UNIVERZITA PARDUBICE
Fakulta elektrotechniky a informatiky

Speciální vyhledávací algoritmy v datových strukturách uchovávajících geografická a prostorová data

Bc. Radek Novotný

Diplomová práce
2013

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2011/2012**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: .....
Osobní číslo: .....
Studijní program: **N2646 Informační technologie**
Studijní obor: **Informační technologie**
Název tématu: .....
.....
Zadávající katedra: **Katedra informačních technologií**

Zásady pro vypracování:

.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....

Rozsah grafických prací:

Rozsah pracovní zprávy:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

MURPHY, M.,L. Android 2 - Průvodce programováním mobilních aplikací. Brno: Computer Press, 2011. 371 s. EAN 9788025131947.
LEWIS, H. R., DENENBERG, L. Data structures and their algorithms. Berkley, Adison-Wesley, 1997.
Android developers Homepage [online]. 2011 [cit. 2011-10-07]. Dostupné z WWW: http://developer.android.com/.

Vedoucí diplomové práce: .....
Katedra softwarových technologií

Datum zadání diplomové práce: **31. října 2011**
Termín odevzdání diplomové práce: **18. května 2012**

Univerzita Pardubice Česká republika

L.S.

prof. Ing. Simeon Karamazov, Dr.
děkan

prof. Ing. Antonín Kavička, Ph.D.
vedoucí katedry

V Pardubicích dne 15. listopadu 2011

Prohlášení autora

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 6.5. 2013 Bc. Radek Novotný

Poděkování

Chtěl bych touto cestou poděkovat prof. Ing. Ph.D. Antonínu Kavičkovi za vedení diplomové práce a za věcné připomínky při její tvorbě.

Anotace

Diplomová práce se zabývá principem algoritmů hledání nejbližšího souseda nad vybranými datovými strukturami. Ve vzorových aplikacích byly realizovány jejich implementace.

Všechny implementované algoritmy byly otestovány a vzájemně porovnány za účelem jejich doporučení s ohledem na aplikaci.

Klíčová slova

datové struktury, multidimenzionální data, vyhledávací algoritmus, nejbližší soused, nejlepší první, první do hloubky, větve a hranice, aproximace

Title

Special search algorithms of multidimensional data structures.

Annotation

This thesis deals with the principle of nearest neighbor search algorithms over selected data structures. The sample applications were made for their implementation.

All implemented algorithms were tested and compared for their recommendations with regard to the application.

Keywords

data structure, multidimensional data, search algorithm, nearest neighbor, best first, depth first, branch and bound, approximation

Obsah

Slovník

Algorithm	algoritmus
Nearest neighbor	nejbližší soused
Depth-first	první do hloubky
Best-first	nejlepší první
Approximation	aproximace
Farthest	nejvzdálenější
Incremental	inkrementální
Non-incremental	neinkrementální
With duplicate instances of object	s duplicitními instancemi objektu
Branch and bound	větve a hranice
Minimum bounding rectangle	minimální ohraničující obdélník

Seznam zkratek

MBR	Minimum Bounding Rectangle
JDK	Java Development Kid

Seznam obrázků

Seznam tabulek

1 Úvod

Cílem diplomové práce je realizace speciálních vyhledávacích algoritmů v datových strukturách uchovávajících geografická a prostorová data. Pod pojmem speciální vyhledávací algoritmus, se skrývá postup hledání nejbližšího souseda k zadanému objektu. Jedná se zejména o algoritmy:

- Best-first nearest neighbor[1]
- Depth-first *k*-nearest neighbor[2]
- Approximate nearest neighbor[3]
- kombinující výše uvedené postupy

Hledání nejbližšího souseda je realizovatelné pro všechny typy objektů a na různých datových strukturách. Důraz je kladen na aplikaci v hierarchických datových strukturách (tvořené jak bodovými tak i objektovými daty). Pro účely implementace a porovnávání efektivity těchto algoritmů, jsou využity takové datové struktury, které se liší nejenom typem dat, ale také dimenzí dekomponovaného prostoru. Mezi vybrané datové struktury jsou zařazeny:

- k-D strom
- Prioritní vyhledávací strom
- Bodový quad strom
- Regionální quad strom
- R-strom
- Oktalový strom

Pro každou z vybraných struktur, je možné implementovat celou řadu vyhledávacích algoritmů, které se liší svým přístupem a efektivitou. Z tohoto důvodu jsou realizovány jen některé z popsaných v teoretické části práce.

Dalším cílem diplomové práce je provedení testování různých přístupů hledání nejbližších sousedů u výše zmíněných datových struktur, u kterých je kladen důraz na jejich vyváženost pro zajištění relativně srovnatelných podmínek. Jinými slovy je brán v úvahu především statický charakter vstupních dat. Účelem testování, je provedení jejich srovnání a doporučení ohledně jejich nasazení.

Součástí diplomové práce jsou i vzorové aplikace pro účely zobrazení datových struktur, které mohou být tvořeny jak náhodnými, tak i reálnými geografickými daty území České republiky, kde reálná data byla využita zejména pro testování. Tyto aplikace mimo jiné demonstrují funkčnost jednotlivých vyhledávacích algoritmů.

[1] Hledání nejbližšího souseda principem procházení stromové hierarchie do šířky.
[2] Hledání k-nejbližších sousedů principem procházení stromové hierarchie do hloubky.
[3] Hledání nejbližších sousedů pomocí aproximace.

2 Popis jednotlivých vyhledávacích algoritmů

Kapitola teoreticky popisuje vybrané spaciální vyhledávací algoritmy a případně i jejich variace. Zaměřuje se především na nastínění principu řešení hledání nejbližšího souseda u různých algoritmů, doplněného o tzv. pseudokód. Ten slouží pro lepší pochopení a pro implementaci v počítačovém programu. Jsou tu teoreticky rozebrány i takové algoritmy, které nejsou v praktické části realizovány.

2.1 Úvodní parametry

Problém hledání nejbližšího souseda je podobné operaci hledání umístění bodu ve struktuře. V mnoha algoritmech je prvním krokem právě nalezení objektu dotazu *q* a teprve až pak následuje samotné hledání nejbližších sousedů. Často je žádoucí získat nejbližší sousedy v rostoucím pořadí a často ani nevíme, kolik jich bude nutné prozkoumat. Většina klasických metod využívá zejména *depth-first* strategii, alternativně pak strategii *best-first*. Rozdíl mezi těmito dvěma strategiemi je v tom, že v případě *best-first* jsou prvky, z nichž je konstruována datová struktura, zpracovávány v rostoucím pořadí jejich vzdálenosti od dotazovaného objektu *q*. V kontrastu pak u *depth-first* strategie jsou prvky datové struktury zpracovávány v pořadí, jakou určuje operace průchodu do hloubky a porovnávány s dosud nejbližším nalezeným objektem. [1]

2.2 Inkrementální algoritmus využívající principu procházení do šířky[4]

Podstatou tohoto algoritmu je hledání nejbližších sousedů jednoho po druhém podle jejich vzdálenosti od dotazovaného objektu. Hlavním předpokladem, je procházení stromové hierarchie do šířky tzn. metodou *best-first*. Jinými slovy, v každém kroku algoritmu se navštěvují ty prvky, které mají nejmenší vzdálenost ze všech doposud ještě nenavštívených prvků struktury od dotazovaného bodu. K zjišťování aktuálně nejbližšího prvku se využívá globální seznam, kde jsou prvky v pořadí jejich vzdálenosti a platí, že prvek s největší vzdáleností je zařazen na konci tohoto seznamu. Mezi typicky využívanou datovou strukturou, která by odrážela vlastnosti tohoto globálního seznamu je prioritní fronta.

Pseudokód základního inkrementálního algoritmu je uveden níže, kde vstupem do algoritmu je pouze objekt dotazu *q*. Algoritmus začíná inicializováním prioritní fronty kořenovým prvkem struktury. V hlavním cyklu je odebrán prvek s nejmenší vzdáleností. V případě-li se jedná o objekt je považován za nejbližšího souseda a algoritmus končí, v ostatních případech se do fronty zařadí jeho potomci a cyklus se opakuje. [1]

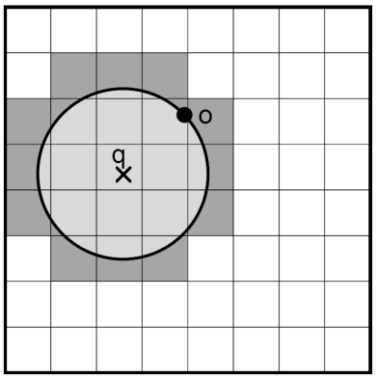
Algoritmus 1

```
fronta = new PrioritniFronta();                    1
     fronta.pridej(koren);                         2
     while !fronta.jePrazdna()                     3
          ep = fronta.odeber();                    4
          if ep.jeObjekt()                         5
```

[4] Incremental best-first.

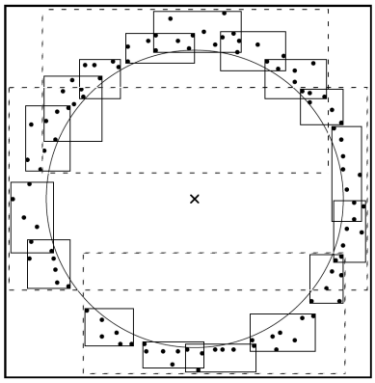
```
            //ep je dalším nejbližším sousedem
        konec if;
        foreach ep.seznamPotomku()
            epnovy = ep.dalsiPotomek();
            fronta.pridej(epnovy);
        konec foreach;
    konec while;
```

Algoritmus je aplikovatelný například i na datovou strukturu R-strom, kde prioritní fronta je inicializována kořenem R-stromu a následně jsou opět v každém cyklu odebírány prvky ze začátku fronty. Hledáme listový uzel obsahující nejbližší objekt. Níže je zobrazen průběh algoritmu pro objekt dotazu *q* (Obrázek 1). Obsahuje kruhový vyhledávající region, který má střed v bodě *q* a startovací poloměr roven nule. Pokaždé, když se region dotkne objektu je nalezen další nejbližší objekt ke *q*. V případě jiného než bodového dotazu, bude mít vyhledávající region složitější tvar. [1]



Obrázek 1: Vyhledávací region, zdroj [1]

Nevýhodou algoritmu je jeho prioritní fronta, která může v nehorším případě obsahovat všechny objekty datové struktury. V případě R-stromu problém příliš rozsáhlé velikosti prioritní fronty vzniká, když všechny listové uzly jsou v menší vzdálenosti od *q*, než všechny jejich objekty (Obrázek 2). [1]



Obrázek 2: Nejhorší případ vyhledávacího regionu, zdroj [1]

2.3 Algoritmus s duplicitními instancemi objektu[5]

Předešlí algoritmus byl aplikovatelný pouze na datové struktury, kde každý objekt je uchováván pouze jednou. Existují však datové struktury, které objekty dělí do několika bloků a ty jsou pak uloženy v různých jejích částech. Mluvíme například o datových strukturách typu R^+-strom, mx quad strom a o členech rodiny pm quad strom atd.

Jedná se o rozšíření inkrementálního algoritmu o dvě základní podmínky. První spočívá v odhalení duplicitních objektů v prioritní frontě. Druhý pak zamezuje vkládání duplicitních objektů, které již byly z fronty zpracovány.

Algoritmus 2

```
fronta = new PrioritniFronta();                                  1
     fronta.pridej(koren);                                       2
     while !fronta.jePrazdna()                                   3
          ep = fronta.odeber();                                  4
          if ep.jeObjekt()                                       5
               while ep==fronta.dejPrvni()                       6
                    fronta.odeberPrvni();                        7
               konec while;                                      8
          else if                                                9
               foreach ep.seznamPotomku()                        10
                epnovy = ep.dalsiPotomek();                      11
                if vzdalenost(q,epnovy)>=vzdalenost(q,ep)        12
                    fronta.pridej(epnovy);                       13
                konec if;                                        14
               konec foreach;                                    15
          konec if;                                              16
     konec while;                                                17
```

Ve vzorovém pseudokódu je patrné, že prvnímu je zabráněno na řádku 12, kde je uvedena podmínka, která zaručuje že žádný objekt, který již byl zpracován, nebude znovu zařazen do prioritní fronty. Ošetření druhé podmínky je v cyklu na řádku 6-8, kde se kontroluje, zda ve frontě není více instancí stejného objektu. [1]

2.4 Neinkrementální algoritmus využívající principu procházení do šířky[6]

Inkrementální algoritmus lze snadno modifikovat, aby nehledal pouze jednoho nejbližšího souseda. Takový algoritmus, který hledá zadaný počet sousedů, se nazývá tzv. *k-nearest neighbor*. Nejjednodušší cestou je právě modifikace inkrementálního algoritmu o tzv. D_k. Jinými slovy o proměnnou, která udržuje vzdálenost od zadaného objektu dotazu *q*, ke *k*-tému nejbližšímu sousedovi. V každém kroku algoritmu tedy hledáme prvek, který má menší vzdálenost než je D_k. Další rozdíl je, že algoritmus obsahuje ještě jednu prioritní

[5] Algorithms with duplicate instances of object.
[6] Non-incremental best-first.

frontu *L*, která obsahuje aktuálně nalezené nejbližší sousedy. Její velikost nesmí přesáhnout *k*, a proto musí být zaručena redukce její velikosti v případě nutnosti.

Obecný algoritmus začíná inicializací D_k na nekonečno. Inicializují se dvě prioritní fronty. *Fronta* obsahuje ještě nenavštívené prvky řazené podle jejich vzdálenosti od *q*. A již zmíněnou frontu *L*. V každém cyklu algoritmu se vzdálenost prvku odebraného z *fronta* porovná se vzdáleností D_k, ale jen v případě jedná-li se o listový prvek. Pokud je menší zařadí se prvek do *L* a jeho potomci se vloží do *fronta*. Velikost D_k se aktualizuje a velikost *L* se případně redukuje. Algoritmus končí, pokud prvek odebraný z *fronta* bude mít větší vzdálenost než je *k*-tý nejbližší soused, tedy D_k. [1]

Algoritmus 3

```
fronta = new PrioritniFronta();                          1
L = new PrioritniFronta();                               2
     fronta.pridej(koren);                               3
     Dk = ∞;                                             4
     while !fronta.jePrazdna()                           5
          ep = fronta.odeber();                          6
          if vzdalenost(q,ep) > Dk                       7
               //k nejbližších sousedů nalezeno          8
               return L;                                 9
          else if ep.jeList()                            10
               foreach ep.seznamObjektu()                11
                    o = ep.dalsiObjekt();                12
                         if vzdalenost(q,o) < Dk         13
                              L.pridej(o);               14
                              //případná aktualizace Dk  15
                         konec if;                       16
               konec foreach;                            17
          else                                           18
               foreach ep.seznamPotomku()                19
                    epnovy = ep.dalsiPotomek();          20
                    fronta.pridej(epnovy);               21
               konec foreach;                            22
          konec if;                                      23
     konec while;                                        24
```

2.5 Nejvzdálenější sousedé[7]

Užitečným rozšířením inkrementálního algoritmu je uzpůsobit ho k hledání nejvzdálenějšího objektu od dotazu *q*. Modifikace lze dosáhnout reverzním řazením objektů v prioritní frontě tzn. objekt s nejmenší vzdáleností od *q* bude na konci. [1]

[7] Farthest neighbors.

2.6 Algoritmus *k*-nejbližších sousedů využívající princip procházení do hloubky[8]

Algoritmus jehož hlavní podstatou je zpracovávání prvků v pořadí, která využívá principu prohledávání hierarchické struktury do hloubky.

Algoritmus je popsán na ukázkovém pseudokódu níže, kde vstup do algoritmu je rozšířen o číslo *k*, které udává kolik nejbližších sousedů hledáme. Algoritmus je tvořen prioritní frontou *L*, která obsahuje *k*-nejbližších sousedů, které jsou uspořádány dle rostoucí vzdálenosti od *q*. Prioritní fronta znovu redukuje svoji velikost, která musí být vždy maximálně *k*. K prořezávání slouží opět proměnná *Dk*. Při nalezení prvních *k*-nejbližších sousedů, se aktualizuje proměnná *Dk*. Aktivní seznam tentokrát obsahuje prvky v pořadí, aby odpovídaly procházení stromu do hloubky. [1]

Algoritmus 4

```
L = new PrioritniFronta();                                    1
Dk = ∞;                                                       2
dfTrav(q,k,koren,L,Dk); //volání metody                       3
return L;                                                     4
                                                              5
dfTrav(↓q, ↓k, ↓ep, ↓L, ↓Dk) //metoda                         6
     A = new AktivniSeznam();                                 7
     if ep.jeList()                                           8
          foreach ep.seznamObjektu()                          9
               o = ep.dalsiObjekt();                          10
               if vzdalenost(q,o) < Dk                        11
                    L.pridej(o);                              12
                    //případná aktualizace Dk                 13
               konec if;                                      14
          konec foreach;                                      15
     else                                                     16
          foreach A.seznamPrvku()                             17
               ep = A.dalsiPrvek();                           18
               //rekurzivní volání metody                     19
               dfTrav(q,k,ep,L,Dk);                           20
          konec foreach;                                      21
     konec if;                                                22
konec dfTrav;                                                 23
```

Jak je patrné základní algoritmus prochází všechny prvky hierarchické struktury. Lepší výkon přinese, když nebudeme navštěvovat všechny prvky, a to přidáním jednoduché podmínky, která daný prvek přeskočí v případě má-li větší vzdálenost od *q* než *Dk*, tzn. nezařadí-ho do aktivního seznamu. Následující modifikace se týká řádků 17 až 21.

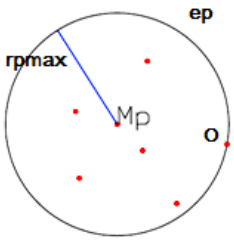
```
          foreach A.seznamPrvku()                             1
               ep = A.dalsiPrvek();                           2
               if vzdalenost(q,ep) > Dk                       3
```

[8] Depth-first *k* – nearest neighbor.

```
            //přeskočení daného prvku                4
        else                                           5
            dfTrav(q,k,ep,L,Dk);                       6
        konec if;                                      7
    konec foreach;                                     8
```

2.7 Algoritmus Fukugana a Narendra využívající princip procházení do hloubky[9]

Algoritmus založený na použití přirovnávacích úvah a požadavků, s cílem zvýšit výkonnost obecného *depth first k-nearest neighbor*. Nevyžaduje, aby sousedé byly zpracovávány v rostoucím pořadí jejich vzdálenosti od dotazovaného objektu *q*. Vylepšení klasického algoritmu dosahuje díky přiřazení několika informací ke každému prvku hierarchie. Zejména se jedná o tzv. střední bod označován jako M_p a vzdálenost $r_{p,\,max}$. Tato vzdálenost je rovna vzdálenosti ze středu uzlu e_p k nejvzdálenějšímu objektu v tomtéž uzlu (Obrázek 3). [1]

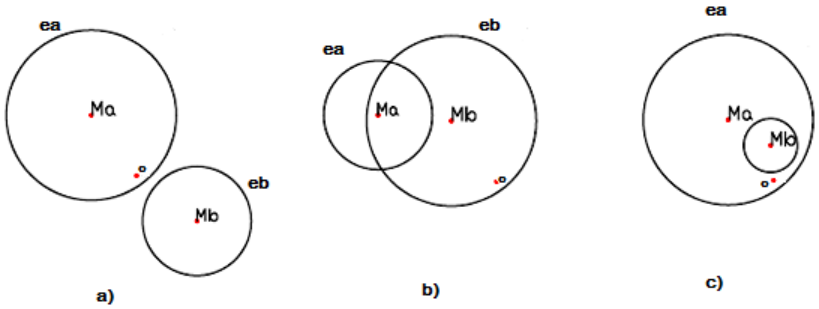


Obrázek 3: Ukázka vzdálenosti r_{pmax}, zdroj [1]

Fukugana a Narendra definovali vzdálenost $d(o,e_p)$ mezi objektem *o* a nelistovým uzlem e_p, který je representován svým středovým bodem M_p a maximálním poloměrem $r_{p,\,max}$. V praxi se pro tuto vzdálenost používá název $MEANORPIVOTDIST(o,e_p)$ (vzdálenost ke středu). Pokud objekt *o* nebo *q*, případně oba, se nacházejí v uzlu e_p platí vztah:

$$MEANORPIVOTDIST(o,e_p) = d(o,e_p) = d(o,M_p)$$

Na obrázku níže je vidět, že objekt *o*, nemusí nutně být v uzlu, jehož středový bod je bodu *o* nejblíže (Obrázek 4).



Obrázek 4: Vztah mezi středovým bodem M a objektem o, zdroj [1]

[9] Algoritmus Fukugana and Narendra.

Výše uvedené informace jsou využívány v pěti prořezávacích pravidlech. Hlavním cílem těchto pravidel je vyhnout se posuzováním nelistových uzlů, v případě kdy všechny jejich potencionální objekty jsou dále od dotazu q než je D_k. Jinými slovy prořezávání těchto uzlů znamená, že se vyhneme zbytečným výpočtům mezi nimi a dotazovaným objektem q, když víme, že nebudou zařazeny ve výsledné množině k-nejbližších sousedů.

