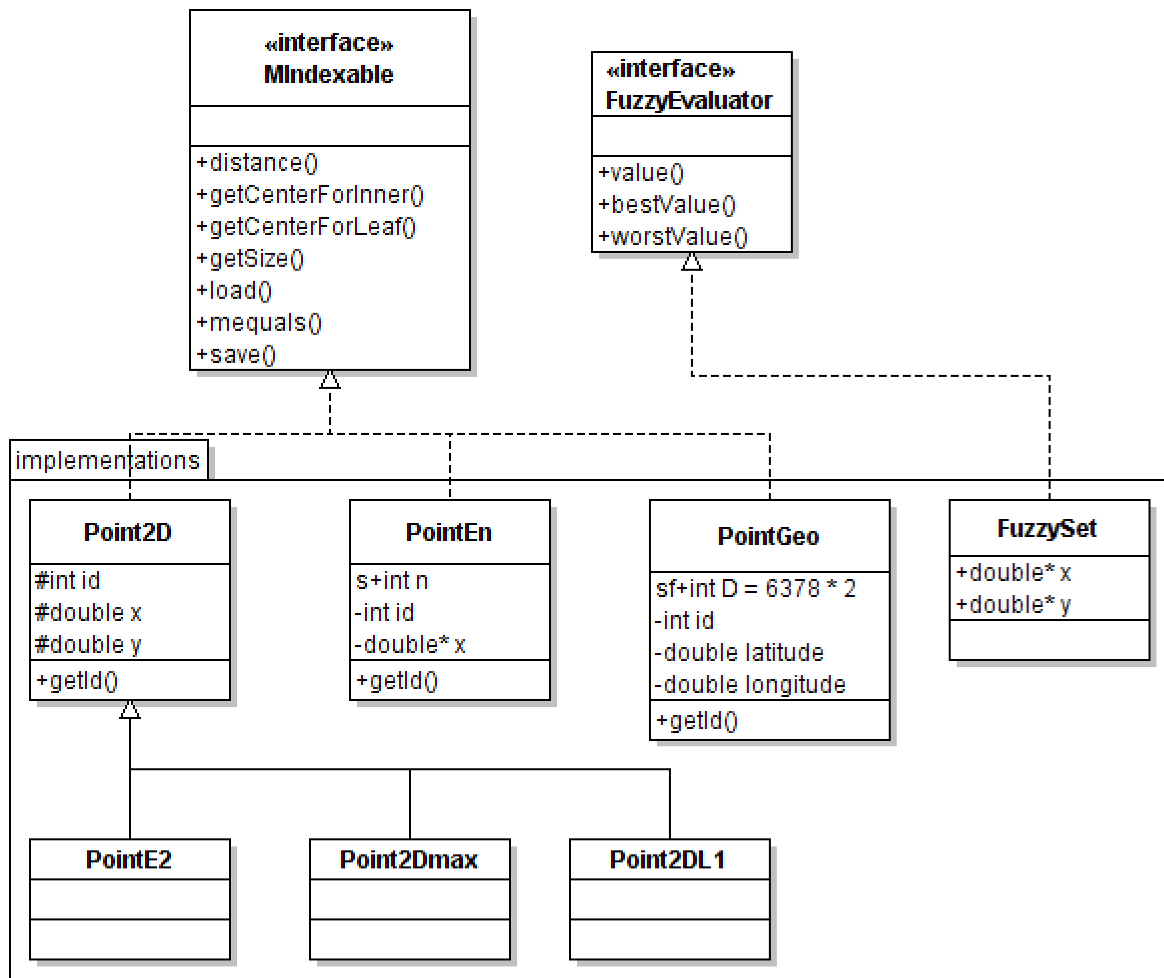
$$d(A, B) = |A.x - B.x| + |A.y - B.y| \quad (5.1)$$

Trieda `PointE2` používa pre výpočet vzdialenosti euklidovskú metriku, výpočet vzdialenosti vyzerá takto:

$$d(A,B) = \sqrt{(A.x - B.x)^2 + (A.y - B.y)^2} \quad (5.2)$$

Trieda `Point2Dmax` používa pre výpočet vzdialenosti tzv. maximálnu metriku a výpočet vzdialenosti vyzerá nasledovne:

$$d(A, B) = max\{|A.x - B.x|, |A.y - B.y|\} \quad (5.3)$$



Obr. 5.2 Diagram tried implementujúcich rozhrania `MIndexable` a `FuzzyEvaluator`

Trieda `PointEn` reprezentuje bod v *n*-rozmernom euklidovskom priestore. Dimenzionalita priestoru sa nastaví statickým parametrom `n`. Bod je potom *n*-rozmerný vektor súradníc. Takto pomocou jednej triedy dokážeme reprezentovať body hocikoľko rozmerného euklidovského priestoru bez nutnosti akéhokoľvek ďalšieho programovania.

Trieda `PointGeo` reprezentuje body na zemskom povrchu, ktoré sú koordinované pomocou sférických súradníc (zemepisná šírka a dĺžka). Vzdialenosť medzi dvoma bodmi je určená dĺžkou najkratšieho oblúka medzi nimi ležiacom na zemskom pvrchu.

Prejdime k opisu implementácie hlavného balíka, ktorým je `mindex`. Jeho hlavná trieda `MTree` poskytuje možnosť uchovávať objekty jedného druhu. Aby bola takáto trieda nezávislá od konkrétnych objektov, obyčajne je to riešené tak, že trieda prijíma objekty triedy `Object`. Ostatné je už na programátorovi, aby zabezpečil to, že do stromu sa budú pridávať objekty rovnakej triedy a že tieto objekty budú poskytovať potrebnú funkcionalitu. Ak toto programátor nezabezpečí, dôsledky sa prejavia počas behu programu. Java 1.5 prišla s novinkou v podobe parametrizovaných tried. Pomocou parametrizácie triedy `MTree` dokážeme už na strane kompilácie zabezpečiť kontrolu, či objekty, ktoré budeme do stromu pridávať, poskytujú požadovanú funkcionalitu. Vhodnou parametrizáciou vynútime, aby sme mohli do stromu pridávať len také objekty, ktoré implementujú požadované rozhranie `MIndexable`. Hlavička triedy `MTree` potom vyzerá teda nasledovne:

```
public class MTree<M extends MIndexable>
```

Pri deklarácii premennej typu MTree<M> musíme parameter M zastúpť triedou, ktorá implementuje rozhranie MIndexable. Ak parameter M nahradíme triedou PointGeo, tak do takto deklarovaného stromu typu MTree<PointGeo> môžeme pridávať len objekty triedy PointGeo a žiadne iné bez ohľadu na to, či implementujú rozhranie MIndexable.

Ďalšou dôležitou požiadavkou je to, aby M-strom používal pre uloženie uzlov súbor a neuchovával všetky dáta v primárnej pamäti počítača. Jedným z parametrov konštruktora triedy MTree je meno súboru, do ktorého sa majú uzly ukladať, a veľkosť jedného uzla v bajtoch. Do zadaného súboru sa uzly ukladajú za sebou tak, že každý uzol zaberá presne toľko bajtov, koľko bolo zadané pri vytváraní objektu triedy MTree (je to hodnota spoločná pre všetky uzly stromu). Pre identifikáciu jednotlivých uzlov stromu je použitý ich offset v súbore. Do súboru sa na vyhradené miesto binárne zapisujú primitívne javovské dátové typy, nie celé objekty. Pri čítaní sa následne vytvárajú nové objekty, ktorých atribúty sú inicializované na hodnoty prečítané zo súboru. Aby bolo možné zapísať do súboru indexované body, rozhranie MIndexable deklaruje metódy load() a save(). Tie sú v triedach implementujúcich rozhranie MIndexable naprogramované špecificky podľa toho, aké atribúty daná trieda obsahuje. Ešte treba dodať, že M-strom potrebuje mať danú fixnú kapacitu uzla, t.j. maximálny možný počet bodov, ktorý je možné vložiť do jedného uzla. Na výpočet kapacity uzla treba vedieť, koľko bajtov zaberá jeden bod. Túto informáciu stromu poskytne trieda indexovaných bodov implementáciou metódy getSize() deklarovanej v rozhraní MIndexable. Napríklad trieda PointE2, ktorá obsahuje len jeden atribút typu int (t.j. 4B pre atribút id) a dva atribúty typu double (t.j. 8B + 8B pre atribúty x, y) implementuje metódu getSize() takto:

```
public int getSize()
{
	return 20;
}
```

Trieda PointEn implementuje metódu getSize() nasledovne:

```
public int getSize()
{
	return 4 + n * 8;
}
```

Listy majú väčšiu kapacitu ako vnútorné uzly, lebo záznamy vo vnútorných uzloch obsahujú oproti listovým záznamom navyše polomer poduzla (typ double) a jeho offset v súbore (typ long). Záznam vnútorného uzla zaberá v súbore o 16B viac ako listový záznam.