Prořezávací pravidla jsou zobrazeny v pseudokódu níže. Od předchozího algoritmu se liší především používáním těchto pravidel a také v požadavku na aktivní seznam. V tomto případě jsou prvky v aktivním seznamu v rostoucím pořadí podle jejich *MEANORPIVOTDIST* vzdálenosti od objektu q. [1]

Algoritmus 5

```
dfTrav(↓q, ↓k, ↓ep, ↓L, ↓Dk) //metoda                                  1
     A = new AktivniSeznam();                                          2
     if ep.jeList()                                                    3
          foreach ep.seznamObjektu()                                   4
               o = ep.dalsiObjekt();                                   5
               if vzdalenost(q,M) – vzdalenost(o,M) > Dk //#2          6
                    //ignorování objektu o                             7
               else if vzdalenost(o,M) - vzdalenost(q,M) > Dk //#4     8
                    //ignorování objektu o                             9
               else                                                    10
                    if vzdalenost(q,o) < Dk                            11
                         L.pridej(o);                                  12
                         //případná aktualizace Dk                     13
                    konec if;                                          14
               konec if;                                               15
          konec foreach;                                               16
      else                                                             17
          if k == 1                                                    18
               foreach A.seznamPrvku() //#5                            19
                    ep = A.dalsiPrvek();                               20
                    if vzdalenost(q,Mp) > D1                           21
                         //ignorování ep                               22
                    else if vzdalenost(q,Mp) + rpmin < D1              23
                         D1 = vzdalenost(q,Mp) + rpmin;                24
                    konec if;                                          25
               konec foreach;                                          26
          else if                                                      27
               foreach A.seznamPrvku()                                 28
                    if vzdalenost(q,Mp)-rpmax > Dk //#1                29
                         //ignorování ep                               30
                    else if rpmin - vzdalenost(q,Mp) > Dk //#3          31
                         //ignorování ep                               32
                    else                                               33
                         //rekurzivní volání metody                    34
                         dfTrav(q,k,ep,L,Dk);                          35
                    konec if;                                          36
               konec foreach;                                          37
          konec if;                                                    38
       konec if;                                                       39
```

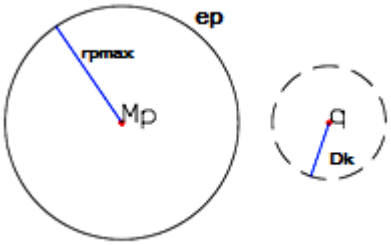
```
konec dfTrav;
```


2.7.1 Prořezávací pravidlo I.

Uzel e_p z aktivního seznamu s M_p a vzdáleností $r_{p,\,max}$ od M_p do nejvzdálenějšího objektu v e_p nemůže obsahovat k-tého nejbližšího souseda dotazu q právě když:

$$D_k + r_{p,max} < d(q, M_p)$$

Počítá minimální vzdálenost od objektu dotazu q k objektu v e_p (označovanou jako *MINDIST*) a stanovuje, jestli je kratší než D_k, tzv. vzdálenost od objektu q k nejvzdálenějšímu sousedovi z k aktuálně nalezených sousedů (Obrázek 5). [1]



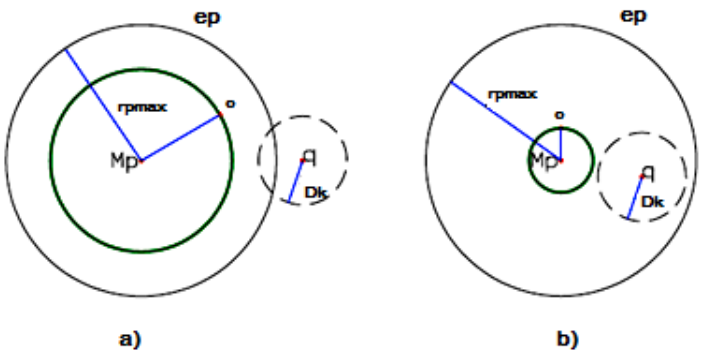
Obrázek 5: Prořezávací pravidlo I, zdroj [1]

2.7.2 Prořezávací pravidlo II.

Objekt o v uzlu e_p na nejhlubší úrovni hierarchie hledání s M_p a vzdáleností $r_{p,\,max}$ od M_p do nejvzdálenějšího objektu v e_p nemůže být k-tý nejbližší soused dotazu q právě když:

$$D_k + d(o, M_p) < d(q, M_p)$$

Toto pravidlo je související k pravidlu prvnímu v tom, že odpovídá situaci, kdy některé prvky v e_p jsou dostatečně daleko od M_p, aby byly kandidáty pro množinu výsledných k-nejbližších sousedů dotazu q. Může existovat objekt o, který je dostatečně blízko k M_p, abychom ho mohli vyřadit z výsledné množiny k-nejbližších sousedů (Obrázek 6).



Obrázek 6: Prořezávací pravidlo II, zdroj [1]

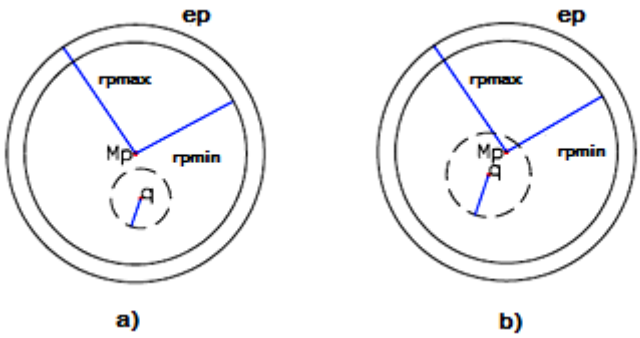
Kamgar-Parsi a Kanal zlepšili algoritmus Fukugana a Narendra tím, že přispěli dalšími dvěma pravidly. Pravidlo třetí je aplikovatelné na prvky v aktivním seznamu, zatímco pravidlo čtvrté je aplikovatelné na individuální objekty. Jejich rozdíl oproti předchozím dvou je, že každý uzel mimo vlastnosti M_p a $r_{p,\,max}$ je opatřen o další vlastnost tzv. $r_{p,\,min}$. Analogicky k $r_{p,\,max}$ se jedná o vzdálenost M_p k nejbližšímu objektu v e_p. [1]

2.7.3 Prořezávací pravidlo III.

Uzel e_p z aktivního seznamu s M_p a vzdáleností $r_{p,\,min}$ od M_p do nejbližšího objektu v e_p nemůže obsahovat k-tého nejbližšího souseda dotazu q právě když:

$$D_k + d(q, M_p) < r_{p,min}$$

Ilustrace pravidla níže ukazuje, že toto pravidlo počítá minimální vzdálenost od bodu dotazu q k objektu v e_p a stanovuje, jestli je kratší než D_k. Oproti prvnímu pravidlu se liší, protože v případě prvního pravidla, uzel e_p má tvar vícerozměrné kulové vrstvy oproti tomu pravidlo třetí má smysl jen tehdy, pokud q je uvnitř vnitřní kulové vrstvy odpovídajícího e_p (Obrázek 7). [1]



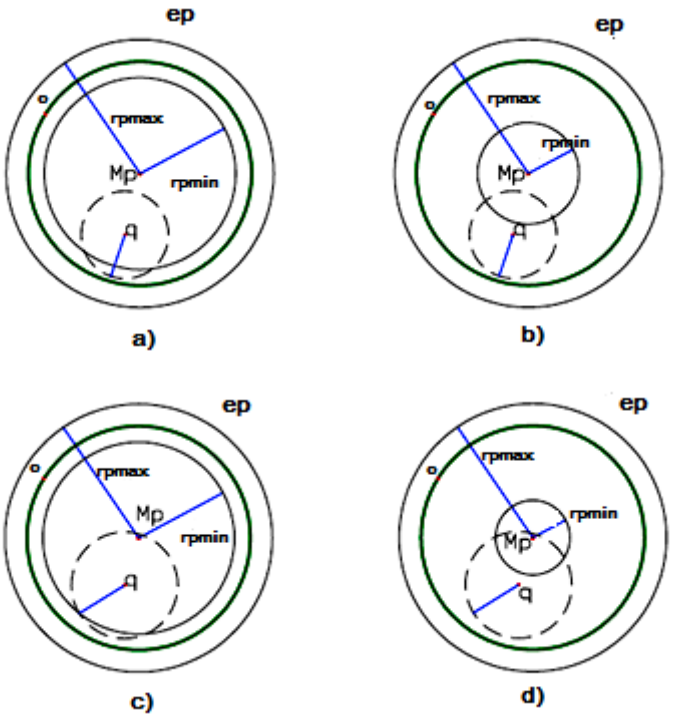
Obrázek 7: Prořezávací pravidlo III, zdroj [1]

2.7.4 Prořezávací pravidlo IV.

Objekt o v uzlu e_p na nejhlubší úrovni hierarchie hledání s průměrným M_p a vzdáleností $r_{p,\,min}$ od M_p do nejvzdálenějšího objektu v e_p nemůže být k-tý nejbližší soused dotazu q právě když:

$$D_k + d(q, M_p) < d(o, M_p)$$

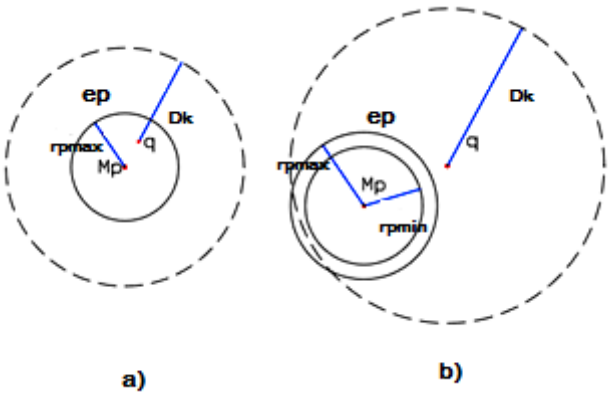
Jinými slovy pravidlo má opodstatnění jen když objekt dotazu q je blíže, než-li objekt o. Podmínky aplikace pravidla jsou, aby objekt q byl uvnitř uzlu e_p nebo uvnitř vnitřní kruhové vrstvy uzlu e_p (Obrázek 8). [1]



Obrázek 8: Prořezávací pravidlo IV, zdroj [1]

2.7.5 Prořezávací pravidlo V.

Pro k = 1, Fukugana a Narendra navrhli ještě jedno prořezávací pravidlo, které je charakterizováno menší optimalizací. Používá se při konstrukci aktivního seznamu A obsahující potomky uzlu e_p. Jeho podstatou je výpočet maximální možné vzdálenosti od q k jeho nejbližšímu sousedovi i v e_p (označovanou jako *MAXNEARESTDIST*). Tento prvek i se dále kontroluje, zda může případně sloužit jako nejbližší soused q, protože je blíže než aktuální soused x. Ten jsme zjistili, výpočtem vzdálenosti od q k nejvzdálenějšímu možnému nejbližšímu sousedovi v e_p, namísto výpočtu vzdálenosti k nejvzdálenějšímu možnému objektu v e_p (Obrázek 9).



Obrázek 9: Prořezávací pravidlo V, zdroj [1]

Pro každý prvek v e_p musí platit, že především jejich vzdálenost od q nepřekročí D_1, které je dáno vzdáleností $d(q, M_p) + r_{p,\ min}$. Pokud se hodnota překročí prvek není dále zpracováván. V případě kdy se tato hodnota nepřekročí, dochází k pře-inicializaci D_1.

V případě, kdy je minimální ohraničující objekt kruhový, pak aplikace prořezávacích pravidel 1-4 je úměrná principu popsaným výše. V případě kdy minimální ohraničující objekty jsou tvořeny obdélníky, pak pravidla 2 a 4 jsou aplikovány stejně, jen s dobře zvoleným středem ohraničujícího objektu, zatímco pravidla 1 a 3 musí být přeformulovány z hlediska minimální vzdálenosti dotazovaného objektu a minimálního ohraničujícího obdélníku. [1]

2.8 Speciální metrické vzdálenosti

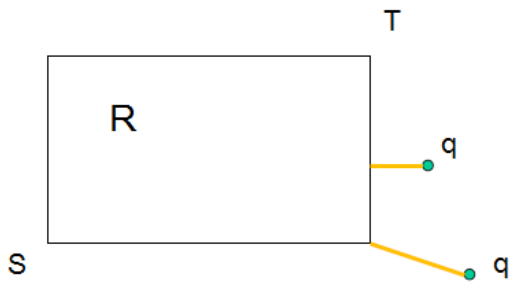
Kapitola je zaměřena na přiblížení principů výpočtu vzdálenosti mezi dvěma rozdílnými typy objektů. Znalost těchto vzdáleností je nutná především v datové struktuře R-strom, která zapouzdřuje své objekty do minimálních obdélníků, a proto je nutné znát vzdálenost mezi dotazovaným bodem a zapouzdřujícím objektem. Na základě výsledku pak můžeme rozhodnout, zda danou větev zařadíme do seznamu kandidátů nejbližších sousedů či nikoli.

2.8.1 Mindist a maxdist

Jedná se o minimální vzdálenost mezi bodem q a obdélníkem R, případně jiným objektem (Obrázek 10). Máme-li obdélník R, který je zadaný dvěma body na diagonále s a t, přičemž pro všechny dimenze platí $k = 1..n$, $s_k < t_k$, pak pro výpočet vzdálenosti z bodu q do R využíváme vztahu:

$$MINDIST(q, R) = \sqrt{\sum_{i=1}^{d} (q_i - r_i)^2}$$

Pro hodnotu r_i dále platí zda-li je $q_i < s_i$, pak $r_i = s_i$, pokud $q_i > t_i$, $r_i = t_i$ jinak $r_i = q_i$. [2]



Obrázek 10: Vzdálenost mindist, zdroj [2]

Analogicky, pak vzdálenost *MAXDIST* vyjadřuje maximální možnou vzdálenost mezi bodem q a obdélníkem R.

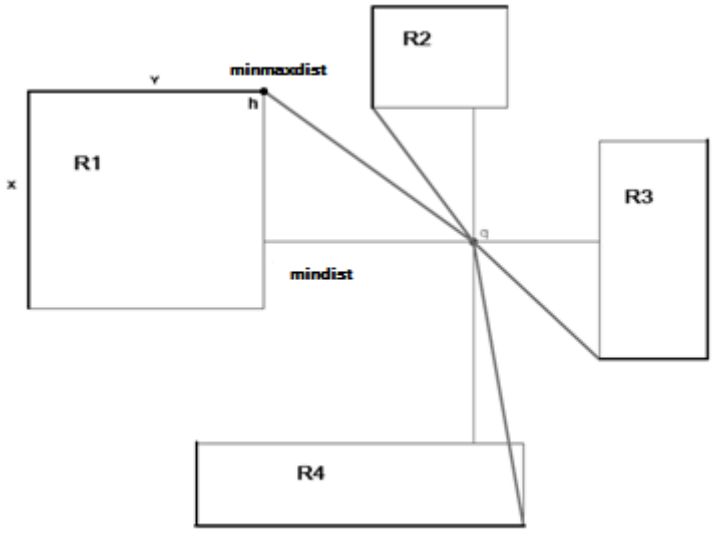
2.8.2 Minmaxdist

Jedná se o výpočet maximální vzdálenosti v každé dimenzi a výběr jejich minima. Pro parametry vzorce dále platí podmínka, jestliže $q_k \leq \frac{(s_k + t_k)}{2}$, pak $rm_k = s_k$, jinak $rm_k = t_k$. Druhá podmínka je, jestliže $q_i \geq \frac{(s_i + t_i)}{2}$, pak $rM_i = s_i$ jinak $rM_i = t_i$. [2]

$$\min(|q_k - rm_k|^2 + \sum_{i=1, i \neq k}^{n} |q_k - rM_i|^2)$$

Obrázek (Obrázek 11) naznačuje vztah mezi vzdáleností *MINMAXDIST* a *MINDIST*, pro které platí:

$$MINMAXDIST(q, R) \geq MINDIST(q, R)$$



Obrázek 11: Vzdálenost minmaxdist, zdroj [autor]

Při výpočtu *MINMAXDIST(q,R1)*, se v první řadě zjistí nejvzdálenější hrany obdélníku *R1* a to pro každou dimenzi. V případě dvourozměrného obdélníku se jedná o hrany *X* a *Y*. Z těchto dvou nalezených hran se vezme ten bod, který je bodu *q* nejblíže, tzn. platí:

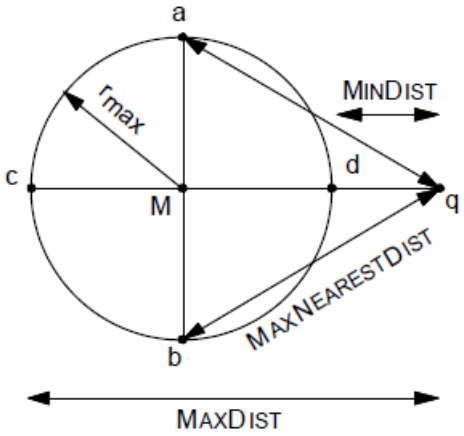
$$MINMAXDIST(q, R1) = \sqrt{\sum_{i=1}^{d} (q_i - h_i)^2}$$

2.8.3 Maxnearestdist

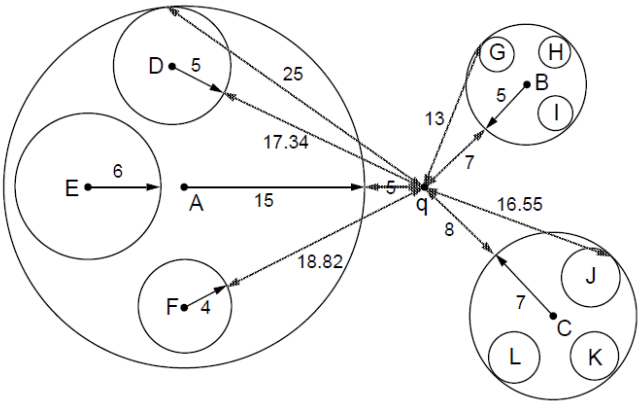
Maximální možná vzdálenost mezi objektem dotazu *q* a nejbližším sousedem v *e*. Obrázek (Obrázek 12) níže znázorňuje vztah mezi vzdálenostmi *MINDIST*, *MAXNEARESTDIST* a *MAXDIST* za předpokladu euklidovské metrické vzdálenosti v případě, že objekt *e* je ve

formě minimálního kruhu a je tvořen dalšími objekty ležících v něm. V takovém to případě, je hodnota *MAXNEARESTDIST* rovna: [1]

$$MAXNEARESTDIST(q,e) = \sqrt{d(q,M)^2 + {r_{max}}^2}$$



Obrázek 12: Vzdálenost maxnearestdist, zdroj [1]



MinDist(q,A) = 5
MinDist(q,B) = 7
MinDist(q,C) = 8
MaxNearestDist(q,A) = 25
MaxNearestDist(q,B) = 13
MaxNearestDist(q,C) = 16.55

Obrázek 13: Příklad výpočtu maxnearestdist, zdroj [1]

2.9 Algoritmus větví a hranic využívající princip procházení do hloubky[10]

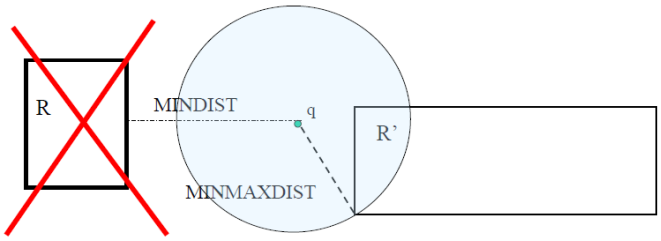
Tento algoritmus je postavený na využití speciálních metrických vzdáleností a několika základních prořezávacích pravidlech, které vedou k urychlení získání *k*-nejbližších sousedů. Algoritmus je použitelný především nad datovou strukturou R-strom. Výhodou těchto pravidel je znovu potenciální odstranění velké části prohledávaného prostoru. [2] [3]

2.9.1 Prořezávací pravidlo H1

Pravidlo uplatněné při průchodu strukturou a výběru vhodných uzlů do aktivního seznamu (Obrázek 14). Uzel s *MBR R* není zařazen do dalších výpočtu, jestliže existuje jiný uzel s *MBR R'*, pro který platí:

[10] Depth-first branch and bound.

$$MINDIST(q, R) > MINMAXDIST(q, R')$$

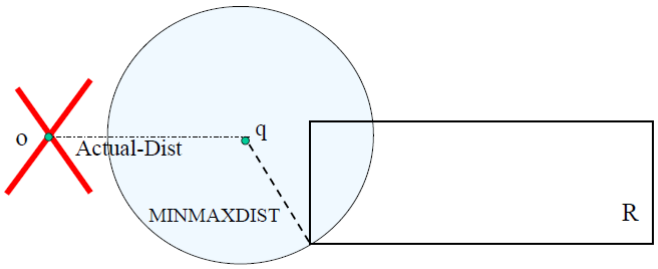


Obrázek 14: Prořezávací pravidlo H1, zdroj [2]

2.9.2 Prořezávací pravidlo H2

Pravidlo uplatitelné znovu ve stejném případě jako předchozí s tím rozdílem, že jde o vyloučení samotných objektů z dalšího vyhledávání (Obrázek 15). Pro prořezávací pravidlo *H2* platí, že objekt *o* je vyřazen, pokud existuje nějaký uzel s *MBR R*, pro který platí:

$$Actual\text{-}Dist\ (q, o) > MINMAXDIST(q, R)$$

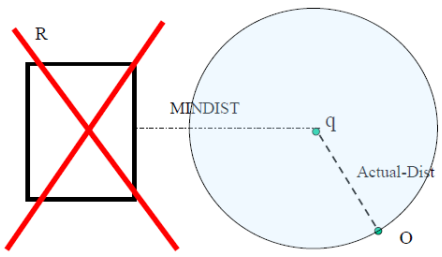


Obrázek 15: Prořezávací pravidlo H2, zdroj [2]

2.9.3 Prořezávací pravidlo H3

Na rozdíl od předchozích dvou pravidel, je toto aplikovatelné pro uzly již odebírané z aktivního seznamu, nikoli pro výběr uzlů do aktivního seznamu (Obrázek 16). Pravidlo ignoruje všechny uzly s *MBR R*, pokud je nalezen objekt *o*, pro který platí:

$$MINDIST(q, R) > Actual\text{-}Dist\ (q, o)$$



Obrázek 16: Prořezávací pravidlo H3, zdroj [2]

Princip algoritmu *depth-first branch and bound* s výše uvedenými prořezávacími pravidly *H1*, *H2* , *H3* je uveden níže. Výhoda algoritmu je v jeho složitosti, která je pouze logaritmická v počtu uzlů stromu. Vrací pouze jednoho nejbližšího souseda, v případě že má být algoritmus nasazen k hledání *k*-nejbližších sousedů, musí se k prořezávání využít vzdálenost ke *k*-tému nejbližšímu sousedovi. [2] [3]

Algoritmus 6

```
oNN = null; //výsledný nejbližší soused                          1
dist = ∞; //vzdálenost mezi oNN a q                              2
ABL = new AktivniSeznam();                                       3
  foreach A.seznamPrvku()                                        4
    ep = A.dalsiPotomek();                                       5
    ABL.pridej(ep);                                              6
  konec foreach;                                                 7
    while !ABL.jePrazdny()                                       8
        ep = ABL.odeberPrvni();                                  9
        if ep.jeList()                                           10
            foreach ep.seznamObjektu()                           11
                o = ep.dalsiObjekt();                            12
                if vzdalenost(q,o) < dist                        13
                    dist = vzdalenost(q,o);                      14
                    oNN = o;                                     15
                    //nalezení nového nej. souseda               16
                konec if;                                        17
            konec foreach;                                       18
        else                                                     19
            if Mindist(q,ep) > dist //H3                         20
                //ignorování ep                                  21
            else                                                 22
                foreach ep.seznamPotomku()                       23
                    epnovy = ep.dalsiPotomek();                  24
                    fronta.pridejPrvni(epnovy);  //H2,H1         25
                konec foreach;                                   26
            konec if;                                            27
        konec if;                                                28
    konec while;                                                 29
return oNN;                                                      30
```

2.10 Využití vzdálenosti typu maxnearestdist

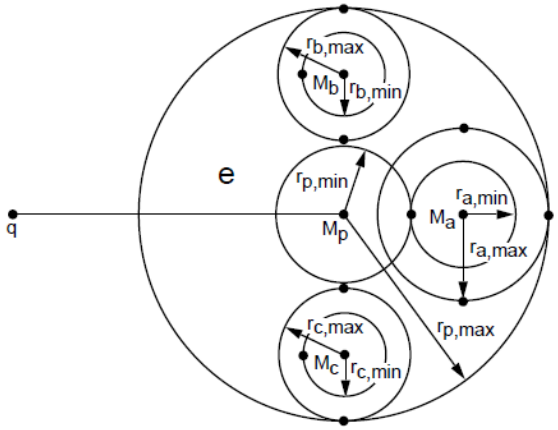
Jak už název napovídá, existují i takové algoritmy, které využívají speciální vzdálenosti *MAXNEAREARESTDIST*. V následujících dvou podkapitolách jsou uvedeny dva zástupci, které se liší pouze strategií hledání nebližších sousedů respektive procházení stromové hierarchie. Hlavní výhodou algoritmů, je jejich rychlejší konvergence k finální hodnotě D_k, která slouží k rozhodování, zda daná větev stromu má či nemá být dále zpracovávána. Obecně pak tento princip může vést k rychlejšímu vyhledání *k*-nejbližších sousedů.

Rozdílem oproti předešlým algoritmům je v prioritní frontě *L*. Ta ukládá jak listy stromu s odpovídající vzdáleností od *q*, tak i nelistové uzly. Těm je před vložením do fronty *L*

nastavena vzdálenost *MAXNEARESTDIST* ke *q*. Ukládání obou typů dat vede k rychlejšímu zaplnění prioritní fronty a tím i k rychlejšímu získání vzdálenosti D_k.

Při zpracovávání každého nelistového uzlu vkládáme do prioritní fronty *L* všechny jeho potomky, které jsou blíže než je *k*-tý nejbližší objekt. Zároveň musíme daný nelistový uzel z fronty odebrat tak abychom zaručili, že mezi žádnými prvky fronty *L* nebude vztah předek-potomek. [1] [5]

Obrázek níže (Obrázek 17) ukazuje, že se počet prvků v prioritní frontě *L* může snížit, když *MAXNEARESTDIST* tří různých neobjektových uzlů e_a, e_b, e_c v uzlu *e* je větší než aktuální hodnota D_k, která je rovna *MAXNEARESTDIST* vzdálenosti k *e*. [1]



Obrázek 17: Ukázka redukce dále prohledávaných uzlů, zdroj [1]

2.10.1 Algoritmus využívající princip procházení do šířky a speciální vzdálenosti maxnearestdist[11]

Oba zástupci se skládají ze tří metod, z něhož dvě jsou totožné. Jedná se o metodu *vlozDoL*, která slouží ke vkládání nového prvku, jak listového tak nelistového, do prioritní fronty *L*. Jejím dalším úkolem je i odebrat poslední nelistové prvky se stejnou vzdáleností, v případě kdy je prioritní fronta plná a samozřejmě aktualizovat vzdálenost D_k.

Druhá metoda *odeberZL* je potřebná pro odebrání specifikovaného prvku z prioritní fronty *L*, tak aby se zaručilo, že v ní není uložen právě zpracovávaný prvek a tudíž žádné prvky nebudou ve vztahu předek-potomek.

Poslední metoda je hlavní metodou, která určuje princip procházení stromové hierarchie, tzn. pořadí zpracovávaných prvků je dáno jejich vzdáleností od *q*.

Algoritmus 7

```
vlozDoL(↓ep,↓vzdalenost) //metoda vkládání nových prvku do L          1
    if velikost(L) == k                                               2
        h = L.odeberMax();                                            3
```

[11] Maxnearest best-first.

```
          if !h.jeObjekt()                                                    4
            while !L.jePrazdna() and !L.vratMax().jeObjekt()                  5
              and vzdalenost(L.vratMax()) == vzdalenost(h)                    6
                   L.odeberMax();                                             7
            konec while;                                                      8
          konec if;                                                           9
     konec if;                                                                10
     L.pridej(ep,vzdalenost);                                                 11
     if velikost(L) == k                                                      12
          if vzdalenost(L.vratMax()) < Dk                                     13
               Dk = vzdalenost(L.vratMax());                                  14
          konec if;                                                           15
     konec if;                                                                16
konec vlozDoL;                                                                17
                                                                              18
odeberZL(↓ep)                                                                 19
     if Maxnearestdist(q,ep) < Dk or                                          20
        (Maxnearestdist(q,ep) == Dk and vzdalenost(L.vratMax())==Dk           21
         and !L.vratMax().jeObjekt()                                          22
               L.odeber(ep);                                                  23
     konec if;                                                                24
konec odeberZL;                                                               25
                                                                              26
maxNearestDistBF(↓ep)                                                         27
     L.pridej(ep,Maxnearestdist(q,ep));                                       28
     fronta.pridej(ep,0);                                                     29
     while !fronta.jePrazdny()                                                30
          ep = fronta.odeber();                                               31
          If vzdalenost(ep) > Dk                                              32
              return L;                                                       33
          else odeberZL(ep);                                                  34
          konec if;                                                           35
          if ep.jeList()                                                      36
               foreach ep.seznamPotomku()                                     37
                    o = ep.dalsiPotomek();                                    38
                    if vzdalenost(q,ep)<Dk or (vzdalenost(q,ep)==Dk           39
                      and velikost(L)<k)                                      40
                          vlozDoL(o,vzdalenost(q,o));                         41
                    konec if;                                                 42
          else                                                                43
               foreach ep.seznamPotomku()                                     44
                    epnovy = ep.dalsiPotomek();                               45
                    if Mindist(q,epnovy) < Dk                                 46
                         if Maxnearestdist(q,epnovy) < Dk                     47
                            vlozDoL(epnovy,Maxnearestdist(q,epnovy);          48
                         konec if;                                            49
                         fronta.pridej(epnovy,Mindist(q,epnovy));             50
                    konec if;                                                 51
               konec foreach;                                                 52
          konec if;                                                           53
     konec while;                                                             54
   return L;                                                                  55
konec maxNearestDistBF;                                                       56
```

2.10.2 Algoritmus využívající princip procházení do hloubky a speciální vzdálenosti maxnearestdist[12]

Zástupce *depth-first* strategie je odlišný ve své hlavní metodě. Obsahuje rekurzní volání sama sebe, čímž zaručuje procházení stromové hierarchie do hloubky a zároveň nepotřebuje pomocnou prioritní frontu, která by mu určovala pořadí zpracovávaných prvků.

Algoritmus 8

```
maxNearestDistDF(↓ep)                                                    1
     if ep.jeList()                                                      2
          foreach ep.seznamPotomku()                                     3
               o = ep.dalsiPotomek();                                    4
               if vzdalenost(q,ep)<Dk or (vzdalenost(q,ep)==Dk           5
               and velikost(L)<k)                                        6
                    vlozDoL(o,vzdalenost(q,o));                          7
          konec if;                                                      8
     else                                                                9
               foreach ep.seznamPotomku()                                10
                    epnovy = ep.dalsiPotomek();                          11
                    if Mindist(q,epnovy) > Dk                            12
                         break;                                          13
                    else                                                 14
                         vlozDoL(epnovy,Maxnearestdist(q,epnovy);        15
                    konec if;                                            16
               konec foreach;                                            17
               foreach ep.seznamPotomku()                                18
                    epnovy = ep.dalsiPotomek();                          19
                    if Mindist(q,epnovy) <= Dk                           20
                         odeberZL(epnovy);                               21
                         MaxnearestdistDF(epnovy);                       22
                    konec if;                                            23
               konec foreach;                                            24
     konec if;                                                           25
konec maxNearestDistDF;                                                  26
```

2.11 Aproximační algoritmy

V mnoha aplikacích není zásadní přesný výsledek, ale spíše dosáhnutí co největší rychlosti vyhledávání. Existuje několik různých variant jak vylepšit při hledání nejbližšího souseda rychlost na úkor přesnosti. [1]

2.11.1 Algoritmus aproximační chybové tolerance[13]

Jednou z hlavních metod aproximačního hledání nejbližšího souseda je využití aproximační chybové tolerance ε. Aproximační kritérium říká, že vzdálenost mezi bodem

[12] Maxnearest depth-first.
[13] Approximation error tolerance.

dotazu *q* a případným kandidátem na nejbližšího sousedem *o'*, je v rámci *(1 + ε)* násobku vzdálenosti od aktuálního nejbližšího souseda *o*. [1] [4]

$$d(q, o') \leq (1 + \varepsilon) * d(q, o)$$

Oba výše zmíněné algoritmy *best-first* i *depth-first k-nearest neighbor* mohou být modifikovány o aproximační chybový faktor. Je nutné podotknout, že první zmíněný je více přizpůsobivý. Níže je uvedený zástupce algoritmu *best-first*, hledající *k*-nejbližších sousedů s využitím aproximační chybové tolerance. Tento modifikovaný algoritmus má stejnou strukturu jako ten, ze kterého vychází (podkapitola 2.4).

Algoritmus 9

```
fronta = new PrioritniFronta();                                    1
L = new PrioritniFronta();                                         2
     fronta.pridej(koren);                                         3
     Dk = ∞;                                                       4
     while !fronta.jePrazdna()                                     5
          ep = fronta.odeber();                                    6
          if vzdalenost(q,ep) > Dk/(1+ε)                           7
               //k approx. nejbližších sousedů nalezeno            8
               return L;                                           9
          else if ep.jeList()                                      10
               foreach ep.seznamObjektu()                          11
                    o = ep.dalsiObjekt();                          12
                         if vzdalenost(q,o) < Dk                   13
                              L.pridej(o);                         14
                              //případná aktualizace Dk            15
                         konec if;                                 16
               konec foreach;                                      17
          else                                                     18
               foreach ep.seznamPotomku()                          19
                    epnovy = ep.dalsiPotomek();                    20
                    if vzdalenost(q,epnovy) < Dk/(1+ε)             21
                         fronta.pridej(epnovy);                    22
                    konec if;                                      23
               konec foreach;                                      24
          konec if;                                                25
     konec while;                                                  26
```

Algoritmus řadí prvky v prioritní frontě *fronta* v rostoucím pořadí jejich vzdálenosti od *q*. Do této fronty se vkládají pouze ty prvky, jejichž vzdálenost je menší než vzdálenost ke *k*-tému nejbližšímu sousedovi poníženou o chybovou toleranci *(1+ ε)*. Z tohoto vztahu vyplívá, že pokud bude chybová tolerance rovna nule *(ε = 0)*, pak ve výsledku dostáváme přesné nejbližší sousedy.

V případě aplikace na základní inkrementální *best-first* algoritmus (podkapitola 2.2), stačí jen prvky e_p v prioritní frontě *fronta* řadit dle vzdálenosti e_p a objektu *q* vynásobenou chybovou tolerancí *(1+ε)* tedy:

$$(1 + \varepsilon) * d(q, e_p)$$

Stejnou techniku a to aplikaci chybové tolerance, je možná implementovat pro většinu popsaných algoritmů. [1]

2.11.2 Algoritmus založený na pravděpodobnosti[14]

Základem je schopnost předpovědět pravděpodobnost, že v daném uzlu, bude respektive nebude dostupný objekt *o* takový, že pro jeho vzdálenost od *q* platí: [4]

$$d(q, o) < ActualDist$$

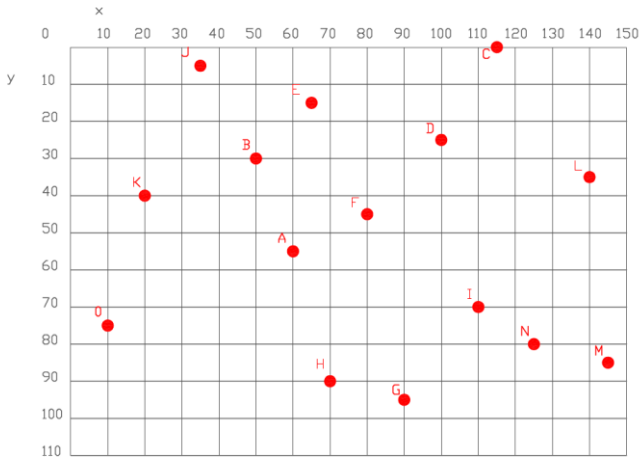
K prořezávání se používá parametr α, který zaručuje, že každý uzel e_p je vynechán z dalšího zkoumání, pokud pro něj platí:

$$\frac{ActualDist - ep.MinDist}{ep.MaxDist - ep.MinDist} = \alpha$$

3 Základní charakteristiky vybraných datových struktur

Kapitola popisuje charakteristiky vybraných datový struktur, nad kterými budou implementovány některé z vyhledávajících algoritmů uvedených v předchozí kapitole. Hlavní náplní je především představení vybraných datových struktur a přiblížit čtenáři postup jejich budování za předpokladu, že neuvažujeme dynamický proud vstupních dat. Pro lepší pochopení byly základní kroky těchto operací graficky znázorněny. Každá podkapitola navíc obsahuje i grafickou podobu výsledné dekomponované oblasti, která charakterizuje danou datovou strukturu.

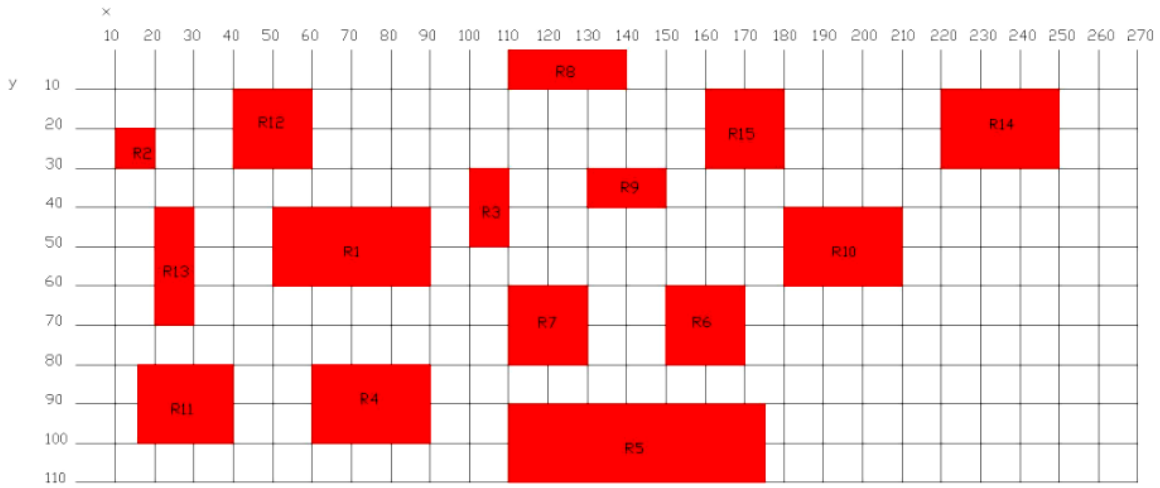
Pro popis operace statického vybudování bylo využito experimentálních dat malého rozsahu. U většiny struktur jsou používána dvourozměrná bodová data, jejichž geografická poloha je znázorněna na obrázku (Obrázek 18).



Obrázek 18: Vzorová bodová data, zdroj [autor]

[14] Probabilistic approximation.

Pro názorné vysvětlení stavby datové struktury R-strom, jsou použita dvourozměrná data jiného charakteru (Obrázek 19).



Obrázek 19: Vzorová plošná data, zdroj [autor]

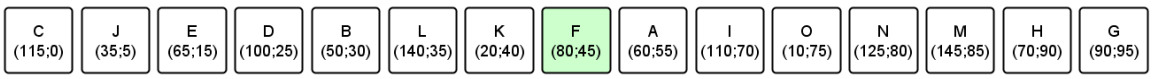
3.1 k-d strom

K-d stromy jsou binární stromy, kde hodnota *k* udává počet dimenzí prohledávaného prostoru a každý uzel představuje *k*-rozměrný bod. Prostor je rozdělován postupným zpracováváním jednotlivých os tak, že bodem dělení je volen medián souřadnic bodů v příslušném podintervalu. Tento postup vede k vytvoření vyváženého stromu. [1]

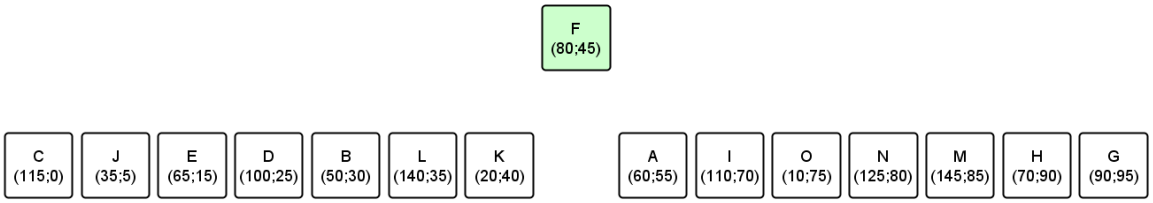
3.1.1 Vybudování struktury

Konstrukce spočívá především ve střídání os v každé úrovni stromu pro základ dělení zbylých dat. Pro každý uzel stromu platí, že všechny prvky v jeho levém podstromu budou mít hodnotu souřadnice, podle které se v dané úrovni dělilo, menší a naopak. Jak už bylo řečeno, dělení prvku do podstromů se provádí pomocí mediánů. Existují i jiné způsoby, které nám však nezaručí, že výsledný strom bude vyvážený.

Prvním krokem při budování 2-D stromu z předem známých dat, je nalezení optimálního kořene stromu, tak jak to platí u většiny hierarchických datových struktur. Nový kořen se nalezne jako medián setříděných dat dle zadané souřadnice. V našem případě bylo stanoveno na první úrovni dělení podle souřadnice y. Následující obrázky (Obrázek 20, Obrázek 21) zobrazují výše popsaný postup, kdy se kořenem stromu stává bod *F* a zbylá data jsou rozdělena do levého a pravého podstromu.

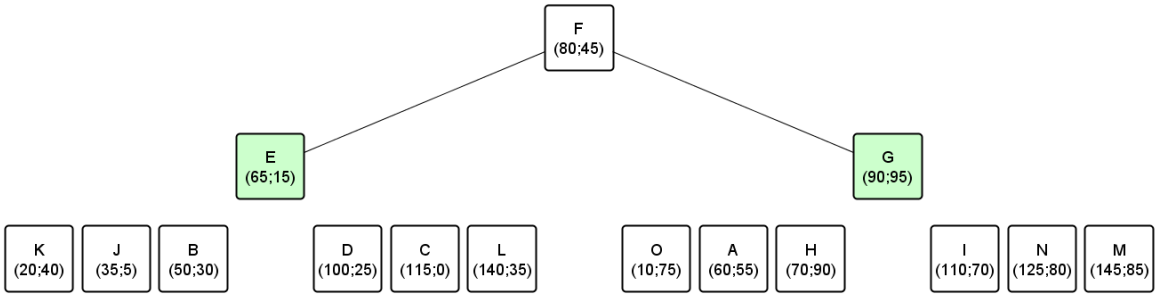


Obrázek 20: 2-D strom - seřazení vstupních dat, zdroj [autor]



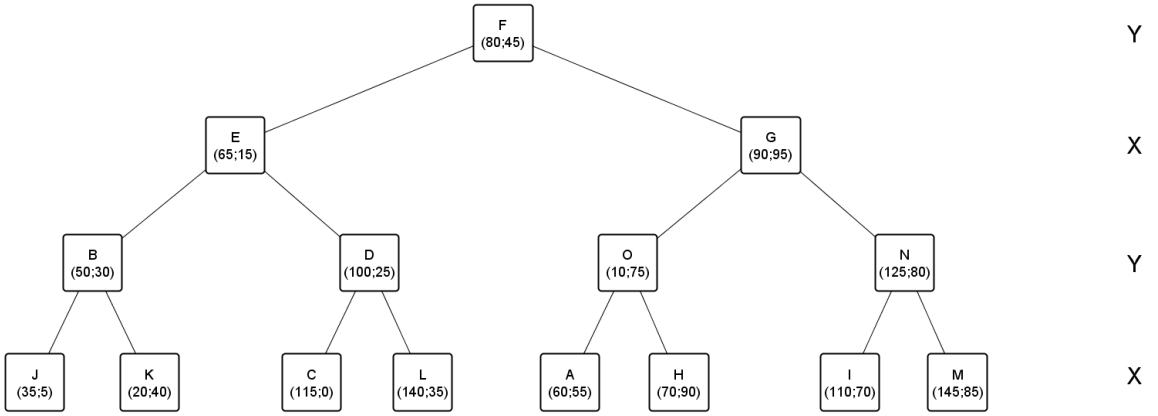
Obrázek 21: 2-D strom - kořen stromu, zdroj [autor]

Následující krok je totožný s prvním krokem, ale provádí se jednotlivě pro každý podstrom (Obrázek 22). V tomto případě se data setřídí podle souřadnice x, což vede jak k nalezení nových přímých potomků kořene, tak i k rozdělení zbylých dat do dalších podstromů.