Pre efektívne čítanie a zapisovanie do súboru je použitý bufferovaný prístup. Veľkosť buffera je zhodná s veľkosťou uzla a jeden prístup k súboru tak vykoná prečítanie (alebo zápis) jedného uzla zo (do) súboru. Ako buffer je použitý objekt triedy ByteBuffer z balíka java.nio, ktorý je rýchlejší ako bežné javovské polia bajov použité pre bufferovaný prístup iba s využitím balíka java.io. Pre flexibilný pohyb po súbore a čítanie (zapisovanie) potrebného počtu bajtov od požadovaného offsetu je použitá trieda RandomAccessFile z balíka java.io.

Prístup k uzlom uloženým v súbore je centralizovaný a riadený samotnou cache pamäťou pre uzly. To znamená, že keď nejaký uzol potrebuje pristúpť k svojmu poduzlu,

nemôže ho začať čítať z disku sám. Musí zavolať metódu getMNode() hlavného objektu triedy MTree, ktorej predá offset svojho poduzla a táto metóda mu vráti požadovaný uzol priamo ako objekt. Metóda getMNode() po obdržaní požiadavky na uzol s daným offsetom overí, či tento uzol nemá uložený v cache pamäti. Ak áno, jednoducho vráti uzol z pamäti a skončí. Ak nie, prečíta požadovaný uzol z disku, vloží ho do cache pamäti (ak je cache plná, posledný uzol z cache vyhodí vyhodí, ak ten má príznak, že bol zmenený, zapíše ho na disk), vráti na výstup a skončí. Poradie uzlov v cache podľa toho ako boli do cache pridávané zabezpečuje objekt triedy LinkedList (z balíka java.util) obsahujúci offsety zacacheovaných uzlov. Mapovanie reálnych objektov uzlov na ich offsety a rýchle vyhľadávanie uzlov v cache pamäti podľa offsetu zabezpečuje trieda HashMap (z balíka java.util).

Cache pamäť musí mať kapacitu aspoň pre jeden uzol, vtedy sa v pamäti udržiava len aktuálny uzol, s ktorým sa pracuje. Pri požiadavke na ďalší uzol sa aktuálne zacacheovaný uzol nahradí aktuálne prečítaným uzlom. Kapacita cache pamäte sa udáva v počte uzlov. Treba voliť vhodnú kapacitu, aby nedošlo k prekročeniu pamäte pridelenej pre JVM. Javovské objekty zaberajú v operačnej pamäti viac miesta ako presne špecifikovaný počet bajtov, ktorý zaberajú uzly v súbore.

Jednou z požiadaviek bolo, aby sa dal celý strom uložiť na disk a bol znovupoužiteľný pri ďalšom spustení aplikácie bez nutnosti vytvárania nového stromu. Táto požiadavka sa dá elegantne splniť pomocou serializácie objektov, ktorú java poskytuje. Objekty sa takto dajú zapísať napríklad do súboru jedným príkazom. Aby bolo možné takto objekty zapísať na disk (a znovu ich jedným príkazom prečítať), musia implementovať javovské rozhranie Serializable. Implementácia tohto rozhrania je jednoduchá a jedná sa len o formálnu záležitosť, nakoľko toto rozhranie nedeklaruje žiadne mtódy. Keďže objekt triedy MTree obsahuje množstvo ďalších objektov v podobe svojich atribútov, všetky ich triedy musia tiež implementovať rozhranie Serializable, vrátanie triedy zastupujúcej parameter M a reprezentujúcej indexované body. Finálny tvar hlavičky triedy MTree potom vyzerá nasledovane:

```
public class MTree<M extends MIndexable & Serializable>
implements Serializable
```

Pred samotným serializovaním objektu triedy MTree do súboru si treba uvedomiť, že tento objekt pracuje so súborom pre uloženie uzlov. Tento súbor pre uloženie uzlov drží otvorený v exkluzívnom režime pre zápis pomocou objektu triedy RandomAccessFile, ktorý neimplementuje rozhranie Serializable. Preto prv než objekt triedy MTree zapíšeme do súboru, musíme uzavrieť jeho prístup k súboru pre uloženie uzlov. Na tento účel slúži metóda close(), ktorá uzavrie súbor pre uloženie uzlov a príslušný atribút pre prístup k tomu súboru t.j. objekt triedy RandomAccessFile nastaví na hodnotu null. Podobne pri načítani objektu triedy MTree zo súboru alebo pri vytvorení nového objektu triedy MTree treba tento objekt otvoriť pomocou metódy open(). Táto metóda vytvorí nový objekt triedy RandomAccessFile a otvorí súbor pre uloženie uzlov.

Ako posledné v popise implementácie rozoberme spôsob realizácie výsledného a pracovného zoznamu algoritmov kNN query a Sort query. Prvky v týchto zoznamoch musia byť špecificky usporiadané, zoznamy musia byť schopné dynamicky prijímať ďalšie prvky bez znalosti ich počtu vopred, výsledný zoznam musí poskytovať rýchle vyhľadávanie a prvky musí byť možné zo zoznamov odoberať. Všetky tieto požiadavky musia byť splnené čo najefektívnejšie, aby algoritmy bežali čo najrýchlejšie. Pre tieto náročné požiadavky bola vybraná javovská trieda TreeSet z balíka java.util. TreeSet udržiava prvky

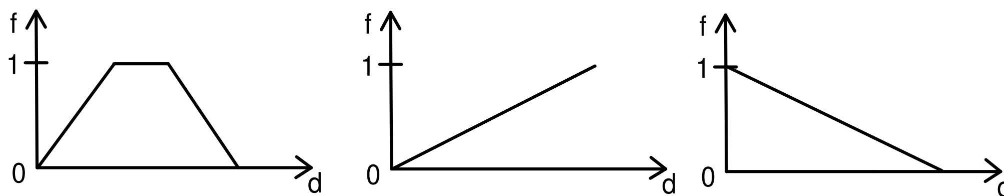
usporiadané a vyhľadávanie, vkladanie a odoberanie prvkov má garantovanú časovú zložitoť $O(\log n)$ vzhľadom na počet prvkov v množine (zozname). Pvrky zoznamov sú realizované triedami `HyperMObject`, `ExtraMObject` a `ExtraMObjectFS`. Inštancie triedy `HyperMObject` predstavujú prvky výsledného zoznamu algoritmu kNN query. Inštancie triedy `ExtraMObject` pedstavujú prvky pracovného zoznamu pre algoritmy kNN query a Sort query. Trieda `ExtraMObjectFS` je použitá pre prvky pracovného zoznamu algoritmu Fuzzy sort query a od triedy `ExtraMObject` sa líši len spôsobom implementácie javovského rozhrania `Comparable` použitom pre usporiadanie. Všetky tieto triedy sú potomkami triedy `SuperMObject` a špecificky implementujú rozhranie `Comparable`, ktoré je triedou `TreeSet` použíté pre usporiadanie prvkov.

6 Testy

Cieľom testov bolo prakticky overiť a vyhodnotiť použiteľnosť navrhnutých a implementovaných algoritmov. Testovanie je rozdelené na dve časti: na testovanie výkonnosti dopytov a na testovanie kvality indexácie M-stromu.

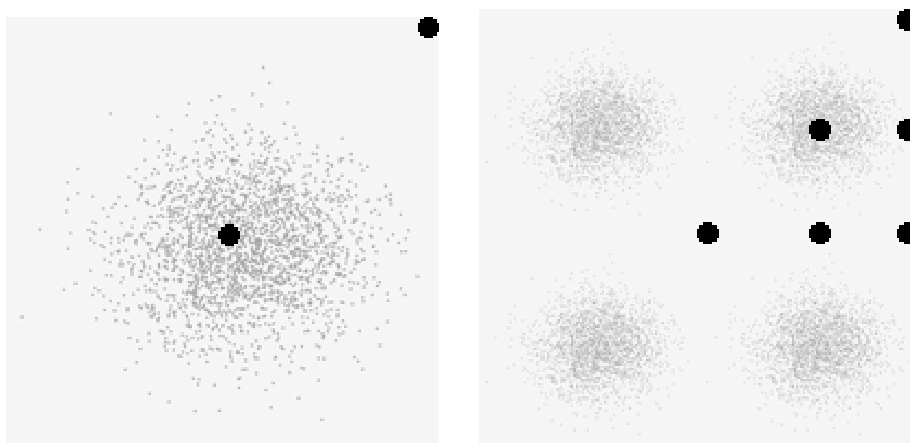
6.1 Testy výkonnosti dopytov

V testoch sme merali čas výpočtu algoritmov pre uvedené dopyty. Porovnávali sme algoritmy kNN query, Sort query a Fuzzy sort query. U Fuzzy sort query sme použili postupne 3 fuzzy funkcie znázornené na obrázku 6.1.



Obr. 6.1 Fuzzy funkcie kopec, rastúca a klesajúca, použité v testoch algoritmu Fuzzy sort query

Testy algoritmov sme vykonali dohromady na siedmich rôznych sadách bodov. Šesť z nich bolo vygenerovaných, jedna sada obsahovala reálne dáta.