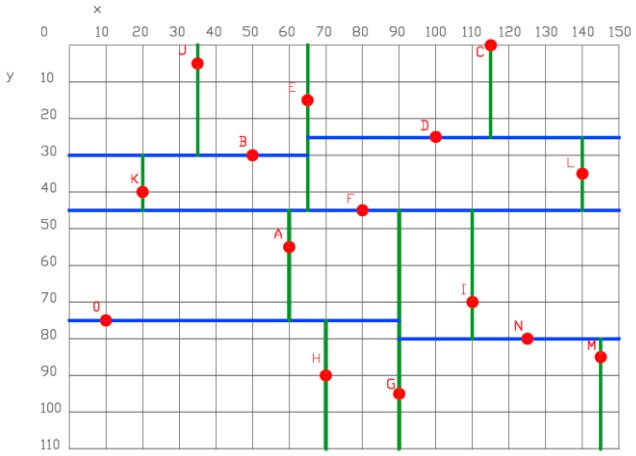


Obrázek 22: 2-D strom - druhý krok, zdroj [autor]

Dva předešlé kroky se nadále rekurzivně opakují do doby, kdy každý podstrom neobsahuje pouze jeden prvek, který je následně zařazen jako list stromu. Výsledný 2-D strom pro vzorová experimentální data společně s ukázkou výsledného děleného prostoru, je k vidění na obrázku níže (Obrázek 23, Obrázek 24).



Obrázek 23: Výsledný 2-D strom, zdroj [autor]



Obrázek 24: Dělení prostoru - 2-D strom, zdroj [autor]

3.2 Prioritní vyhledávací strom

Jedná se o datovou strukturu, která kombinuje princip binárního vyhledávacího stromu a binární haldy. Každý prvek struktury obsahuje dvě informace, z nichž jedna je klíč a druhá je priorita. Obecně pak prioritní vyhledávací strom slouží k ukládání dvourozměrných dat, přičemž princip binárního vyhledávacího stromu je uplatněn na x-ové souřadnici a princip binární haldy na souřadnici y-ové.

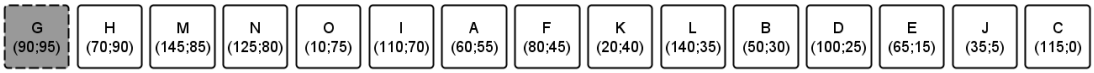
Dále pak každý prvek, stejně jako je tomu u binárního stromu, obsahuje pouze dva potomky, pro které navíc platí, že levý podstrom daného prvku obsahuje pouze prvky s hodnotou x-ové souřadnice, která je menší než je hodnota tzv. atributu *hranice*[15] u daného prvku. Obdobně toto pravidlo platí pro pravý podstrom, který obsahuje pouze prvky s větší hodnotou x-ové souřadnice než je hodnota atributu *hranice*. Základní odlišnost je pak v principu traverzování nově vkládaného prvku hierarchickou strukturou. Při traverzování se neprovádí porovnání x-ové souřadnice vkládaného prvku s x-ovou souřadnicí každého prvku struktury, tak jak je to v případě binárního vyhledávacího stromu, ale s hodnotou atributu *hranice* daného prvku. [1] [6]

3.2.1 Vybudování struktury

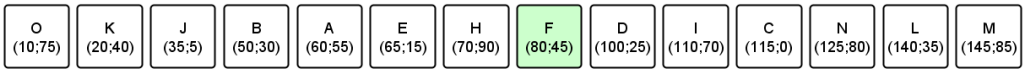
Abychom mohli vybudovat vyvážený prioritní vyhledávací strom, je nutné před samotnou stavbou znát všechna vstupní data. Tyto vstupní data, která v našem případě reprezentují dvourozměrné body, jsou v prvním kroku sestupně setříděna dle y-ové souřadnice. V setříděných datech vybereme nový kořenový prvek struktury, který má hodnotu y-ové souřadnice větší než je u všech ostatních dat tzn. platí princip haldového uspořádání dle souřadnice y. V druhém kroku jsou zbylá data setříděna dle x-ové souřadnice a to tentokrát vzestupně. V setříděných datech se nalezne medián, jehož x-ová souřadnice slouží k nastavení atributu *hranice* u kořenového prvku. Nalezení mediánu vede k rozdělení zbylých dat do levého a pravého podstromu, pro které se oba kroky rekurzivně opakují.

[15] Hodnota x-ové souřadnice mediánu všech potomků levého a pravého podstromu.

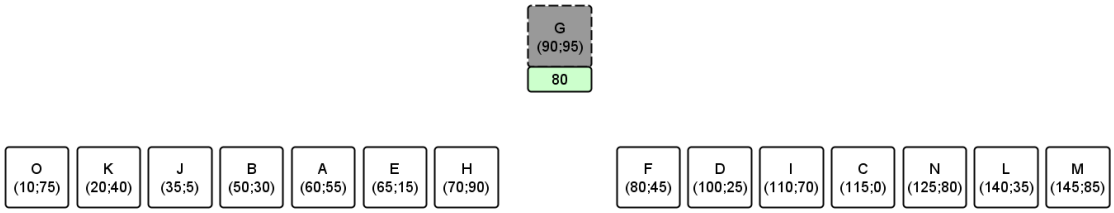
Počáteční krok nalezení prvku, který je zařazen jako kořen stromu v setříděných datech, je přiblížen na obrázkách níže (Obrázek 25, Obrázek 26, Obrázek 27). Je znázorněna i situace dalšího kroku, kdy se kořenovému prvku, v našem případě bodu *G* nastaví hodnota atributu *hranice*. Hodnota tohoto atributu se nastaví způsobem, jak bylo řečeno v úvodu této podkapitoly.



Obrázek 25: Prioritní vyhledávací strom - první krok, zdroj [autor]

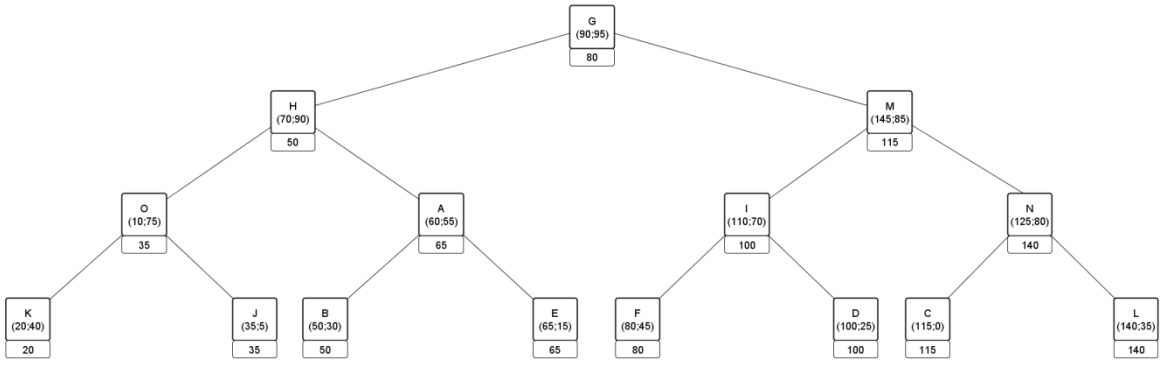


Obrázek 26: Prioritní vyhledávací strom - další krok, zdroj [autor]

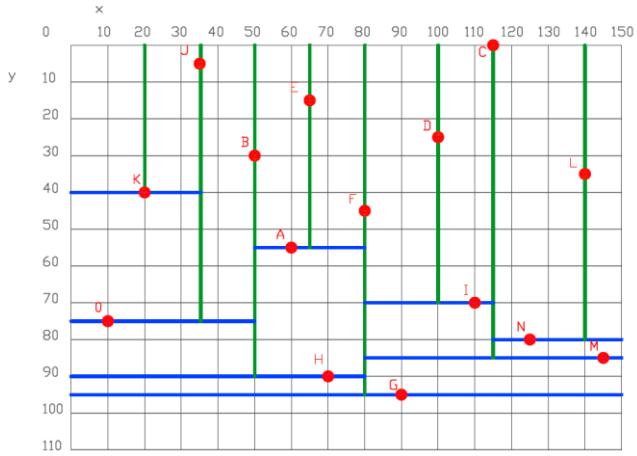


Obrázek 27: Prioritní vyhledávací strom - kořen stromu, zdroj [autor]

Po rekurzivním umístění všech prvků do struktury vznikne konečná podoba prioritního vyhledávacího stromu (Obrázek 28). Výsledné dělení dvojrozměrného prostoru jednoznačně prokazuje, že všechny prvky vzhledem k souřadnici y leží pod kořenem (Obrázek 29).



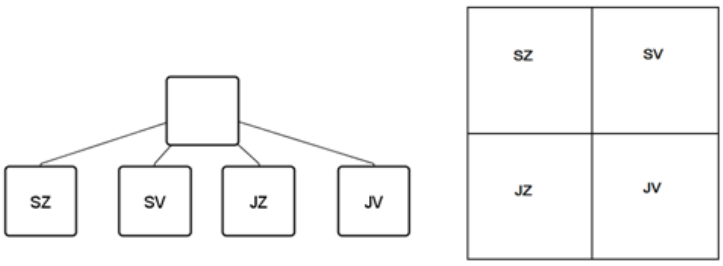
Obrázek 28: Výsledný prioritní vyhledávací strom, zdroj [autor]



Obrázek 29: Dělení prostoru - prioritní vyhledávací strom, zdroj [autor]

3.3 Quad strom

Obecně quad stromy jsou hierarchické datové struktury, jejichž princip je založen na rekurzivním rozkladu dvourozměrného prostoru. Výsledkem je rozdělení každého regionu do čtyř oblastí, které odpovídají jihozápadnímu, severozápadnímu, jihovýchodnímu a severovýchodnímu kvadrantu (Obrázek 30). Platí, že tyto oblasti mohou být čtvercové, obdélníkové či jakéhokoliv jiného tvaru a zároveň nemusejí být nutně stejné. V praxi se objevuje celá řada různých variant, které jsou klasifikovány podle typu dat nebo oblastí, které reprezentují. Dále si představíme pouze dvě z nich a to variantu bodovou a regionální.

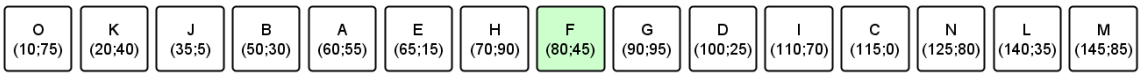


Obrázek 30: Quad strom - kvadranty, zdroj [autor]

Bodový quad strom (z angl. Point quadtree) je zobecněním binárního stromu, který obsahuje právě takový počet prvků, jaký je počet vstupních dat. Tvar struktury závisí na pořadí vkládaných prvků. Na druhé straně regionální quad strom (z angl. PR quadtree) dekomponuje dvourozměrnou oblast rekurzivně do čtyř stejných kvadrantů tak dlouho, dokud každý z nich neobsahuje maximálně jeden objekt. Pokud některý z dvourozměrných bodů leží na hranici více těchto kvadrantů, je zařazen pouze do jednoho z nich. Na rozdíl od bodového quad stromu, jsou objekty uloženy až v samotných listech stromu. Nevýhodou rekurzivního rozkladu, který může vést k rozdílné hloubce každého z listů, je nevyváženost struktury, která nemůže být ovlivněna ani za předpokladu předem známých dat. [1] [6]

3.3.1 Vybudování bodové varianty

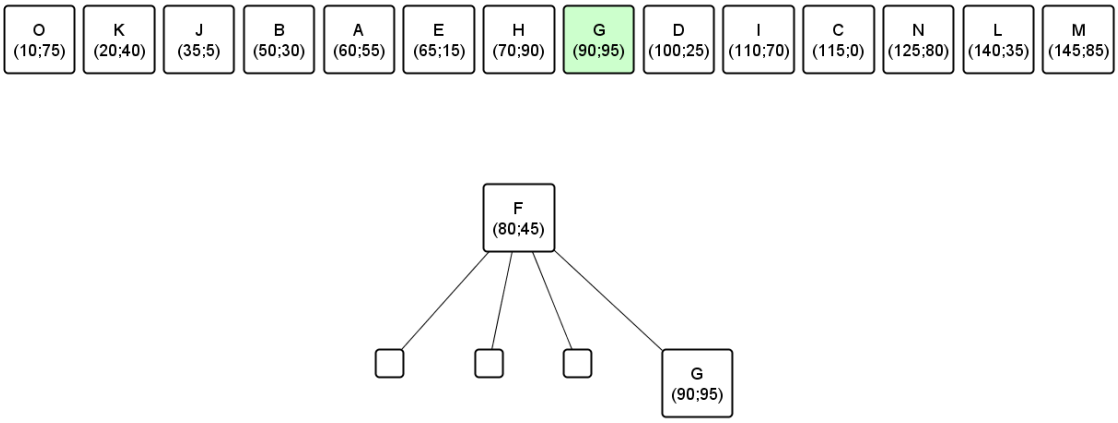
Při budování bodového quad stromu za předpokladu, že jsou vstupní data předem známá, můžeme do jisté míry ovlivnit vyváženost a to nejen vhodnou volbou kořene stromu. Základním krokem je seřazení vstupních dat primárně dle x-ové souřadnice a sekundárně pak dle souřadnice y-ové. Z těchto setříděných dat se následně vybere medián, který je umístěn jako kořen stromu. Zbylá data se opět setřídí dle obou souřadnic a medián je umístěn na správnou pozici v datové struktuře. Tento postup se rekurzivně opakuje do doby, kdy jsou k dispozici ještě vstupní data. Princip budování je blíže znázorněn graficky na následujících obrázkách.



Obrázek 31: Bodový quad strom – kořenový prvek stromu, zdroj [autor]

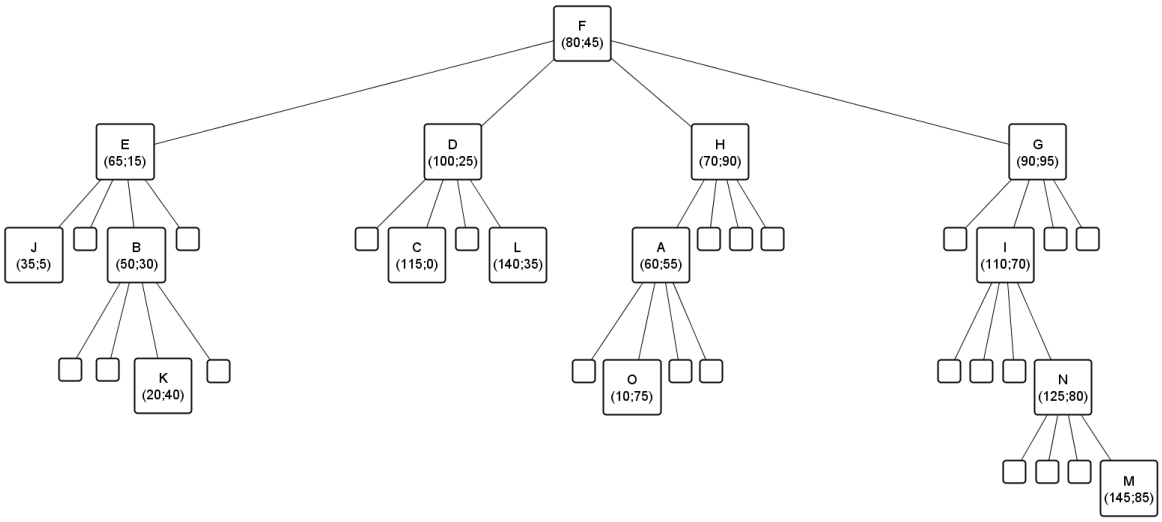
Na předešlém obrázku (Obrázek 31) jsou zobrazena setříděná vstupní data, ve kterých je následně nalezen kořen stromu. V našem případě se jedná o bod *F*. Zařazení nového kořene stromu má za následek dekompozici dvourozměrného prostoru do čtyř nových oblastí, jejichž poloha je dána souřadnicemi kořene.

Následně je vždy ze zbylých seřazených vstupních dat vybrán medián, který je vložen dynamickou operací do datové struktury. Vkládaný prvek traverzuje hierarchickou strukturou a vždy pokračuje tou větví, jejíž region obsahuje právě souřadnice daného vkládaného prvku. Nový prvek traverzuje do té doby, dokud nenarazí na prázdný list stromu, kde je následně umístěn (Obrázek 32).

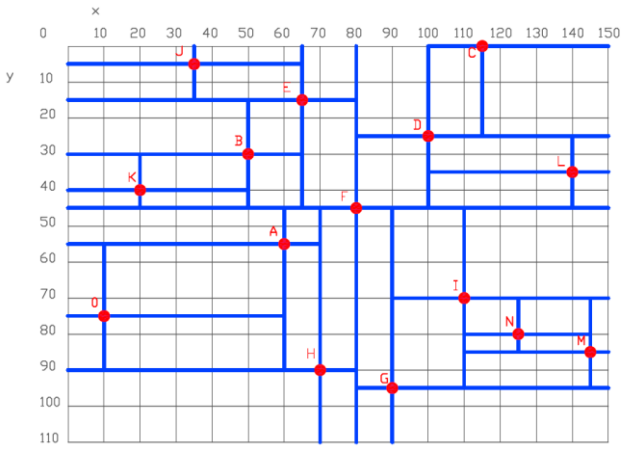


Obrázek 32: Bodový quad strom - vložení dalšího prvku, zdroj [autor]

Obrázky níže graficky ilustrují výsledný bodový quad strom (Obrázek 33) a výslednou dekomponovanou oblast (Obrázek 34).



Obrázek 33: Výsledný bodový quad strom, zdroj [autor]

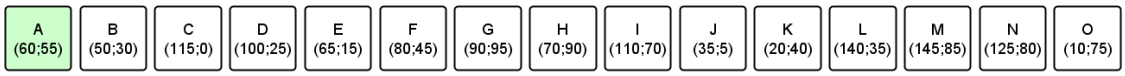


Obrázek 34: Dělení prostoru - bodový quad strom, zdroj [autor]

3.3.2 Vybudování regionální varianty

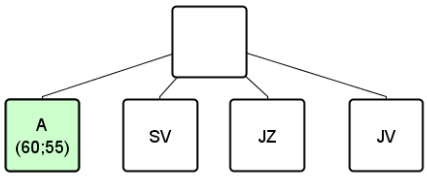
Z důvodu principu dekompozice prostoru regionálním quad stromem, není žádný rozdíl při budování struktury statickým či dynamickým proudem vstupních dat. Obě situace totiž vedou k totožné stavbě datové struktury.

Při vkládání prvku v jejich abecedním pořadí (Obrázek 35), dochází k traverzování hierarchickou strukturou tak, že v každém uzlu stromu prvek pokračuje takovou větví, jejíž region obsahuje souřadnice vkládaného prvku. Průchod stromem končí, dosáhne-li listového uzlu. Tento uzel může být buď prázdný, nebo může obsahovat již nějaký jiný prvek. V první situaci je zřejmé, že se na pozici prázdného uzlu zařadí nový prvek. V opačném případě musí dojít k rekurzivnímu rozdělení daného uzlu do čtyř stejných kvadrantů a to do doby, kdy každý z nich nebude obsahovat pouze jeden.



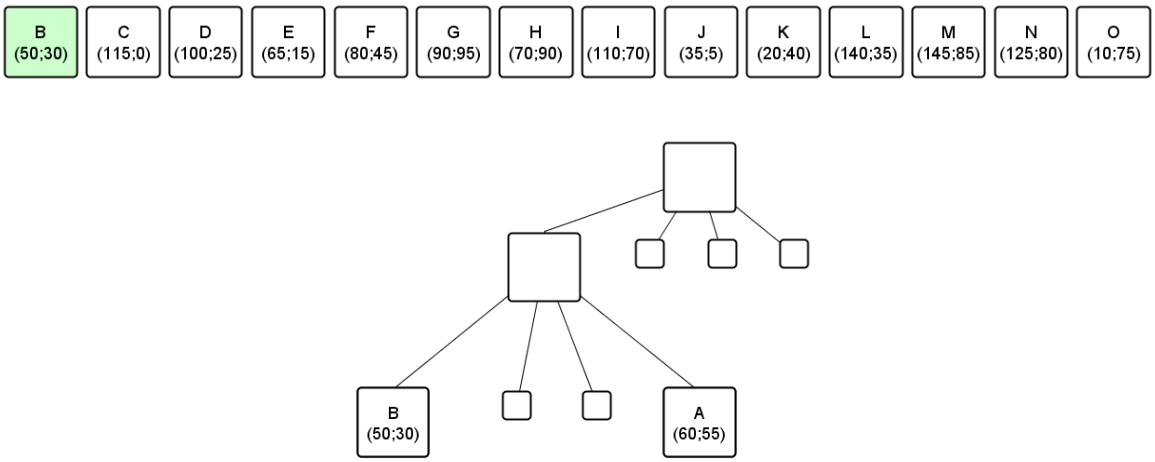
Obrázek 35: Regionální quad strom - pořadí vstupních dat, zdroj [autor]

Následující obrázek (Obrázek 36) naznačuje situaci po vložení prvního prvku, v našem případě bodu A, a jeho výsledné umístění ve stromové struktuře. Dvourozměrný prostor je rozdělen do čtyř kvadrantů, na které odkazuje kořen stromu, z nichž do severozápadního byl umístěn námi vkládaný prvek *A*.



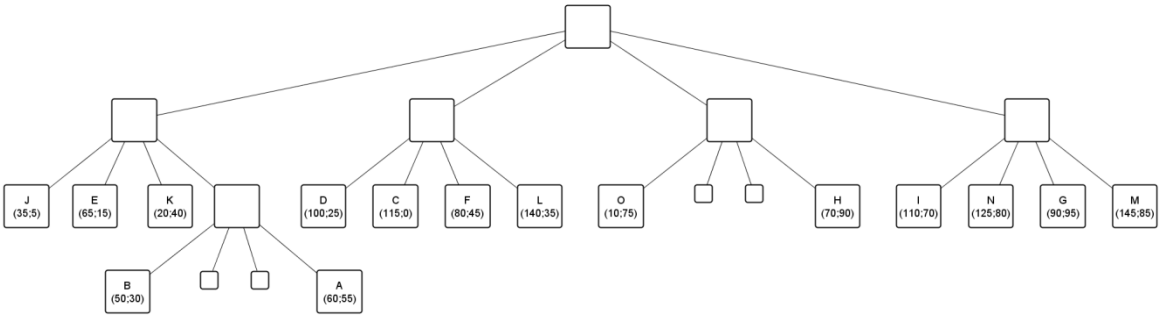
Obrázek 36: Regionální quad strom - první krok, zdroj [autor]

Po vložení dalšího prvku nastane situace, kdy se daný uzel musí rekurzivně dělit. Prvek *B* při traverzování strukturou byl vinou své geografické polohy zařazen do stejného regionu, jako předchozí prvek *A*. Jinými slovy daný region obsahuje dva prvky a to je z hlediska principu quad stromu nepřípustné. Obrázek níže (Obrázek 37) ilustruje stav po rekurzivním dělení daného uzlu.

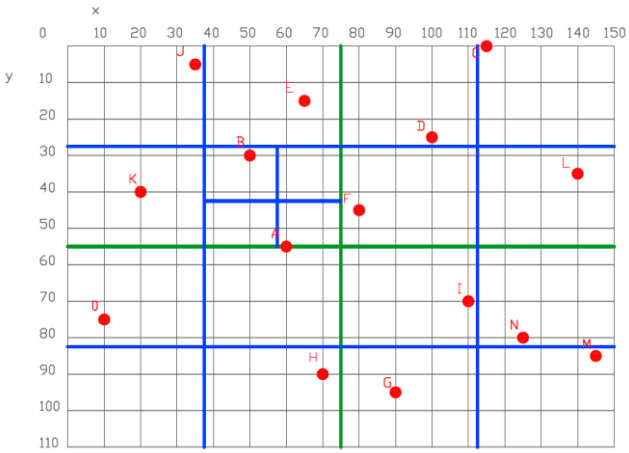


Obrázek 37: Regionální quad strom - další krok, zdroj [autor]

Výsledný regionální quad strom po zařazení všech vstupních dat, které jsou obsaženy jen v samotných listech stromu a výsledný vzhled děleného prostoru je přiblížen na následujících obrázkách (Obrázek 38, Obrázek 39).



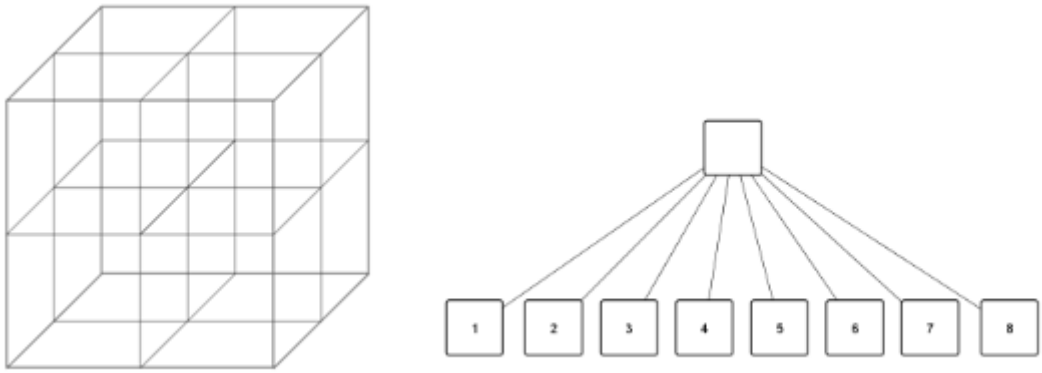
Obrázek 38: Výsledný regionální quad strom, zdroj [autor]



Obrázek 39 : Dělení prostoru - regionální quad strom , zdroj [autor]

3.4 Oktalový strom

Oktalový strom je hierarchická datová struktura, která je zobecněním regionálního quad stromu a to pro trojrozměrný prostor. Způsob dekompozice prostoru je principiálně stejný, jako v případě quad stromu, jen danou oblast rozkládá do osmi stejných oktantů, které pro oktalový strom mají tvar trojrozměrné krychle (Obrázek 40). Každá tato krychle může opět obsahovat pouze jeden objekt, v opačném případě se dále rekurzivně dělí. [1] [6]



Obrázek 40: Oktalový strom - oktanty, zdroj [autor]

3.4.1 Vybudování struktury

Při stavbě oktalového stromu platí stejná pravidla, jako u regionálního quad stromu. Odlišnost je pouze v traverzování nového prvku, kdy prvek v každém uzlu postupuje jednou z osmi větví. Z důvodu stejného principu, který byl popsán v předchozí podkapitole a především složitostí zobrazení daného postupu, který se týká trojrozměrného prostoru nebudu daný problém dále rozvádět.

3.5 R-strom

R-strom je multidimenzionální hierarchicky indexovaná datová struktura podobná B-stromu, kterou v roce 1984 definoval Antonin Guttman. Je charakterizována výškovou vyvážeností a minimálně polovičním zaplněním uzlů. Slouží k indexaci prostorových objektů, od bodů až po mnohem složitější objekty, které jsou ve struktuře reprezentovány jako tzv. minimální ohraničující oblasti. Ty jsou uložené pouze v samotných listech stromu. Pro tuto oblast se v praxi využívá zkratka *MBR* (z angl. minimum bounding rectangle). Tento *MBR* představuje ve dvourozměrném prostoru nejmenší obdélník případně čtverec, který pojme daný objekt. V trojrozměrném prostoru se pak jedná o kvádr nebo krychli. R-strom dělí celý prostor do potencionálně se překrývajících *MBR*, kde kořen obsahuje právě minimální oblast potřebnou pro ohraničení všech objektů, ze kterých je struktura vytvořena.

Datová struktura je tvořena dvěma typy uzlů. Listové uzly obsahují *MBR* indexovaných objektů a ukazatel na celý objekt, který má za následek potlačení fragmentace. Na druhé straně vnitřní uzly nelistové, obsahují regiony respektive oblasti. Jedná se o *MBR* všech svých potomků a ukazatelů na ně. U struktury jsou zavedeny pojmy maximální počet položek v každém uzlu tzv. faktor větvení M a minimální počet položek v uzlu m, pro který platí vztah:

$$m \geq \left\lceil \frac{M}{2} \right\rceil$$

Operace vkládání respektive mazání zaručuje, že počet položek v uzlu bude mezi m a M. Výška stromu nikdy nepřekročí $log_m N - 1$, kde N je počet objektů uložených ve všech listech stromu. [6] [7] [8] [9]

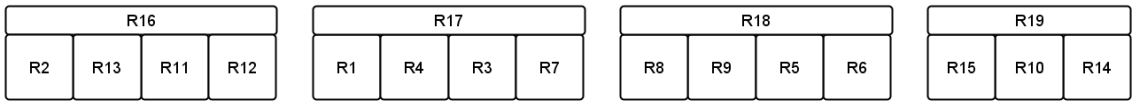
3.5.1 Vybudování struktury

Předpokládáme-li statický charakter vstupních dat, vybudování vyváženého R-stromu, je odlišné než ve všech doposud charakterizovaných datových strukturách. Základní odlišností je budování stromu od samotných listů nikoli od kořene stromu. Počátečním krokem je seřazení vstupních dat respektive jejich *MBR*. Ty následně slouží pro konstrukci listových uzlů. Dalším krokem je rekurzivní budování dalších úrovní takovým způsobem, že uzly z nižší úrovně slouží automaticky, jako položky uzlů úrovně vyšší. Při konstrukci každého uzlu struktury, musíme dbát na jeho maximální zaplnění a zároveň musíme kontrolovat počet zbylých dat, tak aby některý z uzlů neměl méně jak m položek.

Nejdůležitějším faktorem, který má vliv na vyváženost, je setřídění vstupních dat. Jak bylo řečeno výše, jedná se o setřídění minimálních ohraničujících obdélníků, tzn. ve výsledku máme několik způsobů, jak tyto *MBR* třídit. Nejjednodušší a zároveň použitou variantou v praktické části práce je setřídění podle x-ové souřadnice středů a rozdělení do uzlů.

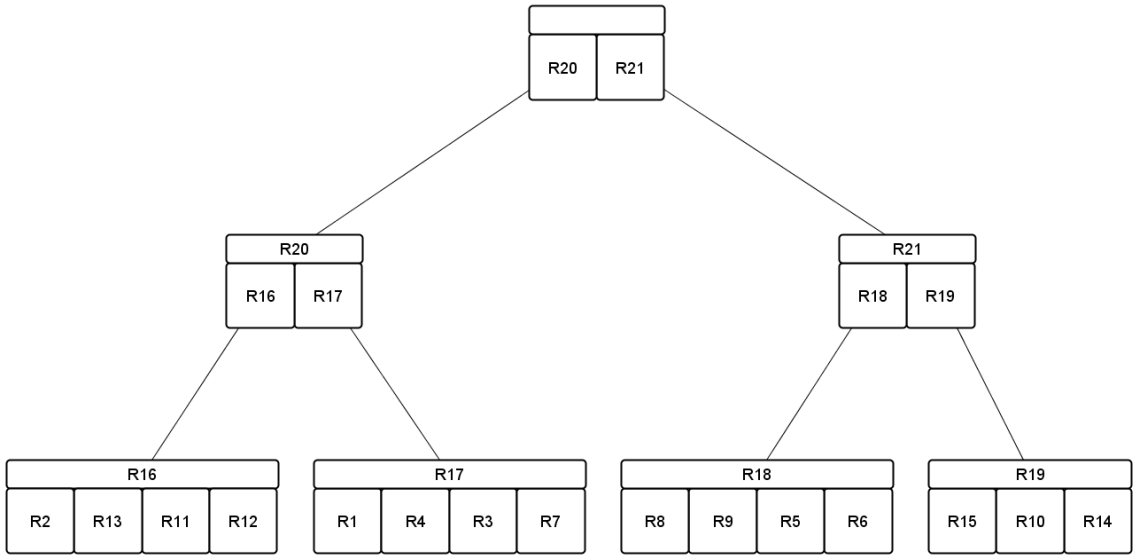
Máme-li vstupní data, jejichž *MBR* jsou graficky zobrazeny v úvodu kapitoly (Obrázek 19). Následující obrázek (Obrázek 42) znázorňuje výsledné listové uzly stromu po provedení prvního kroku, kdy jsou vstupní data seřazena dle x-ové souřadnice středu jejich *MBR* (Obrázek 41). Pro R-strom platí $M = 4$ a $m = 2$.

Obrázek 41: R-strom - setříděná vstupní data, zdroj [autor]



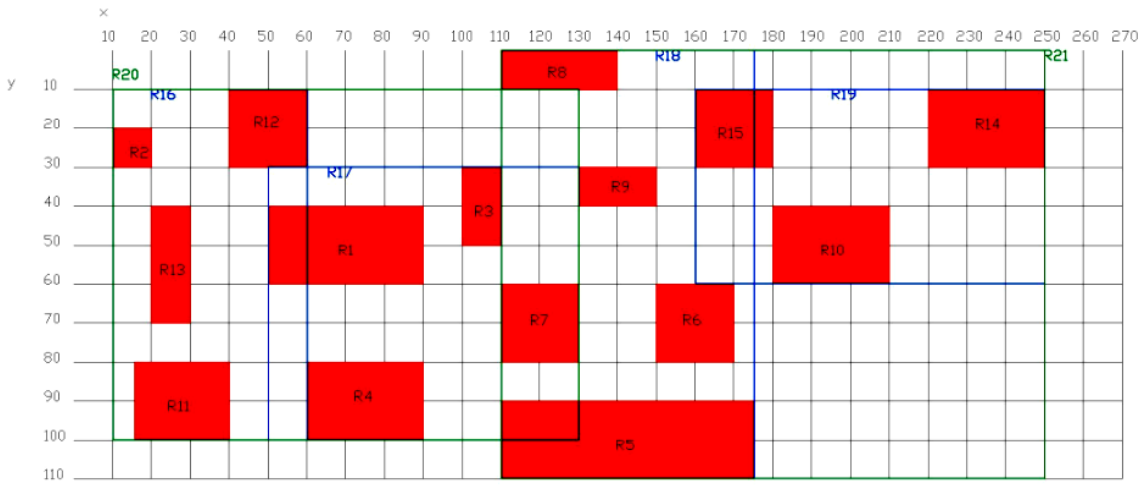
Obrázek 42: R-strom - listové uzly, zdroj [autor]

Po rekurzivním budování každé úrovně, dokud nedosáhneme kořene stromu dostáváme výsledný R-strom. Na obrázku (Obrázek 43) je vidět stejná úroveň všech listů a zároveň dokonalá vyváženost stromu.



Obrázek 43: Výsledný R-strom, zdroj [autor]

Poslední obrázek (Obrázek 44) vizualizuje výslednou dekomponovanou oblast do minimálních obdélníků.



Obrázek 44: Dělení prostoru - R-strom, zdroj [autor]

4 Implementace

Následující kapitola je věnována problematice implementace vybraných vyhledávacích algoritmů. Zejména se jedná o realizaci prioritní fronty, která ve výsledku disponuje *k*-nejbližšími sousedy. Dále jsou zde uvedeny i nutné vlastní modifikace některých algoritmů, tak aby bylo docíleno výsledku.

4.1 Vybrané algoritmy

Před zahájením samotné implementace muselo dojít k analýze aplikovatelnosti jednotlivých typů vyhledávacích algoritmů, pro každou z vybraných datových struktur. Byla vytvořena tabulka, která v levém sloupci obsahuje seznam všech algoritmů představených v teoretické části. Ke každému z nich jsou uvedeny všechny vybrané struktury a text pak určuje, zda může být algoritmus nad danou strukturou realizován či nikoli. Algoritmus, jehož buňka je označena tmavou barvou, byl u dané struktury vybrán k realizaci.

	Quad strom bod	Quad strom region	Oktalový strom	Prioritní vyhledávací strom	3-D strom	R-strom	2-D strom
Incremental best-first	ANO	ANO	ANO	ANO	ANO	ANO	ANO
Algorithms with duplicate instances							
Non-incremental best-first	**ANO**	ANO	ANO	**ANO**	**ANO**	ANO	**ANO**
Farthest neighbors	ANO	ANO	ANO	ANO	ANO	ANO	ANO
Depth-first k-nearest neighbor	**ANO**	ANO	ANO	**ANO**	**ANO**	ANO	**ANO**

Depth-first Fukugana and Narendra							
Depth-first branch and bound NN		**ANO**	**ANO**			**ANO**	
Depth-first branch and bound kNN		**ANO**	**ANO**			**ANO**	
Maxnearest best-first						**ANO**	
Maxnearest depth-first						ANO	
Approximate nearest neighbor	**ANO**	ANO	ANO	**ANO**	**ANO**	ANO	**ANO**

Tabulka 1: Implementační tabulka, zdroj [autor]

Výběr algoritmů jednotlivě u každé datové struktury probíhal s ohledem na jejich efektivitu, princip hledání a jejich závěrečné porovnání. Jedním z příkladů je realizace pouze algoritmu *non-incremental best-first* na úkor varianty *incremental best-first*. To zejména z důvodu odlišné rychlosti, kdy druhý zmíněný prochází všechny prvky struktury a hledá pouze jednoho nejbližšího souseda. V tabulce se vyskytují i zástupci, kteří nejsou implementovány vůbec a to z příčiny nemožnosti aplikace nad vybranými datovými strukturami.

4.2 Obousměrná prioritní fronta

Pro účely realizace vyhledávacích algoritmů, bylo nutné implementovat prioritní frontu, jejíž hlavním úkolem je uchovávání *k*-aktuálně nalezených nejbližších sousedů. Základním požadavkem byla proveditelnost základních operací maximálně s logaritmickou výpočetní složitostí vzhledem k *n* prvkům obsažených ve struktuře (tj. *O(logn)*).

S cílem budování minimální haldy, kde hodnota priority prvku korespondovala s jeho vzdáleností od objektu dotazu *q*, a časté operace odebírání prvku s maximální prioritou, byla zvolena implementace obousměrné prioritní fronty, konkrétně *min-max haldy*.

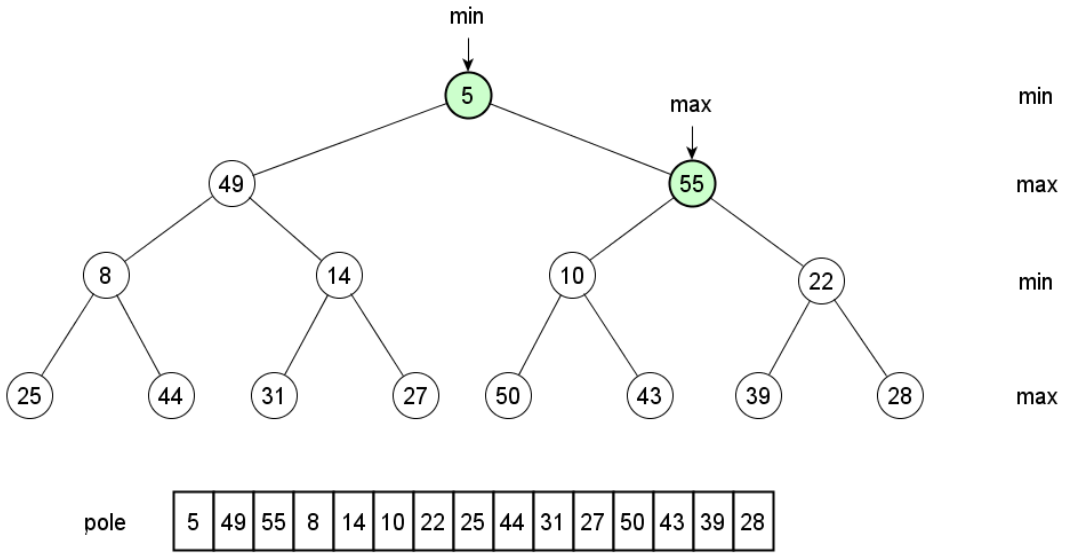
4.2.1 Základní charakteristika

Jedná se o binární strom, kde každý prvek obsahuje prioritu, která určuje jeho umístění ve struktuře. Jde o speciální typ prioritní fronty, která udržuje ukazatel na prvek s maximální i minimální prioritou. Jinými slovy operace založené na hledání minimálního respektive maximálního prvku jsou proveditelné s konstantní výpočetní složitostí *O(1)*.

Celý binární strom je rozdělen do střídajících se úrovní *min* a *max*. Pro prvek na minimální úrovni platí, že hodnota jeho priority je menší než v obou jeho podstromech. Naopak pro prvek s maximální úrovně platí, že hodnota jeho priority je vždy největší s obou přilehlých podstromů. Pokud prioritní fronta obsahuje pouze jeden prvek, pak je požadován zároveň za maximální i minimální. Existuje-li však více prvků v prioritní frontě pak platí, že kořen

binárního stromu je minimálním prvkem a maximálním prvkem se stává větší z přilehlých potomků kořene. [6]

Následující obrázek (Obrázek 45) zobrazuje prioritní frontu s ukazateli na maximální a minimální prvek. V případě, že je binární strom implementován na poli, je zobrazeno i umístění prvků v tomto poli.