Obr. 6.2 Distribúcia bodov pri normálnych rozdeleniach v 2D a kotvové body dopytov

Popíšme najprv vygenerované dáta. Dve sady obsahovali náhodne vygenerované body s rovnomerným rozdelením pravdepodobnosti, pričom jedna obsahovala body z dvojrozmerného euklidovského priestoru, druhá z trojrozmerného euklidovského priestoru. Ďalšie dve sady bodov boli vygenerované s normálnym rozdelením, jedna pre dvojrozmerný priestor, druhá pre trojrozmerný. Posledné dve sady vygenerovaných dát sa tiež líšia len dimenziou. Obsahovali body vygenerované spojením štyroch resp. ôsmich normálnych rozdelení ako znázorňuje obrázok 6.2. Všetky vygenerované sady obsahovali 200 000 bodov.

Reálne dáta tvorilo 18 743 bodov na zemskom povrchu koordinovaných sférickými súradnicami. Všetky tieto body predstavovali byty na území Slovenskej republiky, prevažne vo väčších mestách.

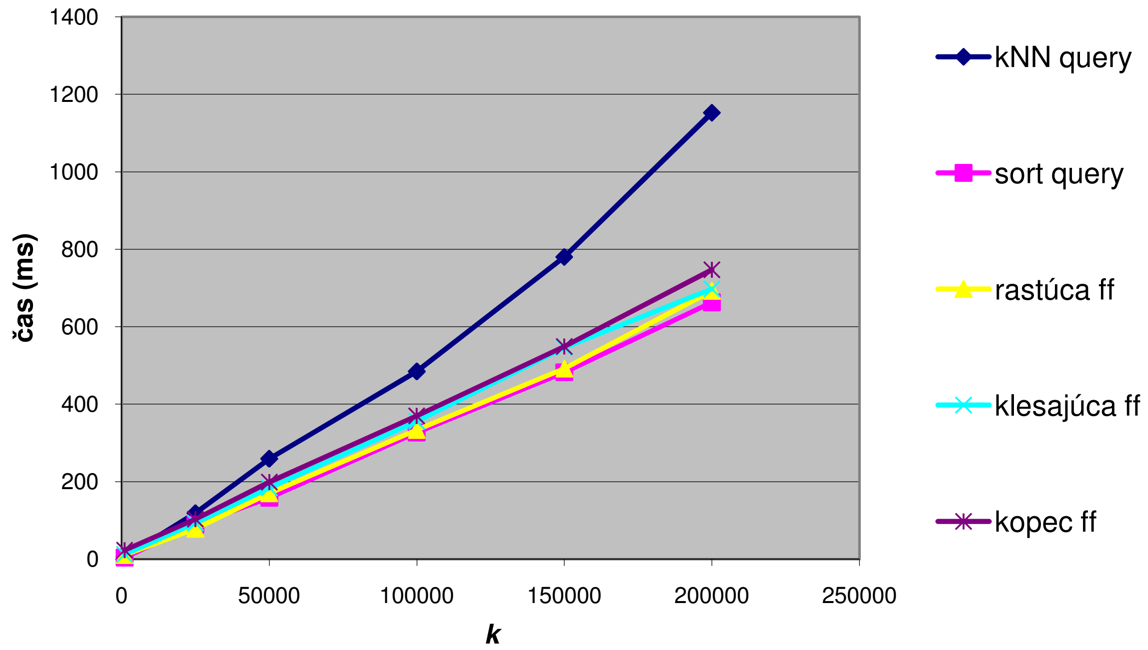
Pre body dvojrozmerného priestoru a body reálnych dát bola použitá kapacita uzla 2048 bytov, pre body trojrozmerného priestoru 2560 čo znamenalo kapacitu uzlov zhruba 60 bodov.

Body boli do stromu pridávané algoritmom Multi way leaf choice popísanom v kapitole 4. Po pridaní všetkých bodov do stromu bol na strom aplikovaný algoritmus Generalized slim down popísaný v tiež v kapitole 4, ktorý zlepší kvalitu indexu.

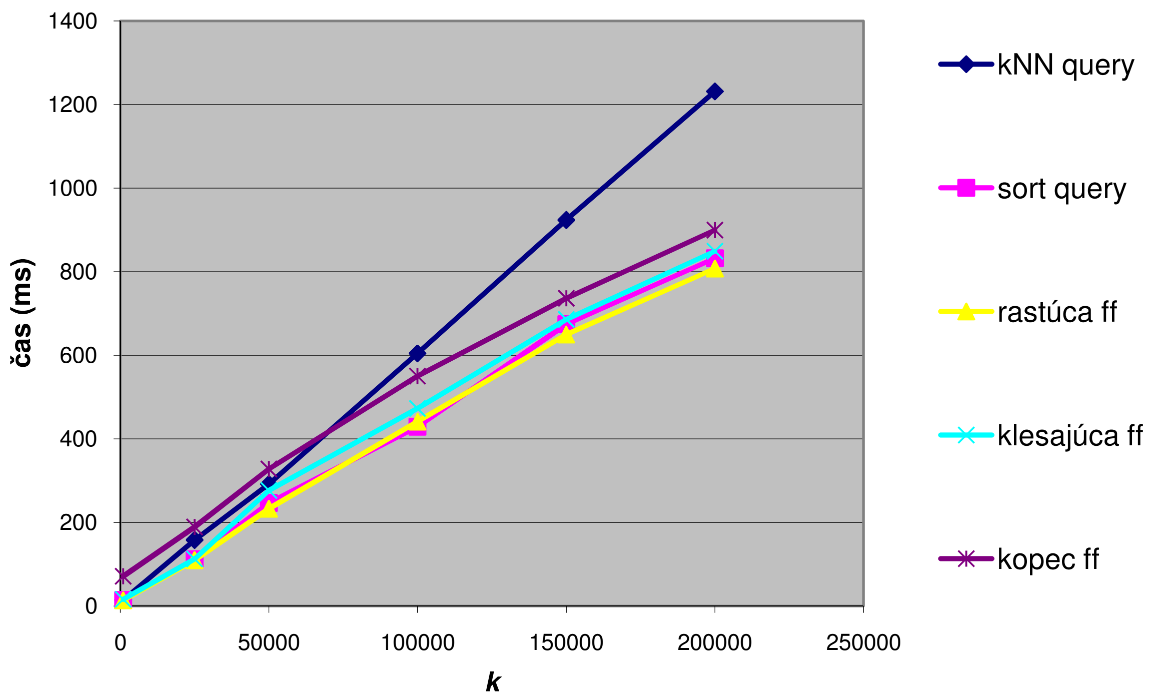
Pre rovnomerne vygenerované dáta bol použitý jeden kotvový bod. Pre dáta s normálnymi rozdeleniami sú použité kotvové body znázornené na obrázku 6.2. U reálnych dát boli použité tri kotvové body a to mestá Bratislava, Banská Bystrica a Košice.

Pre každú sadu dát a zvolený kotvový bod bolo vykonaných päť sérií dopytov. Jedna séria dopytov pre algoritmus kNN query, ďalšia pre Sort query a tri série pre Fuzzy sort query líšiace sa použitou fuzzy funkciou. Použité tri fuzzy funkcie znázorňuje obrázok 6.1. Série dopytov nad vygenerovanými dátami obsahujú 6 dopytov líšiacich sa počtom získaných bodov. Ten bol postupne 1 000, 25 000, 50 000, 100 000, 150 000, 200 000. Série dopytov nad reálnymi dátami obsahovali 4 dopyty postupne pre počty získaných bodov 100, 1 000, 10 000, 18 743.