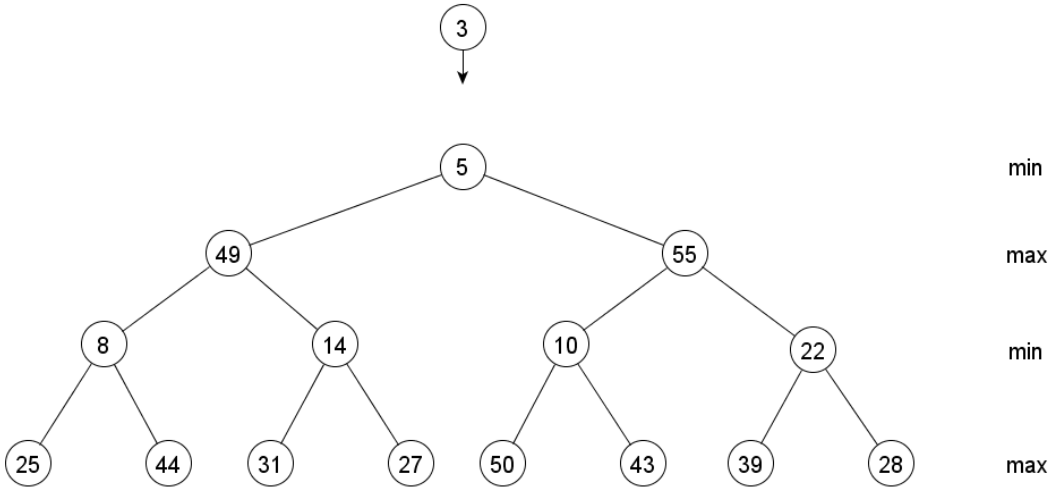
Obrázek 45: Min-max halda, zdroj [autor]

Obousměrná prioritní fronta se vyznačuje základními operacemi, jejichž princip je velice podobný klasické prioritní frontě. [6]

- vratMin() – vrací prvek s minimální prioritou
- vratMax() – vrací prvek s maximální prioritou
- vloz(x) – zařadí nový prvek *x* do prioritní fronty
- odeberMin() – odebere prvek s minimální prioritou
- odeberMax() – odebere prvek s maximální prioritou

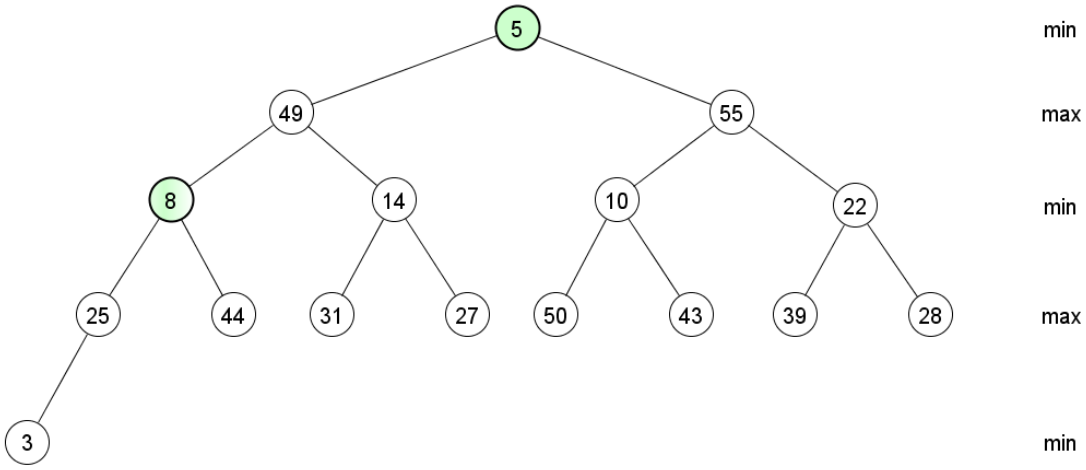
4.2.2 Operace vložení prvku

Dynamická operace vložení nového prvku do struktury se vyznačuje logaritmickou výpočetní složitostí *O(logn)*. Jejím základním krokem je umístění nového prvku do prvního volného místa zleva v poslední úrovni stromu a jeho následné „probublávání“ směrem ke kořeni. Probublávání probíhá pouze ve stejných úrovních a zároveň pro prvek v maximální úrovni musí platit, že jeho priorita musí být větší než priorita jeho praotce, aby došlo k jejich záměně. Pro prvek umístěný v minimální úrovni tento princip platí naopak, tzn. priorita nového prvku musí být menší než priorita praotce. [6]



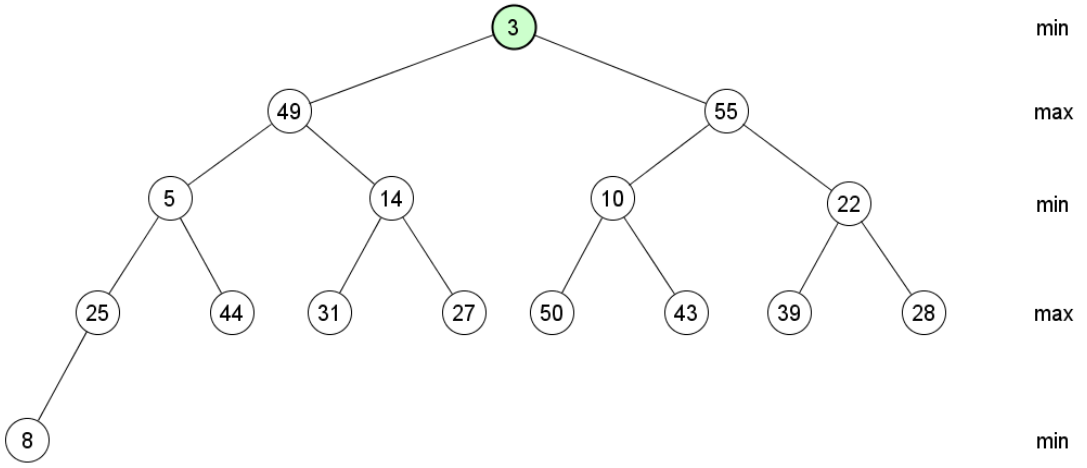
Obrázek 46: Operace vložení nového prvku do prioritní fronty, autor [zdroj]

Předešlý obrázek (Obrázek 46) znázorňuje prioritní frontu složenou z několika prvků, do které se pokoušíme vložit nový prvek s prioritou 3. Již z obrázku je patrné, že výsledkem musí být umístění nového prvku v kořeni stromu, protože jeho priorita je nejmenší ze všech prvků struktury.



Obrázek 47: Počáteční umístění nového prvku v prioritní frontě, zdroj [autor]

Jak již bylo řečeno, vkládány prvek se umístí na poslední pozici v prioritní frontě, v tomto případě se jedná o minimální úroveň (Obrázek 47). Následuje opakované porovnání s praotcem a v případě, že je jeho priorita větší dochází k jejich záměně. Tímto principem se postupně vymění prvek s prioritou 3 s prvkem 8 a 5 a ve výsledku se nový prvek stane kořenem stromu respektive minimálním prvkem (Obrázek 48).

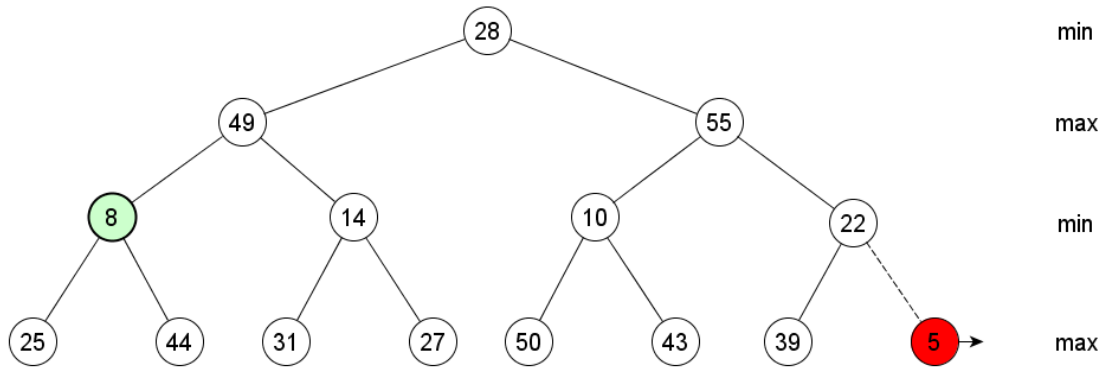


Obrázek 48: Konečná podoba prioritní fronty, zdroj [autor]

4.2.3 Operace odebrání minima a maxima

Operace odebrání maxima a odebrání minima jsou, co se týče principu totožné, proto dále popíšu pouze jednu z nich a to operaci odebrání minima. Obě operace jsou opět proveditelné s logaritmickou výpočetní složitostí.

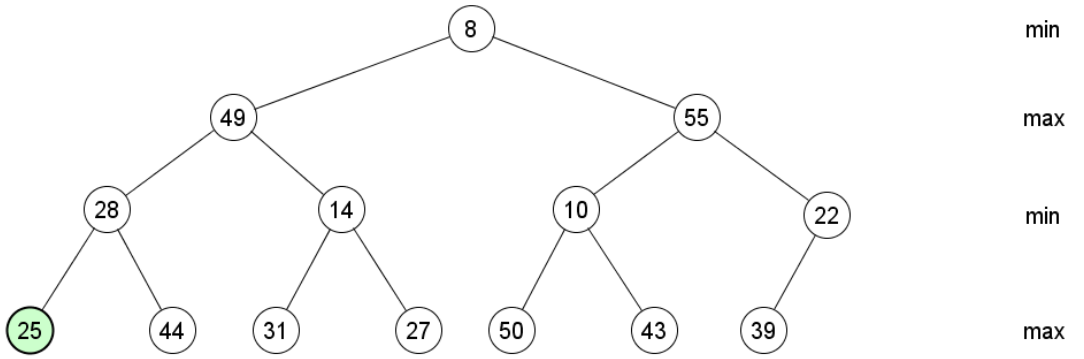
Základem je výměna minimálního prvku struktury s posledním prvkem, který nám zajistí odebrání požadovaného prvku z konce struktury a zamezí ztrátě návaznosti binárního stromu. Nejdůležitějším faktorem je poté probublávání nového kořenového prvku struktury v minimálních úrovních. Výměna probíhá s potomkem, který má nejmenší prioritu z daných potomků v nejbližší minimální úrovni a zároveň je menší než u probublávaného prvku. V konečné fázi, se musí ještě ověřit, zda potomci v nejbližší maximální úrovni nemají prioritu menší než daný probublávaný prvek. [6]



Obrázek 49: Přesun minimálního prvku v prioritní frontě, zdroj [autor]

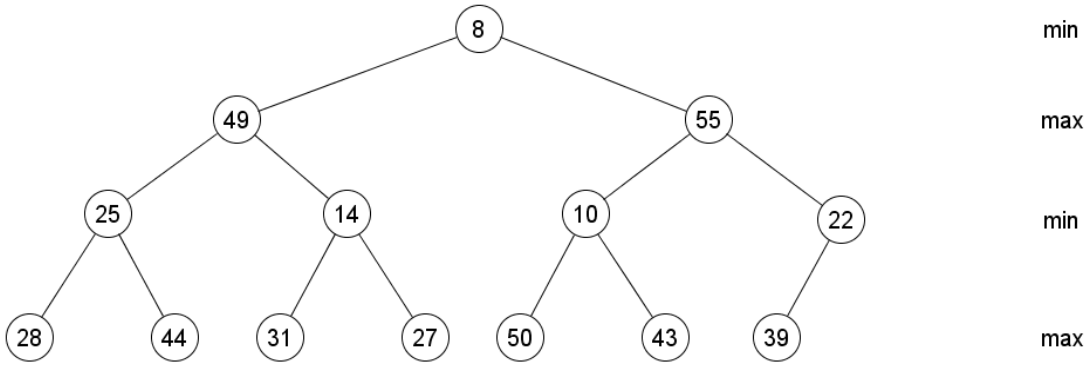
Předpokládáme-li prioritní frontu z obrázku na začátku podkapitoly (Obrázek 45) dochází k výměně koncového prvku s prioritou 28 s minimálním prvek, který je následně ze struktury odebrán (Obrázek 49). Nový kořenový prvek má v tomto případě větší prioritu než je nejmenší priorita z jeho potomků v minimální úrovni, proto dochází k jejich záměně.

Nyní se prvek s prioritou 28 nachází v poslední min úrovni, a proto se ještě provádí porovnání s nejmenší prioritou přímého potomka, tedy 25 a 44. Jelikož podle vlastnosti obousměrné prioritní fronty nemůže existovat prvek v minimální úrovni, jehož priorita je větší než priorita prvku v úrovni maximální, musí dojít k jejich záměně (Obrázek 50).



Obrázek 50: Probublání stávajícího kořenového prvku prioritní frontou, zdroj [autor]

Výslednou datovou strukturu po vykonání operace odebrání minima je zobrazena na obrázku (Obrázek 51).



Obrázek 51: Konečná podoba prioritní fronty, zdroj [autor]

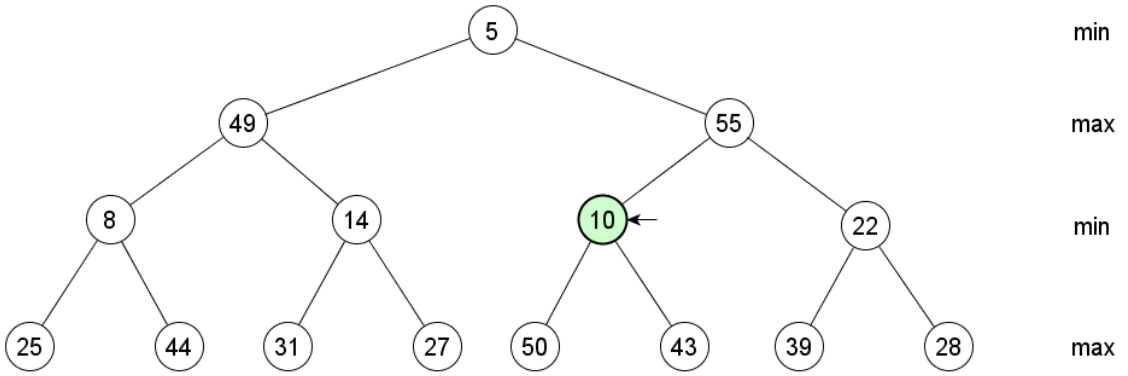
4.2.4 Operace odebrání prvku

Operace odebrání daného prvku není obsažena v základních operacích, ale pro realizaci vyhledávacího algoritmu *maxnearest best-first* muselo dojít k její implementaci. S cílem urychlení algoritmu bylo nutné provedení této operace s logaritmickou výpočetní složitostí.

Hlavním faktorem, který má vliv na provedení operace, je identifikace odebíraného prvku v poli, nad kterým je implementovaná obousměrná prioritní fronta. Z důvodu efektivnosti bylo nutné využít jiného přístupu než sekvečního hledání.

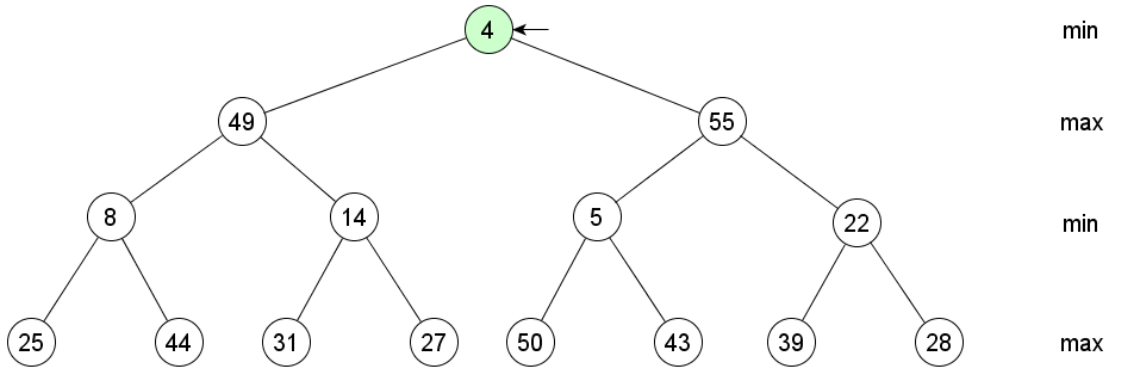
Obousměrná prioritní fronta byla rozšířena o sekundární strukturu, implementovanou v podobě hashovaní tabulky. Prvky tabulky korespondovaly s prvkami prioritní fronty a při

každé situaci, kdy se měnila poloha některého prvku v prioritní frontě muselo dojít k její aktualizaci. Hlavní příčinou výběru hashovaní tabulky je rychlost získání prvku podle klíče, která je proveditelná s konstantní výpočetní složitostí *O(1)*. [6]



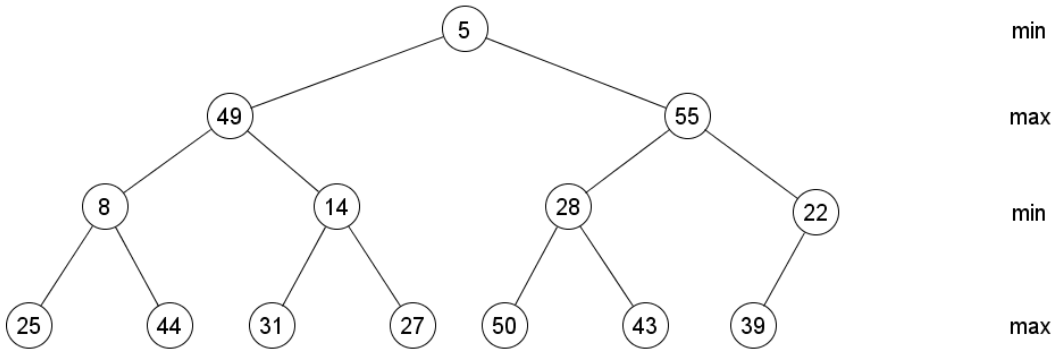
Obrázek 52: Prvek k odebrání z prioritní fronty, zdroj [autor]

Postup odebírání zadaného prvku se skládal z několika triviálních kroků. V první řadě se získala pozice daného prvku v poli z hashovaní tabulky, kde k identifikaci pozice se využila datová část prvku (Obrázek 52). Dalším krokem bylo nastavení odebíranému prvku menší prioritu než má dosavadní minimální prvek.



Obrázek 53: Nastavení nové priority a záměna s kořenem, zdroj [autor]

Následovala záměna s kořenem stromu a „probublání“ původního kořene prioritní frontou směrem nahoru (Obrázek 53). Poslední fází bylo vykonání operace odebrání minimálního prvku (Obrázek 54).



Obrázek 54: Prioritní fronta po odebrání, zdroj [autor]

4.3 Vlastní modifikace algoritmů

Základní motivací pro vlastní modifikace algoritmu *depth-first k-nearest neighbor*, ale také *non-incremental best-first* u výše zmíněných datových struktur je v neaplikovatelnosti prořezávání dle vzdálenosti D_k. V tomto případě je důvodem stavba datové struktury, kdy žádný prvek struktury nezapouzdřuje své potomky. To znamená, že pro žádný prvek nemůžeme určit, že všichni z jeho potomků nebudou blíže než je D_k. Výsledkem tedy zůstává průchod přes všechny prvky struktury a nalezení *k* nejbližších.

4.3.1 Bodový quad strom

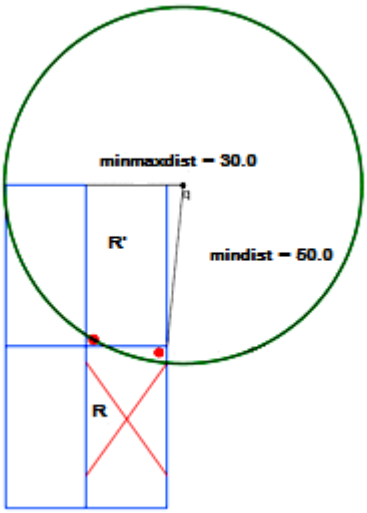
Neaplikovatelnost pravidla vede zejména k procházení všech prvků struktury a to má za následek neefektivnost výpočtu nejbližších sousedů. Bylo navrženo řešení u datové struktury bodový quad strom Každý prvek struktury si navíc uchovává region, ve kterém leží respektive, který dělí do čtyř kvadrantů (Obrázek 34).

Dle informace o regionu u každého z prvků můžeme s využitím vzdálenosti *MINDIST* zjistit, zda všechny potomci leží dále než je aktuální D_k. Využitím tohoto principu vede k omezení zbytečného zpracovávání některých větví stromu a tím i zajištění lepší efektivnosti výpočtu.

4.3.2 Regionální quad strom

Řešení problém algoritmu depth-first branch and bound

V průběhu realizace zmíněného algoritmu nad datovou strukturou regionálního quad stromu, byly nalezeny příčiny neaplikovatelnosti všech prořezávacích pravidel. Jedná se především o prořezávací pravidla využívající vzdálenosti *MINMAXDIST*.



Obrázek 55: Neaplikovatelnost pravidel H1 a H2 u regionálního quad stromu, zdroj [autor]

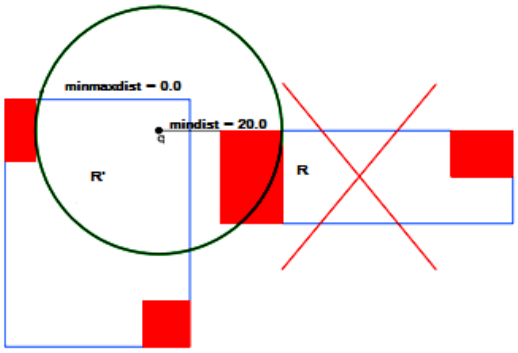
Obrázek (Obrázek 55) ukazuje chybné vyřazení jedné z větví stromu po aplikaci pravidla *H1* (podkapitola 2.9.1) i za předpokladu, že její region *R* obsahuje bližší objekt. Důsledkem je ztráta potencionálního nejbližšího souseda objektu *q*. Příčinou nevyužívání prořezávacích pravidel *H1* a *H2* je, že regiony nejsou minimálními ohraničujícími oblastmi svých objektů.

4.4 R-strom

Řešení problému algoritmu depth-first branch and bound

Při implementaci algoritmu *depth-first branch and bound*, který využívá prořezávacích pravidel *H1*, *H2* a *H3* (podkapitola 2.9), byla odhalena situace, kdy aplikace prořezávacího pravidla *H1* vedla k chybnému nalezení nejbližšího souseda.

Máme-li situaci zobrazenou níže (Obrázek 56), vede aplikace pravidla *H1* k chybnému vyřazení regionu *R*, i když jak je patrné obsahuje objekt, který je blíže od *q*, než jsou všechny objekty v *R'*. Jak bylo řečeno v definici prořezávacího pravidla (podkapitola 2.9.1). K vyřazení došlo z důsledku menší vzdálenosti *MINMAXDIST* k regionu *R'*, než je vzdálenost *MINDIST (q,R)*.