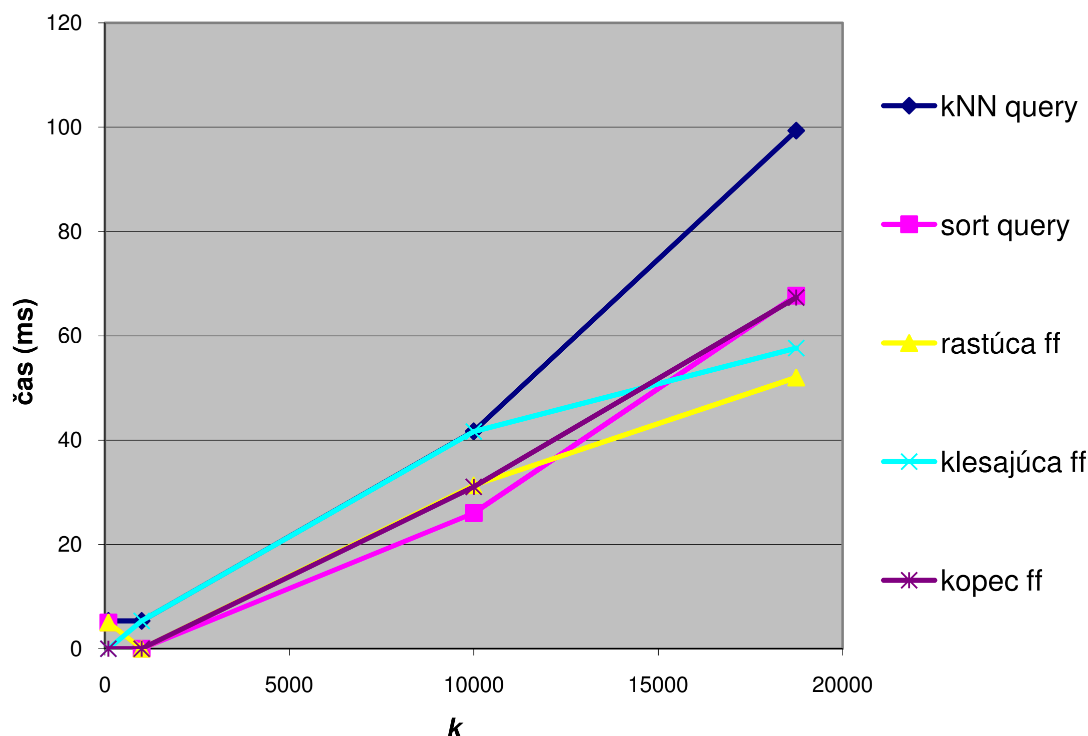
Dohromady bolo vykonaných 540 dopytov nad vygenerovanými dátami a 60 dopytov nad reálnymi dátami. Výsledky z týchto dopytov prinášame spriemerované osobitne pre sady vygenerovaných bodov v dvojrozmernom priestore (obr. 6.3), osobitne pre sady vygenerovaných bodov v trojrozmernom priestore (obr. 6.4) a osobitne pre reálne body na zemskom povrchu (obr. 6.5). Na jednotlivých grafoch sú priemerné časy znázornené osobitne pre jednotlivé série dopytov t.j. pre algoritmy kNN, Sort a Fuzzy sort query, pričom pre algoritmus Fuzzy sort query osobitne pre 3 fuzzy funkcie na obrázku 6.1. Priemerné časy sú pre prezentovanie výsledkov postačujúce, lebo vrámci týchto troch skupín líšiacich sa metrickým priestorom a vrámci jednotlivých sérií dopytov boli rozdiely jednotlivých dopytov minimálne. To znamená že, distribúcia bodov ani poloha kotvového bodu nemá vplyv na výkon dopytov.



Obr. 6.3 Priemerné časy pre dopyty nad sadou vygenerovaných bodov v dvojrozmernom euklidovskom priestore



Obr. 6.4 Priemerné časy pre dopyty nad sadou vygenerovaných bodov v trojrozmernom euklidovskom priestore



Obr. 6.5 Priemerné časy pre dopyty nad sadou reálnych bodov na zemskom povrchu

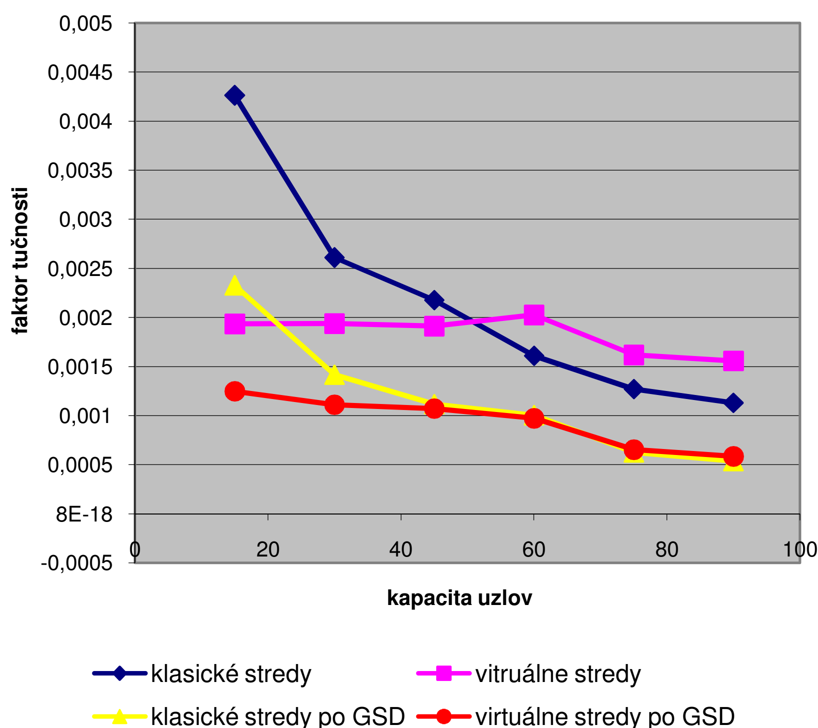
Na grafoch môžeme pozorovať aj to, že na výkon dopytov nemá vplyv ani fuzzy funkcia použitá pre algoritmus fuzzy sort query. Ďalej vidno, že algoritmus Sort query je vo všetkých svojich variantoch rýchlejší ako algoritmus kNN query a pritom Sort query je navyše flexibilnejší! Je to spôsobené tým, že algoritmus kNN query pracuje navyše s výsledným zoznamom veľkosti *k*. Prečítanie uzla v kroku 4.3. algoritmu kNN query často spôsobuje zápis nových prvkov nie len do pracovného zoznamu (ako u algoritmu Sort query), ale aj do výsledného zoznamu (krok 4.4.1.2.2.2. resp. 4.5.1.2.2.2.2. v kNN). Navyše kroky 4.2. a 4.2.1. v kNN query vyžadujú vyhľadávanie a mazanie prvku vo výslednom zozname. Na druhej strane treba povedať, že algoritmus Sort query vkladá do pracovného zoznamu aj body a tento zoznam v priebehu výpočtu neskracuje od konca ako kNN v kroku 4.4.1.2.2.4. resp. 4.5.1.2.2.2.4., čím nastáva to, že pracuje s dlhším zoznamom ako algoritmus kNN. Pokiaľ je tento zoznam vhodne implementovaný (napr. javovskou triedou `TreeSet` použitou v našej implementácii), nespôsobuje jeho väčšia dĺžka žiadny hendikep. Takto si algoritmus Sort query ušetrí prácu s mazaním prvkov z pracovného zoznamu a uchováva v ňom všetky videné uzly, ktoré ešte neprečítal z disku, čím si udržiava dáta pre prípadné usporiadanie všetkých bodov stromu. V prípade algoritmu kNN query sa jeho práca s výsledným zoznamom prejaví na jeho výkone až pri vyšších hodnotách čísla *k* (t.j. počtu požadovaných najbližších bodov), kedy fakt, že vo výslednom zozname sa vystrieda množstvo uzlov a bodov, kým v ňom nakoniec ostane *k*-tica najbližších, spôsobí väčšie časové nároky na celkový výpočet.

Počet prístupov na disk bol u algoritmov kNN query a Sort query za rovnakých podmienok vždy zhodný. U algoritmu Fuzzy sort query s použitím uvedených fuzzy funkcií neboli v počte prístupov na disk pozorované žiadne relevantné rozdiely voči algoritmom kNN resp. Sort query.

6.2 Testy kvality M-stromu

Pre hodnotenie kvality M-stromu sme zaviedli tzv. faktor tučnosti. V testoch je tento ukazovateľ kvality použitý pre vyhodnotenie prínosu dvoch nových vylepšení M-stromu, a to virtuálnych stredov uzlov a ich dynamického prepočítavania. Dáta pre tieto testy tvoria body generované náhodne s rovnomerným rozdelením.

V prvom rade sa zameriame na zlepšenie, ktoré prináša použitie virtuálnych stredov uzlov. Pre porovnanie kvality boli vždy paralelne vytvorené dva M-stromy nad rovnakou množinou bodov, ktoré boli pridávané v rovnakom poradí. Jeden strom používal klasické stredy uzlov, druhý virtuálne. Virtuálne stredy boli počítané algoritmom Subminimal bounding hyperball. Keďže algoritmus Generalized slim down dokáže výrazne zlepšiť kvalitu M-stromu, tak faktor tučnosti bol počítaný a porovnávaný vždy pred aj po aplikácii algoritmu Generalized slim down. V prvom teste bola sledovaná závislosť kvality stromu od kapacity uzlov pre dvojrozmerný euklidovský priestor (štandardnú rovinu). S rastúcou kapacitou uzlov, bol zväčšovaný aj počet bodov pridaných do stromu, aby mali jednotlivé stromy aspoň približne rovnaký počet uzlov (aby výsledky boli porovnateľné).



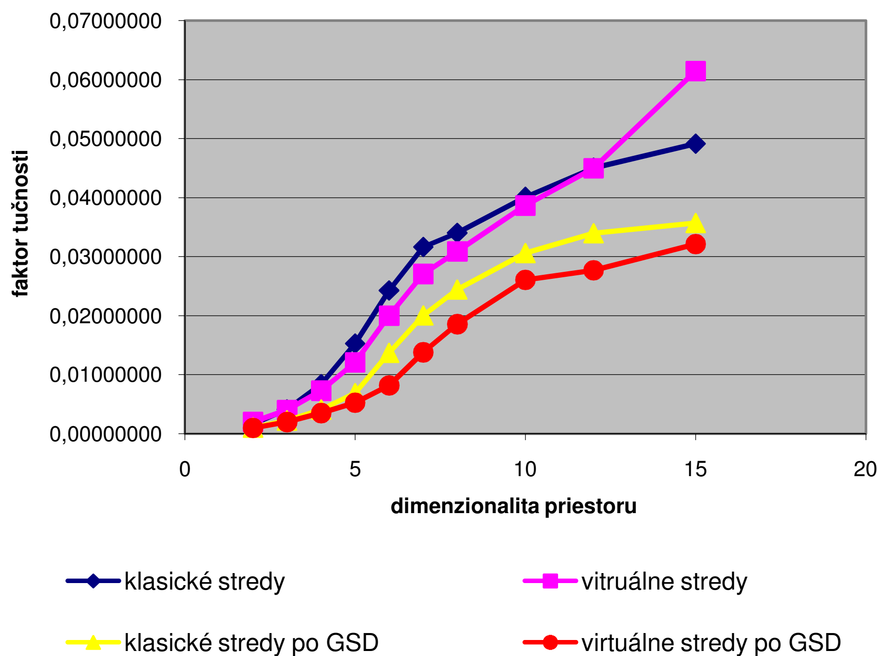
Obr. 6.6 Závislosť faktora tučnosti od kapacity uzlov pre stromy s klasickými a virtuálnymi stredmi, obdiva pred aj po aplikácii algoritmu Generalized slim down (GSD). Dimenzionalita priestoru je 2.

Z grafu na orázku 6.6 je zrejmé, že virtálne stredy prinášajú zlepšenie len pre kapacitu uzlov probližne do 50 bodov. Táto kapacita je len približná, nakoľko listy a vnútorné uzly stromu majú rozdielnu kapacitu. Pre body v rovine obsahujúce len svoje identifikačné čislo a dve súradnice je rozdiel medzi kapacitou listov a vnútorných uzlov stromu pomerne veľký (reálne, kým list má kapacitu 90 bodov, vnútorný uzol má len 60). Čím jeden bod zaberá viac bajtov, tým menší podiel tvoria polomery kruhov a smerníky na poduzly na celkovom objeme dát vo

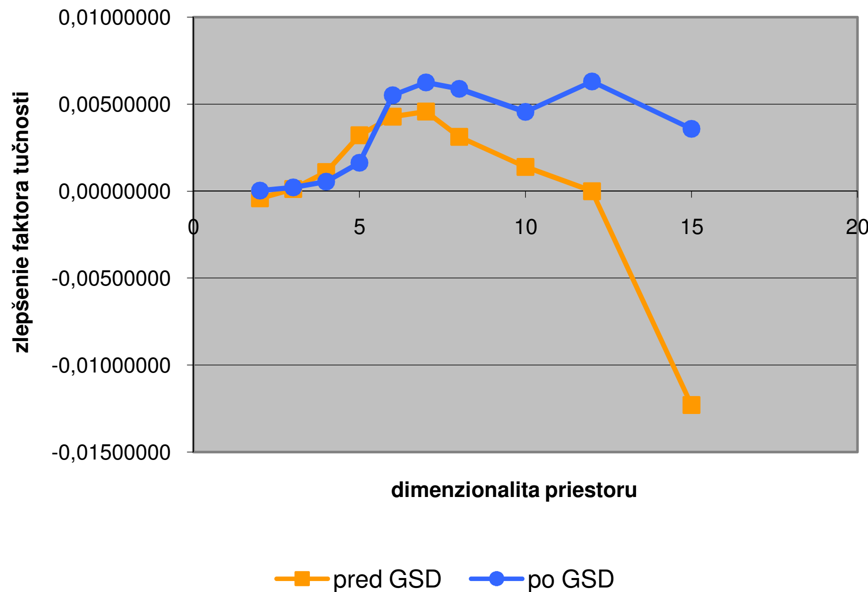
vnútornom uzle a tým menší je rozdiel medzi kapacitou listov a vnútornývh uzlov stromu (reálne pre bod v 20-rozmernom priestore, kým list má kapacitu 90 bodov, vnútorný uzol má 80).

Predchádzajúci test ukázal, že pri kapacite uzlov nad 50 bodov a dvojrozmernom priestore neprinášajú virtuálne stredy žiadne zlepšenie, dokonca naopak. Tento fakt bol motiváciou pre ďalší test, ktorý mal potvrdiť alebo vyvrátiť hypotézu, že pre viacrozmerný priestor prinesú virtuálne stredy zlepšenie aj pre kapacity uzlov nad 50 bodov.

Pre nasledújúci test bola zvolená kapacita uzla fixne na hodnotu približne 60 bodov (konštantný bol tým pádom aj počet pridaných bodov). Dimenzionalita priestoru bola postupne zvyšovaná od hodnoty 2 po hodnotu 15. Faktor tučnosti bol počítaný podobne ako v predchádzajúcom teste pred aj po aplikácii algoritmu Generalized slim down. Výsledky testu znázorňuje priamo obrázok 6.7 a samotné zlepšenie vyjadruje graf na obrázku 6.8.



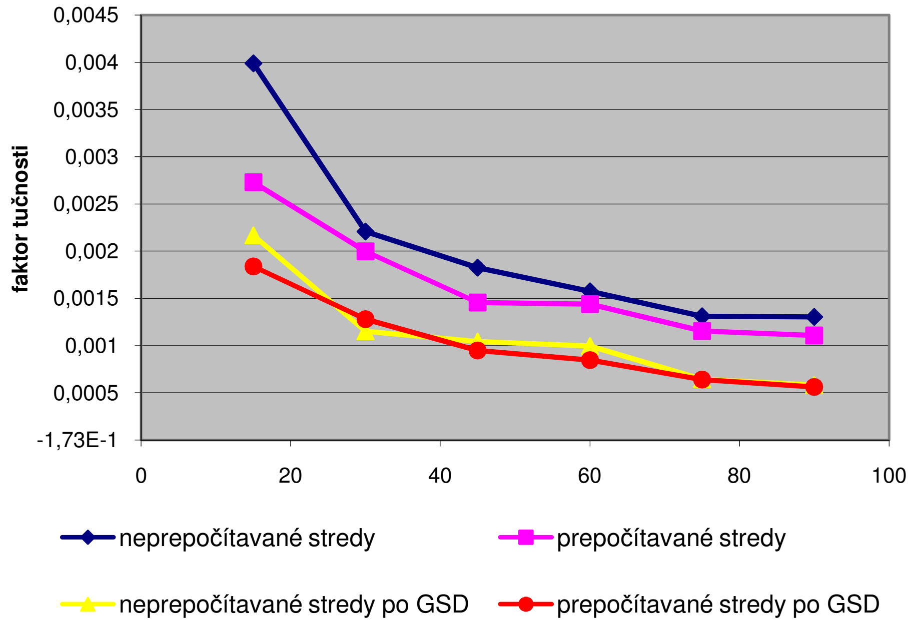
Obr. 6.7 Závislosť faktora tučnosti od dimenzionality priestoru pre stromy s klasickými a virtuálnymi stredmi, obdiva pred aj po aplikácii algoritmu Generalized slim down (GSD). Kapacita uzlov je približne 60 bodov.



Obr. 6.8 Závislosť zlepšenia faktora tučnosti od dimenzionality priestoru pre stromy s klasickými a virtuálnymi stredmi, obdiva pred aj po aplikácii algoritmu Generalized slim down (GSD). Kapacita uzlov je približne 60 bodov.

Graf na obrázku 6.8 vyjadruje rozdiel medzi faktormi tučnosti stromov s klasickými a virtuálnymi stredmi pred a po aplikácii algoritmu Generalized slim down. Jednoducho povedané, oranžová krivka grafu na obrázku 6.8 vyjadruje rozdiel tmavomodrej a fialovej krivky z grafu na obrázku 6.7 (analogicky modrá je rozdiel žltej a červenej). Pre kapacitu uzlov približne 60 bodov prinášajú virtuálne stredy zlepšenie len pre 4 až 12 rozmerné priestory. Pre viac ako 15 rozmerné priestory prinášajú virtuálne stredy počítané algoritmom Subminimal bounding hyperball zhoršenie a tento trend sa už ďalej nemení.