Obrázek 56: Upřesnění pravidla H1 u R-stromu, zdroj [autor]

Řešením je upřesnění prořezávacího pravidla, kdy se nebere v úvahu takový region *R'*, pro který platí:

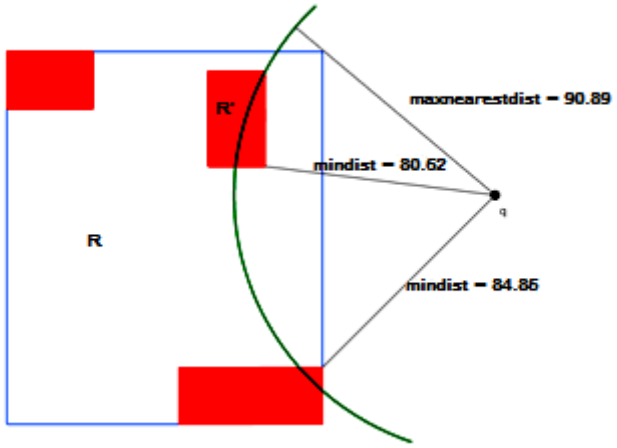
$$MINMAXDIST(q, R') = 0$$

Řešení problému algoritmu maxnearest best-first

S ohledem na princip algoritmu, který využívá vzdálenosti *MAXNEARESTDIST* přiblíženou v teoretické části pro minimální ohraničující kruh, bylo nutné navrhnout výpočet této vzdálenosti mezi minimálním ohraničujícím obdélníkem a dvourozměrným bodem. Hlavním požadavkem je uplatnění stejného postupu výpočtu.

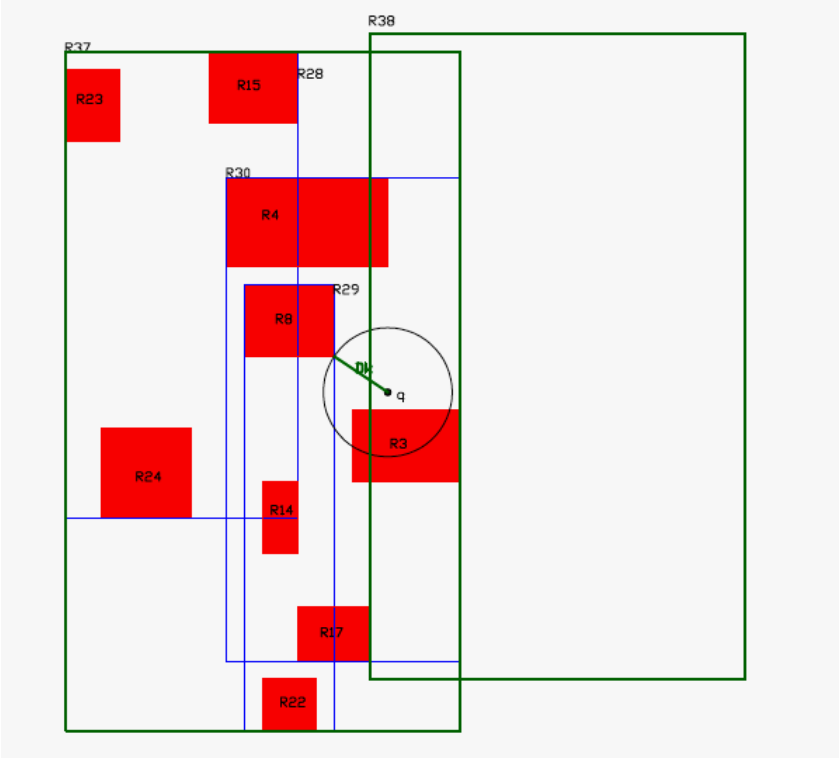
Původně navržený výpočet představuje nalezení objektu v regionu *R*, který splňuje nejkratší vzdálenost *MINDIST*, ze všech objektů. K tomuto objektu, označeného *R'*, je následně vypočítána výsledná vzdálenost *MINMAXDIST*. V příkladě zobrazeného na obrázku (Obrázek 57) pak platí:

$$MAXNEARESTDIST(q, R) = MINMAXDIST(q, R^{'})$$



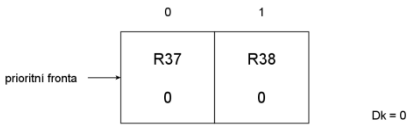
Obrázek 57: Maxnearestdist - obdélník, zdroj [autor]

V průběhu testování funkčnosti algoritmu byl odhalen problém spojený s ukládáním nelistových uzlů R-stromu do prioritní fronty *L*. Problém byl způsoben umístění dotazovaného objektu *q* do některé z *MBR*, tzn. některé nelistové uzly, zařazené v prioritní frontě *L*, mohly disponovat nulovou vzdáleností *MAXNEARESTDIST*. Následkem toho došlo k nezařazení některého z regionů, který obsahoval jeden z výsledných *k*-nejbližších sousedů.



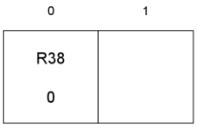
Obrázek 58: Maxnearest best-first, zdroj [autor]

Dále si popíšeme situaci zobrazenou na obrázku (Obrázek 58), která vede právě k popsanému problému. Předpokládáme-li hledání dvou nejbližších sousedů k zadanému bodu *q*, v prvních krocích algoritmu se v prioritní frontě nacházejí dva nelistové uzly *{R37, R38}* s odpovídajícími vzdálenostmi (Obrázek 59).

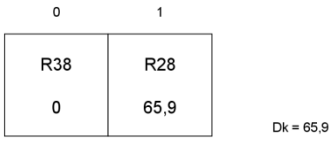


Obrázek 59: Prioritní fronta L - začátek, zdroj [autor]

V dalším kroku postupuje algoritmus ke zpracování uzlů na další úrovni stromu, v prvním případě bude přistupovat k potomkům regionu *R37*. Z důvodu, že prioritní fronta nesmí obsahovat prvky, které jsou ve vzájemném vztahu otec-potomek, dochází k odebrání *R37* a prioritní fronta nyní disponuje jedním prvkem (Obrázek 60).

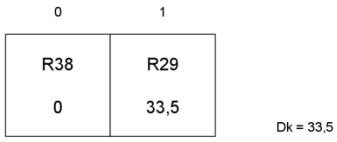


Obrázek 60: Prioritní fronta L - jeden prvek, zdroj [autor]

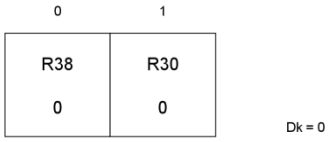


Obrázek 61: Prioritní fronta L - vložení R28, zdroj [autor]

První potomek *R28* je zařazen ihned do prioritní fronty, protože ta v danou chvíli nebyla plná (Obrázek 61). Následuje nahrazení uzlu *R28* uzlem *R29*, z důvodu jeho kratší vzdálenosti ke *q* (Obrázek 62). Následující nahrazení uzlu *R29* uzlem *R30* je z hlediska správnosti výsledku fatální, a to z důvodu, že oba uzly *R29* a *R30* obsahují dva námi hledané nejbližší sousedy, ale pouze jeden z nich je ve výsledné frontě *L* (Obrázek 63).



Obrázek 62: Prioritní fronta L - nahrazení prvku R28, zdroj [autor]



Obrázek 63: Prioritní fronta L - nahrazení prvku R29, zdroj [autor]

Řešením tedy je, zařazovat do prioritní fronty jen takové nelistové uzly R-stromu, které nemají nulovou vzdálenost *MAXNEARESTDIST*.

5 Vzorové aplikace

Obsahem kapitoly je vysvětlení základní koncepce softwarových aplikací pro demonstraci procesů hledání nejbližších sousedů nad různými datovými strukturami.

5.1 Základní popis

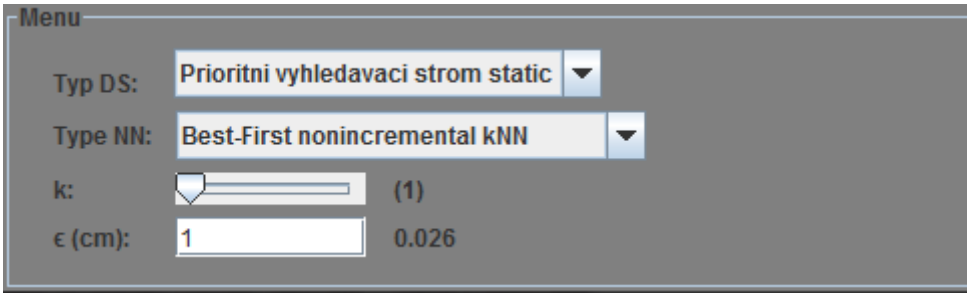
K vytvoření vzorových aplikací byl využit programovací jazyk *Java*, konkrétně pak vývojové prostředí *Netbeans*. Spouštění těchto aplikací, je možné pouze pokud je v počítači k dispozici *Java Virtual Machine*. Jinými slovy musí být nainstalováno *Java JDK* a také *Java 3d*.

5.2 Uživatelské prostředí

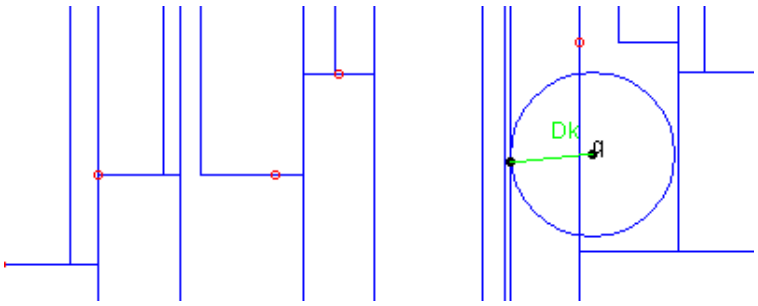
Uživatelské rozhraní je rozděleno do několika základních částí, které se nepatrně liší pouze u datových struktur operujících v trojrozměrném prostoru.

Horní menu (Obrázek 64) obsahuje položky pro načítání a ukládání souborů s daty, které jsou uchovávány v datové struktuře. Ve spodní části aplikace (Obrázek 66) jsou k dispozici nastavení, týkající se nejen typu vybudování, ale také vyhledávajícího algoritmu. Může zde být specifikován typ algoritmu, počet nejbližších sousedů a popřípadě i chybová tolerance. V prostřední části (Obrázek 65) se pak nachází grafický výsledek zachycující nejen dekomponovaný prostor, minimální vyhledávací region, ale také *k*-nejbližších sousedů.

Obrázek 64: Uživatelské rozhraní - menu, zdroj [autor]

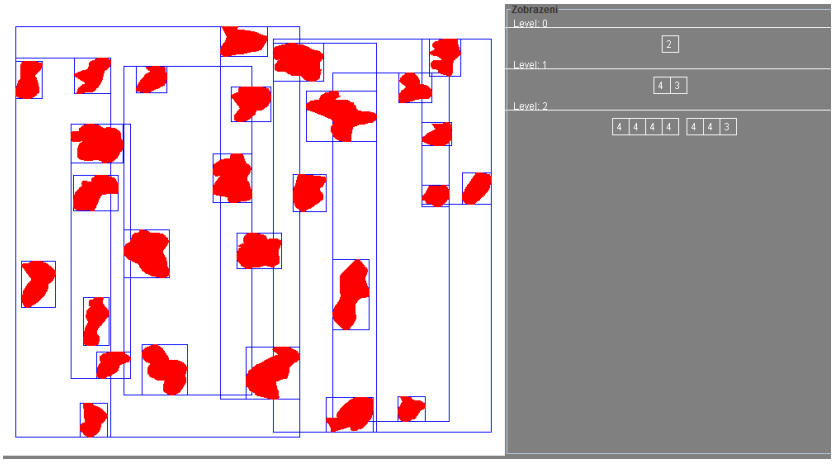


Obrázek 65: Uživatelské rozhraní - nastavení, zdroj [autor]



Obrázek 66: Uživatelské rozhraní - zobrazení, zdroj [autor]

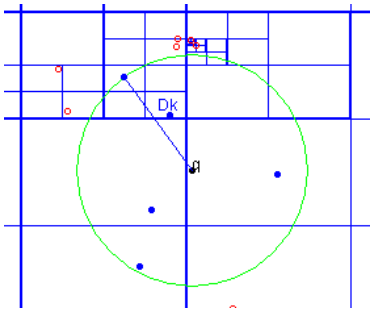
V případě aplikace R-stromu obsahuje uživatelské rozhraní navíc ještě zobrazení stavby stromu v pravé části, tak jak je uvedeno na obrázku (Obrázek 67).



Obrázek 67: Uživatelské rozhraní R-strom, zdroj [autor]

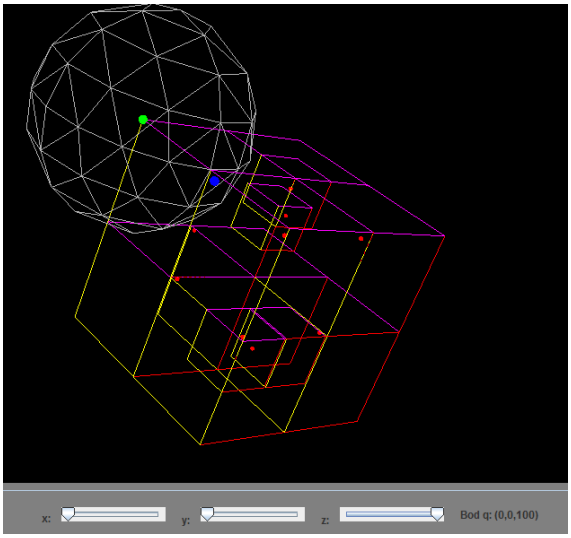
5.3 Funkce a zobrazení

Nejdůležitější částí každé aplikace je oblast obsahující grafický výsledek. Nad touto oblastí lze pomocí myši provádět několik základních operací. Jednou z nich je specifikování polohy bodu *q* a to pravým tlačítkem myši (Obrázek 68). V případě, že aplikace obsahuje i možnost dynamického přidávání prvků do struktury, lze levým tlačítkem myši vytvořit nový prvek, který je určen polohou myši.



Obrázek 68: Grafická reprezentace vyhledávajícího regionu, zdroj [autor]

Odlišností jsou aplikace demonstrující datové struktury uchovávající trojrozměrná data, kde tyto operace v grafické části nejsou k dispozici. K určení polohy bodu *q*, jsou pak uvedeny posuvníky pro každou z dimenzí (Obrázek 69).



Obrázek 69: Definice bodu q u oktalového stromu, zdroj [autor]

6 Testování

Kapitola se zaměřuje především na prezentaci výsledků testování implementovaných algoritmů. Dosažené výsledky pak dále slouží k porovnání a doporučení z hlediska jejich aplikace.

6.1 Testovací data

Jedním z cílů diplomové práce bylo otestování vyhledávacích algoritmů nad vybranými datovými strukturami, které byly konstruovány reálnými daty popisující vybrané objekty na území České republiky. Podíváme-li se na rozdílnou povahu některých datových struktur, bylo zapotřebí i odpovídajících testovacích dat.

Pro posuzování vyhledávacích algoritmů nad dvourozměrnými bodovými datovými strukturami, bylo využito reálných geografických poloh obcí České republiky (Obrázek 70a), jejichž počet se blížil 20ti tisícům. Jednalo se o regionální i bodový quad strom, o 2-D strom a v poslední řadě i o prioritní vyhledávací strom.

Obrázek 70: Reálná testovací data, zdroj [autor]

Při porovnání algoritmů *depth-first branch and bound* a *maxnearest best-first* nad datovou strukturou R-strom musela být využita data jiná, než bodového charakteru. V tomto případě se R-strom budoval z reálných dat silnic prvních a druhých tříd v České republice (Obrázek 70b). Datový soubor byl sestavený z 30 000 položek.

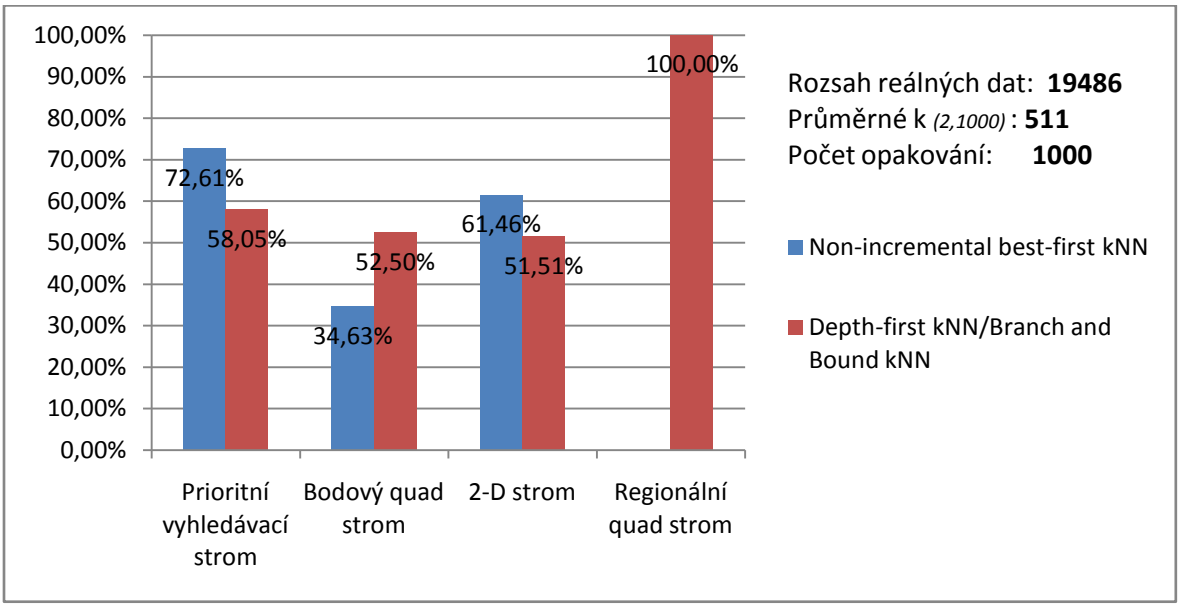
V posledním případě a to testování oktalového a 3-D stromu, které zpracovávají trojrozměrný prostor, nebylo možné najít odpovídající reálná data takového rozsahu. Z tohoto důvodu se testování provádělo pouze nad předem vygenerovanými daty.

6.2 Testy a dosažené výsledky

Testování proběhlo pro všechny vybrané vyhledávající algoritmy, tak jak bylo uvedeno v tabulce výše (Tabulka 1). Bylo provedeno pět testů, které jsou zaměřeny na porovnání dosažených výsledků. Každý test je doplněn o graf, který procentuálně znázorňuje dosažený čas výpočtu. Algoritmus, který trval nejdéle, automaticky obdržel 100% a každý z ostatních dílčích výsledků je vyjádřen v procentech vůči němu. Výsledný čas je průměrem z výsledků algoritmu provedeného tisíckrát s měnící se polohou dotazovaného bodu *q*.

6.2.1 Test I.

Test s pořadovým číslem jedna je určen k porovnání dvou odlišných přístupů procházení stromové hierarchie. Testují se zejména algoritmy *non-incremental best-first* a *depth-first k-nearest neighbor* respektive u regionálního quad stromu pouze zástupce *branch and bound kNN*. Testování jsou podrobeny pouze bodové dvourozměrné datové struktury. Každý z algoritmů je tisíckrát opakován nad stejnými daty, ale pro různý počet nejbližších sousedů.

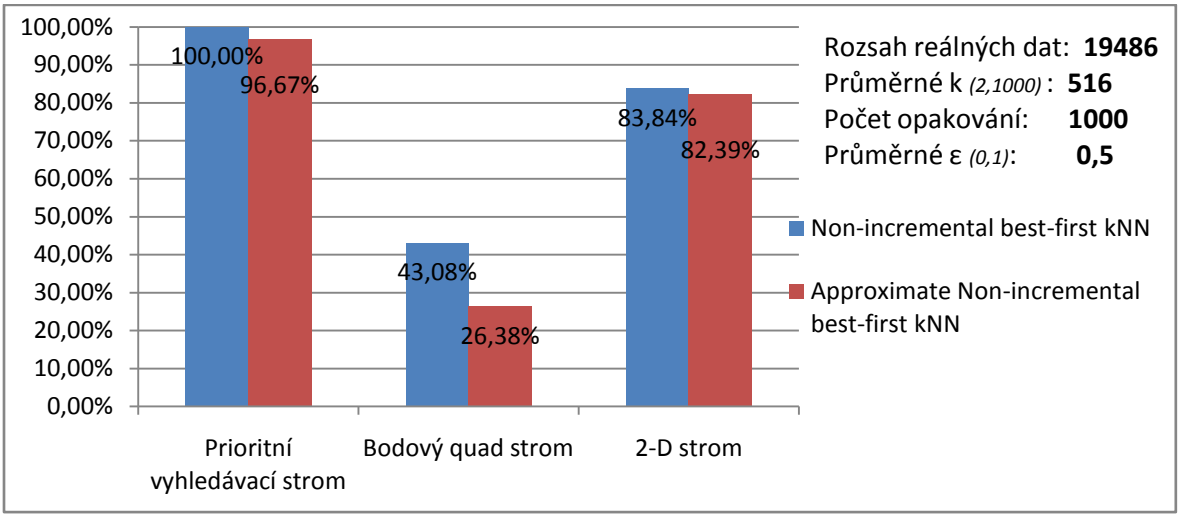


Obrázek 71: Výsledek testu I, zdroj [autor]

Z výsledného grafu (Obrázek 71) lze určit několik závěrů. Metoda hledání tzv. do hloubky je pro většinu datových struktur efektivnější, výjimkou se stává bodový quad strom u kterého vyšel lépe zástupce algoritmu *best-first*. Je to způsobeno především nevyváženosti struktury, jinými slovy různou úrovní umístění každého z listů. Nejhorším zástupcem se pak stává regionální quad strom, u kterého je neefektivita algoritmu způsobena umístěním dat pouze v listech stromu a to má za následek jeho velkou výšku. Na druhé straně graf poukazuje, že modifikace algoritmu v případě bodového quad stromu (podkapitola 4.3.1) dosahuje lepších výsledků, než u struktur nad kterými daná modifikace neproběhla.