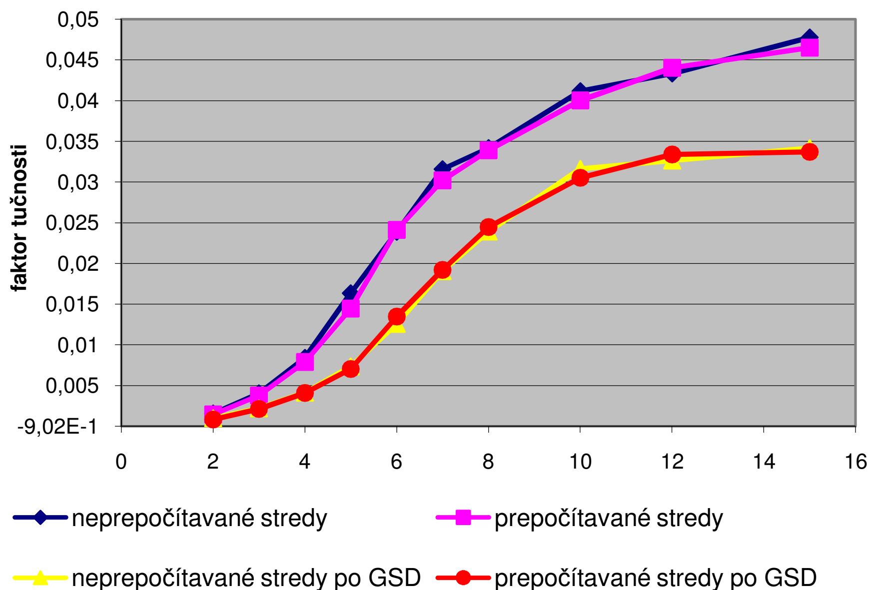
Predmetom druhého testovania bolo ukázať zlepšenie, ktoré prináša dynamické prepočítavanie stredov uzlov. Metodika testovania bola identická s predchádzajúcimi testami, rozdiel bol len v tom, že namiesto druhého stromu s virtuálnymi stredmi, bol vytváraný strom s klasickými stredmi, ktoré boli dynamicky prepočítavané.



Obr. 6.9 Závislosť faktora tučnosti od kapacity uzlov pre stromy s dynamicky prepočítavanými uzlami a bez, obdiva pred aj po aplikácii algoritmu Generalized slim down (GSD). Dimenzionalita priestoru je 2.

Na grafe na obrázku 6.9 vidno, že malé zlepšenie, ktoré dynamické prepočítavanie uzlov prináša dokonca bez ohľadu na kapacitu uzlov, sa stráca po aplikovaní algoritmu Generalized slim down.

Posledný test má odhaliť, ako sa správa kvalita stromu s dynamicky prepočítavanými stredmi uzlov v závislosti od dimenzionality priestoru. Metodika testovania je opäť nezmenená, kapacita uzlov bola fixná, opäť približne 60 bodov.



Obr. 6.10 Závislosť faktora tučnosti od dimenzionality priestoru pre stromy s dynamicky prepočítavanými uzlami a bez, obdiva pred aj po aplikácii algoritmu Generalized slim down (GSD). Kapacita uzlov je približne 60 bodov.

Aj keď z výsledkov na grafe z obrázku 6.10 vidno, že dynamické prepočítavanie stredov neprináša žiadne zlepšenie, reálne hodnoty faktora tučnosti sú v priemere o niečo lepšie. Rozdiel je však z hľadiska výkonu dopytov zanedbateľný.

Testy ukázali, že pokusy o zlepšenie kvality M-stromu neboli také úspešné, aby presadili popísané techniky - virtuálne stredy uzlov a ich dynamické prepočítavanie. Vyplýva z toho snáď len jediný záver a to taký, že zlepšovanie kvality M-stromu sa musí uberať iným smerom, akým sa vybrali uvedené pokusy.

7 Záver

V tejto práci som podrobne rozobral indexáciu bodov v metrickom priestore pomocou M-stromu. V prvej časti práce som sa zaoberal tvorbou M-stromu a dopytmi nad ním. Známe dopyty nad M-stromom sú rozsahový dopyt a dopyt na *k* najbližších susedov. Hlavným prínosom práce je návrh nového algoritmu Sort query a jeho fuzzy verzie pre tzv. sekvenčný dopyt na najbližších susedov. Algoritmus Sort query dokáže plne nahradiť algoritmus kNN query, navyše je rýchlejší, jednoduchší a nevyžaduje znalosť počtu požadovaných bodov vopred.

Druhá časť práce je venovaná kvalite indexácie v M-strome. V tejto časti som najprv rozobral známe prístupy, ktorými sú algoritmuy Generalized slim down a Multi way leaf choice. Potom som predstavil vlastné prístupy, ktorými sú virtuálne stredy uzlov, s nimi spojený algoritmus Subminimal bounding hyperball, a dynamické prepočítavanie uzlov. Tieto riešenia nepriniesli očakávaný efekt, každopádne aspoň bolo experimentálne ukázané, že týmto smerom cesta nevedie.

V praktickej časti som navrhol a naprogramoval knižnicu *mindex.jar* poskytujúcu M-strom so všetkými vlastnosťami a plnou funkcionalitou opísanou v teoretickej časti. Navyše som dobrovoľne naprogramoval aplikáciu MTreeViewer, ktorá slúži na vizualizáciu M-stromu a môže sa stať vynikajúcou pomôckou pri výuke tejto indexovacej štruktúry.

Literatúra

1. Gurský P.: Algoritmy na vyhľadávanie najlepších *k* objektov bez priameho prístupu. Proceedings of Znalosti 2006, pages 95-105, 2006
2. Gurský P., Horváth T., Novotný R., Vaneková V., Vojtáš P.: UPRE: User preference based search system. In IEEE WI 2006
3. Gurský P., Vojtáš P.: Multikriteriálne vyhľadávanie najlepších objektov s podporou viacerých užívateľov. Proceedings of Znalosti 2007.
4. Xin D., Han J, Chang K.: Progressive and Selective Merge: Computing Top-K with Ad-Hoc Ranking Functions. In SIGMOD 2007.
5. Šumák M.: Vyhľadávanie najlepších *k* objektov. Záverečná práca, Ústav informatiky, Prírodovedecká fakulta UPJŠ Košice, 2006

6. Skopal T.: Metric Indexing in Information Retrieval, Ph.D. thesis, VŠB-Technical University of Ostrava, 2004
7. Skopal T., Pokorný J., Krátký M., Snášel V.: Revisiting M-Tree Building Principles. In ADBIS 2003
8. Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In Proceedings of the 23rd Athens Intern. Conf. on VLDB, pages 426–435. Morgan Kaufmann, 1997
9. Chávez E., Navaro G., Baeza-Yates R., Marroquín J. L.: Searching in Metric Spaces. ACM Computing Surveys 33(3):273-321, 2001
10. Welzl E.: Smallest enclosing disks (balls and ellipsoids). Institute of informatics, Berlin University, 1991

Prílohy

Príloha na CD obsahuje okrem iného hlavne javovskú knižnicu (aj s dokumentáciou) obsahujúcu M-strom, vďaka ktorej môže ktokoľvek využívať všetky možnosti M-stromu aj vo svojich vlastných programoch. Pre lepšie a rýchlejšie pochopenie možností, ktoré môj M-strom ponúka odporúčam začať najprv s programom MTreeViewer. Je to spustiteľné jarko (*MTreeViewer.jar*), pomocou ktorého si môžete prakticky a vizuálne vyskúšať M-strom s rôznymi nastaveniami požadovanými v konštruktore triedy `MTree` z knižnice *mindex.jar*. Prílohy A, B a C obsahujú v pradí používateľskú príručku k programu MTreeViewer, vzorové

implementácie rozhraní `FuzzyEvaluator` a `MIndexable`, a vzorový príklad použitia M-stromu z knižnice *mindex.jar*.

Obsah prílohy na CD:

- *MTreeViewer.jar* – spustiteľný program pre názornú ukážku M-stromu a jeho možností a variácií
- *mindex.jar* – knižnica obsahujúca balík `mindex` pre prácu s M-stromom
- *mindex doc* – štandardná javovská webová dokumentácia ku knižnici *mindex.jar*
- *mindex src* – zdrojové kódy tried knižnice *mindex.jar*
- *implementations.jar* – knižnica obsahujúca rozne príklady implementácie rozhraní potrebných pre prácu s M-stromom
- *implementations src* - zdrojové súbory tried knižnice *implementations.jar* implementujúcich príslušné rozhrania
- *sumak.jar* – pomocná knižnica potrebná pre beh programu MTreeViewer

A Používateľská príručka k programu MTreeViewer

Spustenie programu

MTreeViewer je program na vizualizáciu M-stromu. Beží pod Javou 1.6, spustíte ho poklikaním na súbor *MTreeViewer.jar*. V adresári, v ktorom sa nachádza tento program sa musia nachádzať aj potrebné knižnice *mindex.jar*, *implementations.jar* a *sumak.jar*.

Vytvorenie nového M-stromu

Po spustení programu musíte vytvoriť nový strom pomocou menu "Tree/New tree". Tu sa dostávame k nastaveniam, ktoré treba zadať aj v konštruktore triedy `MTree`. Povedzme si podrobne, čo ktorá položka znamená a s ktorým parametrom konštruktora triedy `MTree` súvisí.

- "Tree name" – meno nového stromu. Súvisí s parametrom `fullFileName`. K menu, ktoré zadáte, program pridá príponu ".mnodes" a výsledný reťazec bude tvoriť meno súboru, kde budú fyzicky uložené uzly vášho stromu. Súbor sa vytvorí (ak už existuje, tak sa prepíše) v tom istom adresári, kde je uložený aj súbor *MTreeViewer.jar* (spustený program).
- "Bytes per node" – veľkosť uzla v bajtoch (parameter `buffetSize`). Presne toľko bajtov bude zaberať každý uzol stromu v súbore pre uloženie uzlov. Pre prehľadnosť zobrazovaného stromu v programe MtreeViewer je lepšie voliť menšiu veľkosť uzla, naopak pri použití M-stromu v aplikácii je vhodné voliť väčšiu veľkosť uzla (ale nie veľmi, odporúčaná je kapacita najviac do 100 bodov).
- "Cache size" – veľkosť cache pamäte stromu pre jeho uzly (parameter `cacheCapacity`). Veľkosť tejto pamäte sa udáva v počte uzlov, ktorý si má strom pamätať a táto veľkosť musí byť aspoň 1.
- "Balance factor" – faktor rovnováhy stromu (parameter `balanceFactor`). Pri pridávaní bodov do stromu môže dôjsť k preplneniu nejakého uzla a ten sa musí potom rozdeliť na dva uzly. Balance factor udáva, v akom pomere sa môžu body

preplneného uzla rozdeliť medzi dva nové uzly. Balance faktor musí byť aspoň 2, vtedy sa musia body rozdeliť presne na dve polovice. Väčší balance factor povolí variabilnejšie rozdelenie bodov. Napr. balance factor rovný 3 povolí rozdelenie na 1/3 a 2/3, ak sa tým dosiahnu menšie polomery uzlov a tým pádom menšie prieniky uzlov a lepšia indexácia.

- "Virtual centers" – parameter `virtualCenter`. Je to voľba, či sa má použiť technika počítania virtuálnych stredov uzlov.
- "Recompute new centers" – parameter `recomputeCenter`. Je to voľba, či sa má použiť technika prepočítavania stredov a polomerov uzlov.
- "Multi way leaf choice" – parameter `mwlc`. Je to voľba, či sa má pri pridávaní bodu do stromu použiť algoritmus Multi way leaf choice, alebo sa má použiť jednoduchý algoritmus Single way leaf choice.
- "Metric" – metrika, alebo výber metrického priestoru. Súvisí s parametrom `classM`, ktorým povieme, aké body resp. body akej triedy (resp. body akého metrického priestoru) chceme v M-strome indexovať. Pre druhé dve z ponúkaných volieb nie je implementované počítanie virtuálnych stredov (je to nepovinné), teda voľba metriky sa stane aktívna len po odškrtnutí voľby "Virtual centers".

Pridávanie bodov do stromu

Po vytvorení stromu, je môžné doň pridávať body. Body sa pridávajú buď klikaním na bielu plochu alebo tlačidlom "generate data". Nastavenia týkajúce sa generovania dát je možné zmeniť v menu "Settings/Data generator".

Zobrazovanie uzlov stromu

Jednotlivé uzly stromu sa dajú prezerať klikaním po uzloch virtuálneho modelu stromu v ľavej časti obrazovky. Farby sa cyklicky striedajú, konkrétna farba sa dá vynútiť výberom vo výsuvnom menu (vľavo pod stromom).

Zadávnie dopytov

Nad stromom je možné vyhľadávať pomocou algoritmov Range query, kNN query, Sort query a Fuzzy sort query. Dopyt sa vyberá v ponuke "MTree" hlavného menu. Súradnice kotvového bodu (ani polomer pre Range query) by sa nemal do okienok zadávať ručne, ale klikaním po ploche (okrem prípadu, ak treba zmeniť číslo *k* v dopyte kNN query). Pre spustenie výpočtu dopytu slúži tlačidlo "compute". V prípade Sort query a Fuzzy sort query je možné týmto tlačidlom krokovať a získavať ďalšie body v strome. Tlačidlo "reset" slúži pre začatie nového dopytu, tlačidlo "cancel" pre návrat do režimu pridávania bodov a prezerania stromu. Okienko označené "IOs" udáva počet prečítaných uzlov stromu počas výpočtu aktuálneho dopytu (t.j. počet prístupov na disk).

Pre použitie algoritmu Fuzzy sort query treba zvoliť algoritmus Sort query a v menu "Settings/Fuzzy function" nastaviť požadovanú fuzzy funkciu. Jej použitie sa aktivuje zaškrtnutím "use fuzzy function for sort query". Hodnoty *d* sú vzdialenosti v pixeloch, hodnoty *f*(*d*) sú ich príslušné fuzzy ohodnotenia.

Správa kvality M-stromu

Pre ohodnotenie kvality M-stromu sú k dispozícii dva ukazovatele. Prvým je faktor tučnosti (tlačidlo "fat factor"), druhým je súhrnná veľkosť plôch prienikov uzlov (tlačidlo

"intersections"). Druhý ukazovateľ je možné použiť len pre euklidovskú metriku a platí, že čím menšia celková plocha prienikov, tým lepšia indexácia. Táto hodnota už však nie je taká relatívna ako faktor tučnosti.

Nad stromom je možné spustiť algoritmus Generalized slim down, na čo slúži tlačidlo "slim down". Textový výstup udávajúci počty presunov na jednotlivých úrovniach sa zapisuje do súbru *log.txt*.

Kontrola korektnosti indexácie

Kontrolu korektnosti indexácie spustíte pomocou menu "Tree/Check tree". Táto kontrola overuje korektnosť polomerov všetkých uzlov vzhľadom na obsah ich podstromov, symetriu metriky, správnosť uložených vzdialeností od stredu uzla a pod. Výstup tejto kontroly sa zapisuje do súboru *log.txt*.

Súbor *log.txt*

Všetky chyby a výnimky sa zapisujú do súboru *log.txt*. Ten vznikne po spustení programu v adresári, kde sa nachádza spustený program. Do *log.txt* sa zapisujú aj informácie po stlačení tlačidla "slim down" alebo položky "Check tree" v hlavnom menu.

B Vzorové implementácie rozhraní `MIndexable` a `FuzzyEvaluator`

Nasledujúci kód predstavuje vzorovú implementáciu rozhrania `MIndexable`. Štyri privátne metódy slúžia na výpočet minimálnej obaľujúcej kružnice pre množinu bodov v euklidovskej rovine. Predstavujú reálnu implementáciu metódy *minidisk* z časti 4.6. Dôležité je si všimnúť prítomnosť konštruktora bez parametrov, ktorý neobsahuje žiaden kód. Triedy implementujúce rozhranie `MIndexable` musia obsahovať tento konštruktor, ktorý sa využíva na vytváranie objektov pri čítaní uzla zo súboru. Pomocou tohto konštruktora vznikne nový objekt reprezentujúci bod, ktorý si hodnoty svojich parametrov načíta až v metóde `load()` priamo z buffera na vstupe.

```
import java.io.Serializable;
import mindex.SuperMObject;
import mindex.MIndexable;
import mindex.MInnerEntry;
import mindex.MLeafEntry;

public class PointE2 implements MIndexable, Serializable
{
	private static final long serialVersionUID = 991028438008158775L;

	int	id;
	double x, y;

	public PointE2() {}

	public PointE2(int id, double x, double y)
	{
		this.id = id;
		this.x = x;
		this.y = y;
	}
```

```java
public double distance(MIndexable object)
{
	PointE2 PE2 = (PointE2) object;
	return Math.sqrt((x - PE2.x)*(x - PE2.x) + (y - PE2.y)*(y -
										PE2.y));
}

public boolean mequals(MIndexable p)
{
	PointE2 PE2 = (PointE2) p;

	return (x == PE2.x)&&(y == PE2.y);
}

public void load(ByteBuffer bb)
{
	id = bb.getInt();
	x = bb.getDouble();
	y = bb.getDouble();
}

public void save(ByteBuffer bb)
{
	bb.putInt(id);
	bb.putDouble(x);
	bb.putDouble(y);
}


private PointE2 linesCrossPoint(double ap, double bp, double cp,
							double aq, double bq, double cq)
{
	if (ap * bq == aq * bp) {
		return null;
	}
	double x, y;
	if (ap != 0.0) {
		y = (aq * cp / ap - cq) / (bq - aq * bp / ap);
		x = -bp * y / ap - cp / ap;
		return new PointE2((int) (Math.random() * (-1000000000)),
										x, y);
	}
	else {
		y = (ap * cq / aq - cp) / (bp - ap * bq / aq);
		x = -bq * y / aq - cq / aq;
		return new PointE2((int) (Math.random() * (-1000000000)),
										x, y);
	}
}

private SuperMObject<MIndexable> circle3p(PointE2 a, PointE2 b,
										PointE2 c)
{
	double  ap = a.x - b.x;
	double  bp = a.y - b.y;
	double  cp = -ap * (a.x + b.x) / 2.0 - bp * (a.y + b.y) / 2.0;
	double  aq = a.x - c.x;
	double  bq = a.y - c.y;
	double  cq = -aq * (a.x + c.x) / 2.0 - bq * (a.y + c.y) / 2.0;
	PointE2 s = linesCrossPoint(ap, bp, cp, aq, bq, cq);
	double  dist = s.distance(a);
```

```java
        double  distPom = s.distance(b);
        dist = dist > distPom ? dist : distPom;
        distPom = s.distance(c);
        return new SuperMObject<MIndexable>(s, dist > distPom ? dist :
                                                        distPom);
    }

    private SuperMObject<MIndexable> circle2p(PointE2 a, PointE2 b)
    {
        PointE2 center = new PointE2((int) (Math.random() * (-
            1000000000)), (a.x + b.x) / 2.0, (a.y + b.y) / 2.0);
        double  da = center.distance(a);
        double  db = center.distance(b);
        return new SuperMObject<MIndexable>(center, da > db ? da : db);
    }

    private SuperMObject<MIndexable> minCircle(MLeafEntry[] ps,
                                    MLeafEntry[] rs, int qps, int qrs)
    {
        if (qps == 0) {
            switch (qrs) {
                case 0 : return null;
                case 1 : return new
                    SuperMObject<MIndexable>(rs[0].object, 0.0);
                case 2 : return circle2p((PointE2) rs[0].object,
                                    (PointE2) rs[1].object);
                default: return circle3p((PointE2) rs[0].object,
                (PointE2) rs[1].object, (PointE2) rs[2].object);
            }
        }
        if (qrs == 3) {
            return circle3p((PointE2) rs[0].object, (PointE2)
                            rs[1].object, (PointE2) rs[2].object);
        }
        SuperMObject<MIndexable> circle = minCircle(ps, rs, --qps,
                                                        qrs);
        if ((circle ==
            null)||(ps[qps].object.distance(circle.getObject()) >
                                            circle.getValue())) {
            rs[qrs] = ps[qps];
            circle = minCircle(ps, rs, qps, ++qrs);
        }
        return circle;
    }

    public SuperMObject<MIndexable> getCenterForLeaf(MLeafEntry[]
                                            entries, int quantity)
    {
        return minCircle(entries, new MLeafEntry[3], quantity, 0);
    }

    public SuperMObject<MIndexable> getCenterForInner(MInnerEntry[]
                                            entries, int quantity)
    {
        if (quantity == 1) {
            return new SuperMObject<MIndexable>(entries[0].object,
                                            entries[0].radius);
        }
        double  dist, distC1toC2, maxDistance = Double.MIN_VALUE,
                                    bestDistC1toC2 = Double.MIN_VALUE;
        int     i, j, c1 = 0, c2 = 0;
```

```
		PointE2 center, C1, C2;

		for (i = 0; i < quantity - 1; i++) {
			for (j = i + 1; j < quantity; j++) {
				distC1toC2 =
					entries[i].object.distance(entries[j].object);
				dist = distC1toC2 + entries[i].radius +
								entries[j].radius;
				if (dist > maxDistance) {
					maxDistance = dist;
					bestDistC1toC2 = distC1toC2;
					c1 = i;
					c2 = j;
				}
			}
		}
		C1 = (PointE2) entries[c1].object;
		C2 = (PointE2) entries[c2].object;
		double xx1 = C1.x + entries[c1].radius * (C1.x - C2.x) /
								bestDistC1toC2;
		double yy1 = C1.y + entries[c1].radius * (C1.y - C2.y) /
								bestDistC1toC2;
		double xx2 = C2.x + entries[c2].radius * (C2.x - C1.x) /
								bestDistC1toC2;
		double yy2 = C2.y + entries[c2].radius * (C2.y - C1.y) /
								bestDistC1toC2;
		center = new PointE2((int) (Math.random() * (-1000000000)),
						(xx1 + xx2) / 2.0, (yy1 + yy2) / 2.0);
		dist = center.distance(entries[c1].object) +
								entries[c1].radius;
		maxDistance = center.distance(entries[c2].object) +
								entries[c2].radius;
		maxDistance = dist > maxDistance ? dist : maxDistance;
		for (i = 0; i < quantity; i++) {
			if ((i != c1)&&(i != c2)) {
				dist = center.distance(entries[i].object);
				if (dist + entries[i].radius > maxDistance) {
					PointE2 p = (PointE2) entries[i].object;
					xx1 = p.x + entries[i].radius * (p.x -
								center.x) / dist;
					yy1 = p.y + entries[i].radius * (p.y -
								center.y) / dist;
					xx2 = center.x + maxDistance * (center.x -
									p.x) / dist;
					yy2 = center.y + maxDistance * (center.y -
									p.y) / dist;
					center.x = (xx1 + xx2) / 2.0;
					center.y = (yy1 + yy2) / 2.0;
					maxDistance =
						enter.distance(entries[i].object)
								+ entries[i].radius;
				}
			}
		}
		return new SuperMObject<MIndexable>(center, maxDistance);
	}
}
```

Nasledujúci kód predstavuje vzorovú implementáciu rozhrania `FuzzyEvaluator`, pomocou fuzzy množiny.

```java
import mindex.FuzzyEvaluator;

public class FuzzySet implements FuzzyEvaluator
{
	double[] x, y;

	public FuzzySet(double[] x, double[] y)
	{
		this.x = x;
		this.y = y;
	}

	public double value(double v)
	{
		v = shiftToInterval(v);
		int i = 0;
		while (v > x[i]) i++;
		if (v == x[i]) {
			return y[i];
		}
		else {
			return y[i - 1] + (y[i] - y[i - 1]) * ((v - x[i - 1]) /
										(x[i] - x[i - 1]));
		}
	}

	private double value(double v, int i)
	{
		if (v == x[i]) {
			return y[i];
		}
		else {
			return y[i - 1] + (y[i] - y[i - 1]) * ((v - x[i - 1]) /
										(x[i] - x[i - 1]));
		}
	}

	private double shiftToInterval(double c)
	{
		return c < x[0] ? x[0] : c > x[x.length - 1] ? x[x.length - 1]
													: c;
	}

	public double bestValue(double a, double b)
	{
		a = shiftToInterval(a);
		b = shiftToInterval(b);
		double bestValue, pomBestValue;
		int	i = 0;
		while (a > x[i]) i++;
		bestValue = value(a, i);
		while (x[i] < b) {
			if (y[i] > bestValue) {
				bestValue = y[i];
			}
			i++;
		}
		pomBestValue = value(b, i);
		if (pomBestValue > bestValue) {
			bestValue = pomBestValue;
```

```
        }
        return bestValue;
    }

    public double worstValue(double a, double b)
    {
        a = shiftToInterval(a);
        b = shiftToInterval(b);
        double worstValue, pomWorstValue;
        int    i = 0;
        while (a > x[i]) i++;
        worstValue = value(a, i);
        while (x[i] < b) {
            if (y[i] < worstValue) {
                worstValue = y[i];
            }
            i++;
        }
        pomWorstValue = value(b, i);
        if (pomWorstValue < worstValue) {
            worstValue = pomWorstValue;
        }
        return worstValue;
    }
}
```

C Vzorová ukážka použitia M-stromu

Nasledujúci kód ukazuje použitie M-stromu. Najprv sa vytvorí object triedy `MTree` pre indexovanie bodov triedy `PointE2`. Potom sa strom otvorí a vloží sa doňho 50 000 náhodných bodov. Následne sa vypočíta faktor tučnosti, aplikuje sa algoritmus Slim down a znovu sa vypočíta faktor tučnosti pre porovnanie kvality. Ako posledné sa nad stromom spustí vyhľadávanie najbližších 100 bodov voči zadanému kotvovému bodu algoritmom kNN query a strom sa uzavrie.

```
public static void main(String[] args) throws Exception
{
    int             i;
    double          x, y;
    MTree<PointE2> tree = new MTree<PointE2>(PointE2.class, "nodes1.txt",
                                    2048, 100, 2.5, true, true, true);
    tree.open();
    for (i = 0; i < 50000; i++) {
        x = Math.random() * 1000;
        y = Math.random() * 1000;
            tree.add(new PointE2(i, x, y));
        if ((i + 1) % 10000 == 0) {
            System.out.println(i + 1);
        }
    }

    System.out.println("tree ff: " + tree.fatFactor());
    System.out.println(tree.slimDown());
    System.out.println("tree ff: " + tree.fatFactor());

    ArrayList<SuperMObject<PointE2>> knnqr = tree.kNNQuery(new PointE2(0,
```

```
                                                            500.0, 500.0), 100);

        tree.close();
    }
```