6.2.2 Test II.

Test slouží k potvrzení teoretického tvrzení o aproximačních algoritmech, které by měly disponovat lepším časem výpočtu a to na úkor přesnosti nalezených sousedů. Porovnání klasické varianty s aproximační proběhlo u třech zástupců vybraných datových struktur a to s měnící se hodnotou chybového faktoru.

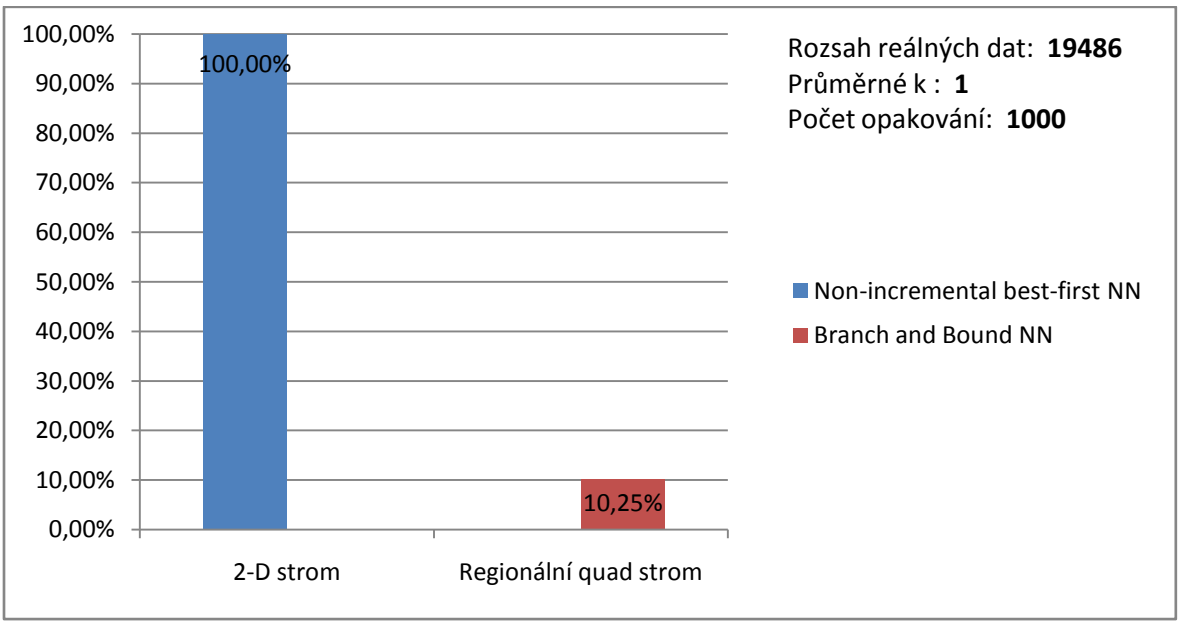


Obrázek 72: Výsledek testu II, zdroj [autor]

Graf dává bezpochyby zapravdu teoretickému základu o aproximačních algoritmech, protože v každém z dílčích porovnání s klasickým algoritmem, dosahuje lepších výsledků. I v tomto případě se projevuje navrhnuté řešení úpravy algoritmu u bodového quad stromu. Výsledek jednoznačně prokazuje, že návrh způsobí získání výsledku minimálně o polovinu rychleji (Obrázek 72).

6.2.3 Test III.

Jedná se o otestování časů výsledků algoritmů hledajících pouze jednoho nejbližšího souseda a to ve velkém počtu reálných dat. Byl vybrán jeden ze zástupců varianty *non-incremental best-first*, v našem případě 2-D strom, a jediný zástupce algoritmu *branch and bound NN*.

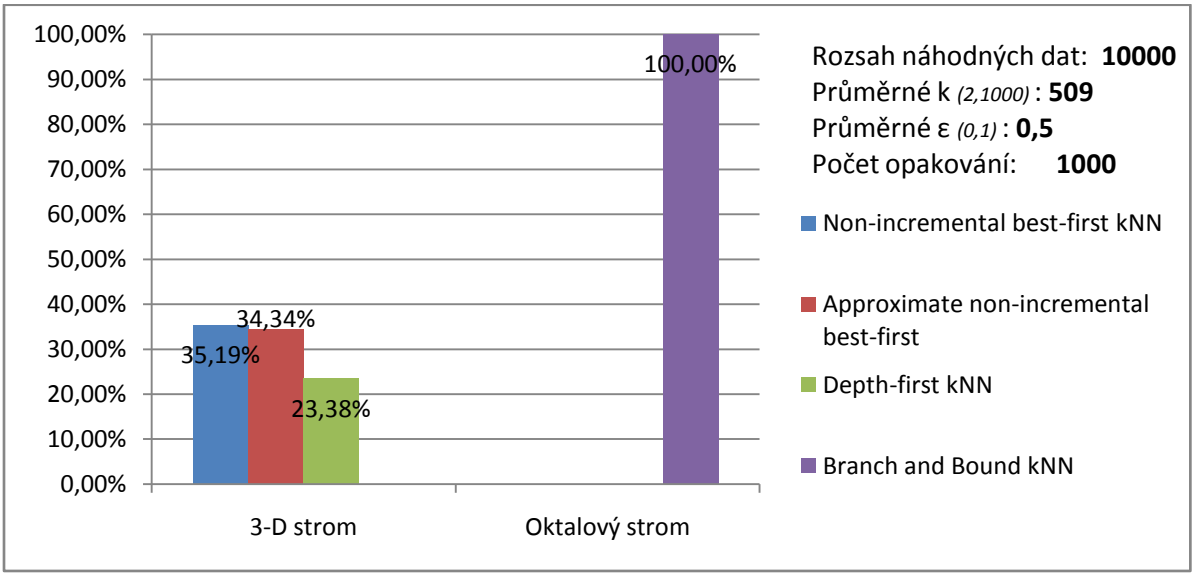


Obrázek 73: Výsledek testu III, zdroj [autor]

Po skončení testu byl sledován průměrný čas potřebný pro nalezení jednoho nejbližšího souseda. Grafický výsledek znázorňuje (Obrázek 73) výhody algoritmu *branch and bound NN* zejména v jeho schopnosti rozpoznat, zda má být daná větev stromu dále zpracovávána.

6.2.4 Test IV.

Jak již bylo řečeno, testování datových struktur uchovávajících trojrozměrná data, probíhal nad náhodně vygenerovanými daty. Účelem testu bylo porovnat všechny vyhledávací algoritmy realizované nad datovou strukturou 3-D strom, s pouze jedním zástupcem u oktalového stromu.

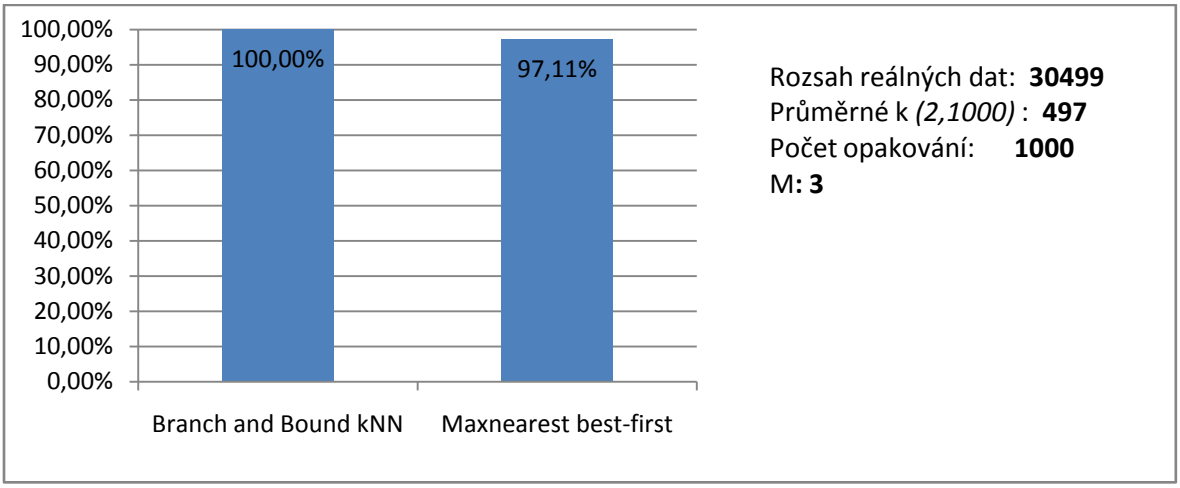


Obrázek 74: Výsledek testu IV, zdroj [autor]

Na obrázku (Obrázek 74) je k vidění výsledek algoritmu *branch and bound kNN*, který ve srovnání s ostatními testovanými algoritmy, vyšel nejhůře a to o více jak polovinu. Příčinou zůstává využití algoritmu založeného na prohlídce do hloubky nad datovou strukturou, která je charakterizována především svojí výškovou nevyvážeností. Zajímavým výsledkem zůstává čas algoritmu *depth-first k-nearest neighbor*, který je o poznání rychlejší než aproximační varianta *non-incremental best-first*.

6.2.5 Test V.

Test je zaměřený pouze na algoritmy aplikované nad datovou strukturou R-strom. Ta byla vybudována z reálných dat území České republiky čítající 30 499 silnic, a pro každý z uzlů R-stromu platila hodnota $M = 3$ (řád stromu). Proběhlo několik opětovných hledání s měnícím se dotazovaným bodem a počtem hledaných sousedů. V grafu (Obrázek 75) je zanesen průměr z těchto dílčích výsledků.



Obrázek 75: Výsledek testu V, zdroj [autor]

I když algoritmus *maxnearest best-first* vyšel ve výsledku lépe, je jeho náskok pouze v rámci jednotek procent. Tento závěr může být způsoben nejen hustým rozložením dat, kdy se dotazovaný bod mohl nacházet uvnitř vícero oblastí, ale také ve výsledku špatným návrhem výpočtu vzdálenosti *MAXNEARESTDIST*.

6.3 Zhodnocení

Ze všech dosažených výsledků lze doporučit použití algoritmu *depth-first* u datových struktur, které disponují podobnou hloubkou každého z listů a neobsahují prázdné prvky, které slouží pouze pro směrování od kořene k listům. Pokud je to naopak, nejvhodnější volbou pak zůstává algoritmus založený na prohlídce *best-first*. Z hlediska vylepšení efektivity obou typů algoritmů byl ověřen návrh, kdy si každý prvek struktury uchovává region, který dále dělí. To vede ke zpracování jen části datové struktury a tím i rychlejší obdržení výsledku.

Aproximační hledání nejbližších sousedů se vyplatí využít v případě, kdy očekáváme výsledek v konkrétním čase a nezáleží nám na vysoké míře přesnosti. Tato varianta vede dokonce k urychlení v řádu procent.

Pokud se zaměříme na testování datových struktur uchovávajících trojrozměrná data, konkrétně na 3-D strom a oktalový strom, není možné učinit obecný závěr a to zejména, protože test probíhal nad náhodnými daty. Pro reálná data se ale dá předpokládat podobné srovnání, jako u 2-D stromu a regionálního quad stromu.

Pokud bychom se zaměřili na doporučení algoritmu u datové struktury R-strom, pak v případě jejich porovnání, bylo dosaženo lepší rychlosti u algoritmu *maxnearest best-first*, ale jen v rámci jednotek procent. Jejich hlavní rozdíl pak zůstává v implementaci, která je u *maxnearest best-first* mnohem náročnější především s ohledem na prioritní frontu.

7 Závěr

Cílem práce bylo charakterizovat vyhledávací algoritmy i s jejich případnými variacemi, následná implementace těchto algoritmů v datových strukturách uchovávajících jak geografická, tak i prostorová data a formulovat doporučení pro použití na základě provedených testů.

Pro splnění cílů bylo zapotřebí několika datových struktur, které se lišily především dekomponovaným prostorem a charakterem ukládaných dat. Byla popsána jejich základní charakteristika a princip statického vybudování na velmi malém rozsahu dat. Dále byla, pro úspěšnou implementaci, navržena řešení pro algoritmy, které za určitých podmínek nedokázaly najít správný výsledek.

Z hlediska přiblížení principu hledání nejbližších sousedů u různých algoritmů byly vytvořeny vzorové aplikace pro každou z vybraných datových struktur.

V testovací části bylo uvedeno několik testů, které sloužily k posouzení algoritmů a doporučení ohledně jejich aplikace u daných struktur. Testování probíhalo nad velkým rozsahem reálných dat, výjimkou jsou pouze struktury uchovávající trojrozměrná data.

8 Použitá literatura

[1] SAMET, Hanan. *Foundations of multidimensional and metric data structures*. San Francisco : Morgan Kaufmann, 2006. 1024 s. ISBN 978-0-12-369446-1.

[2] LU, Hua. *Spatial Databases II: Spatial Queries* [online]. Aalborg University, Spring 2011. Prezentation. Department of Computer Science. Dostupné z WWW: <http:// https://intranet.cs.aau.dk/fileadmin/user_upload/Education/Courses/2011/DBS/17.Spatial_DB_3.pdf >.

[3] MANINDRA, Agrawal; DING-ZHU Du; ZHENHUA Duan; ANGSHENG Li. *Theory and Applications of Models of Computation*, 5th International Conference, TAMC 2008, Xi'an, China, April 25-29, 2008. Proceedings. Lecture Notes in Computer Science 4978 Springer 2008, ISBN 978-3-540-79227-7.

[4] MEI-KANG, Wu. *Evaluation of R-trees for Nearest Neighbor Search*. The Faculty of the Department of Computer Science. University of Houston.

[5] SAMET, Hanan. *K-Nearest Neighbor Finding Using MaxNearestDist*. Computer Science, Department Center for Automation Research, Institute for Advanced Computer Studies. University of Maryland, 2007, 11 s.

[6] MEHTA, Dinesh P.; SAHNI, Sartaj. *Handbook of Data Structures and Applications*. London : Chapman & Hall, 2004. 1391 s. ISBN 1-58488-435-5.

[7] LEUTENEGGER, Scott T.; Mario A. LÓPEZ. *STR: a simple and efficient algorithm for R-tree packing*. Proceedings. 13th International Conference. Denver University, USA, 1997, 13 s. Dostupné z WWW: <http://www.cise.ufl.edu/class/cis4930fa07ilg/slides/str.ppt>.

[8] POKORNÝ, Jaroslav. *Prostorové datové struktury a jejich použití k indexaci prostorových objektů* [online]. [2009] [cit. 2013-03-29]. Dostupný z WWW: <http://gis.vsb.cz/GIS_Ostrava/GIS_Ova_2000/Sbornik/Pokorny/Referat.html>.

[9] GUTTMAN, Antonin. R-*trees: A Dynamic Index Structure for Spatial Searching*. Proceedings of the ACM SIGMOD, Boston, MA, June 1984.

[10] *Astrology Pacific* [online]. 2011 [cit. 2013-03-29]. Databáze měst. Dostupné z WWW: <http://astrolot.cz/city/!city.html>.

Příloha A – CD se vzorovými aplikacemi

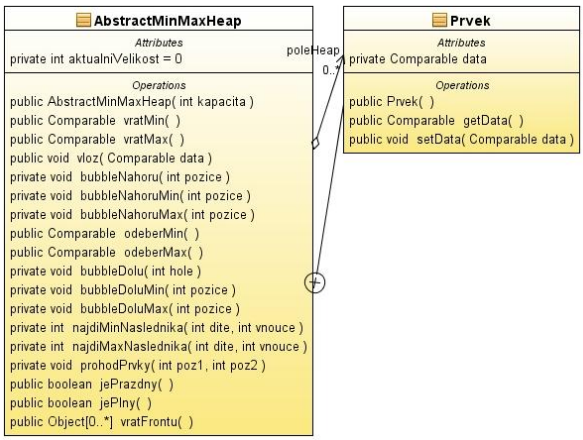
K této práci je přiložené CD, na kterém jsou k dispozici funkční vzorové aplikace i jejich zdrojový kód spustitelný v programu *Netbeans*. U jednotlivých aplikací jsou umístěny i textové soubory se vstupními daty.

Příloha B – programátorská dokumentace

Příloha obsahuje nejenom diagramy tříd vybraných datových struktur, ale i jejich bližší popis. Každá struktura má minimálně implementovanou metodu sloužící k jejímu vybudování. Jedná se o metodu *Vybuduj* případně o metodu *Vloz*. Vyhledávací algoritmy jsou realizovány ve formě metod, pojmenovaných dle názvu algoritmu, umístěné v hlavní třídě datové struktury.

Obousměrná prioritní fronta

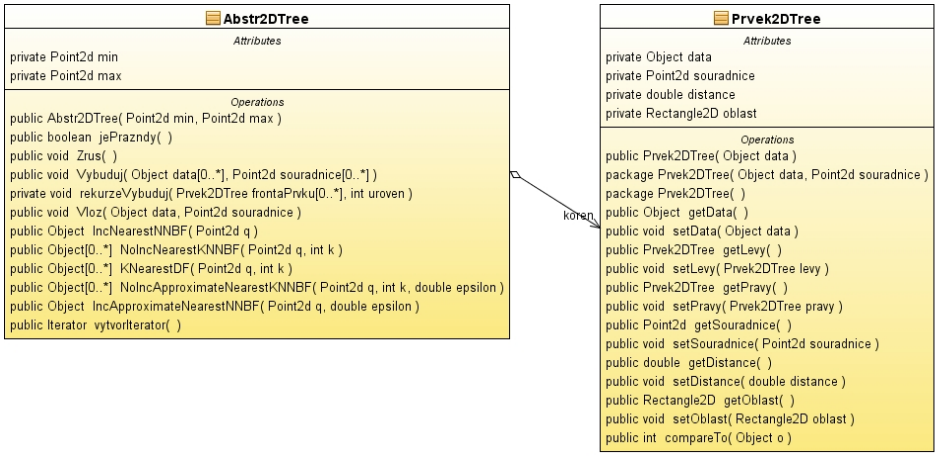
Třída *AbstractMinMaxHeap* a *Prvek* odrážejí funkci obousměrné prioritní fronty. Tato varianta je využívána u všech datových struktur vyjímaje R-stromu. Prioritní fronta je implementována na poli, jemuž je určena velikost v konstruktoru třídy. Třída navíc implementuje rozhraní *Comparable*, které slouží k porovnávání prvků a tím i jejich správné umístění ve struktuře.



Třídy prioritní fronty, zdroj [autor]

2-D strom

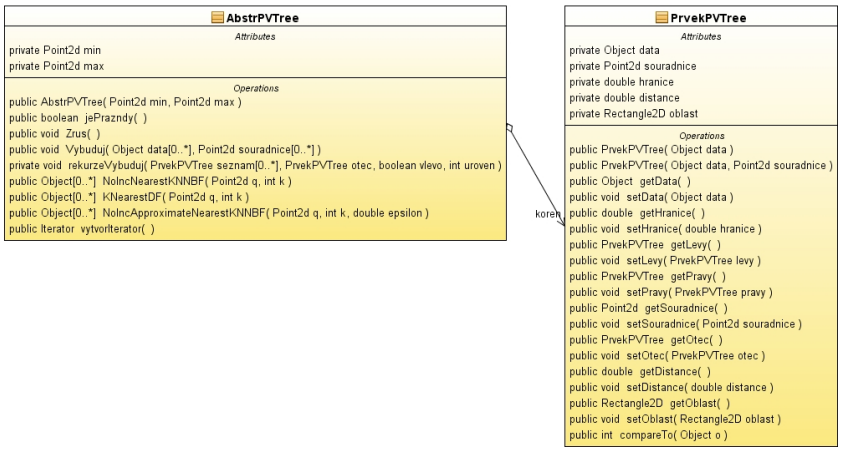
2-D strom je implementován třídou *Abstr2DTree* a třídou *Prvek2DTree*, která charakterizuje prvky, se kterými daný strom pracuje. Na obrázku je patrné, že třída *Prvek2DTree* obsahuje zejména ukazatel na levého a pravého potomka což demonstruje, že se skutečně jedná o binární strom. Mezi další důležité atributy prvku patří oblast, kterou daný prvek dělí, souřadnice podle který traverzuje stromovou hierarchií při vložení a v poslední řadě vzdálenost k zadanému bodu dotazu. Tato vzdálenost slouží později, jako priorita v obousměrné prioritní frontě.



Třídy 2-D strom, zdroj [autor]

Prioritní vyhledávací strom

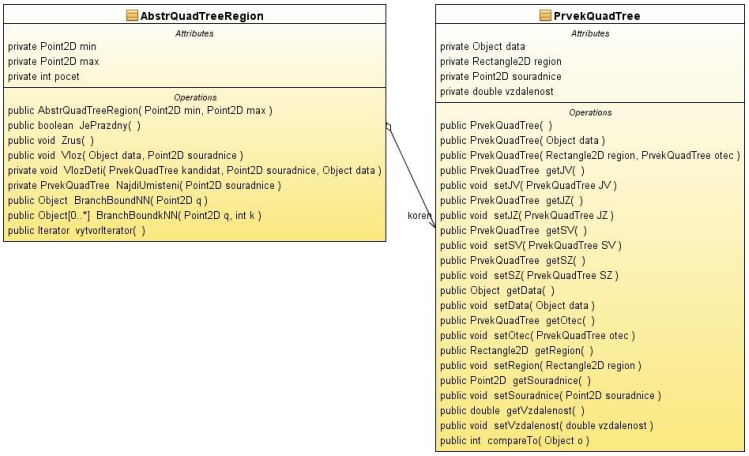
Prioritní vyhledávací strom je svojí strukturou tříd podobný předchozímu 2-D stromu, pouze třída *PrvekPVTree* odrážející prvek struktury, navíc obsahuje atribut *hranice*. Znovu se jedná o binární strom, a tudíž si prvek uchovává mimo jiného ukazatele na levého a pravého potomka.



Třídy prioritní vyhledávací strom, zdroj [autor]

Regionální quad strom

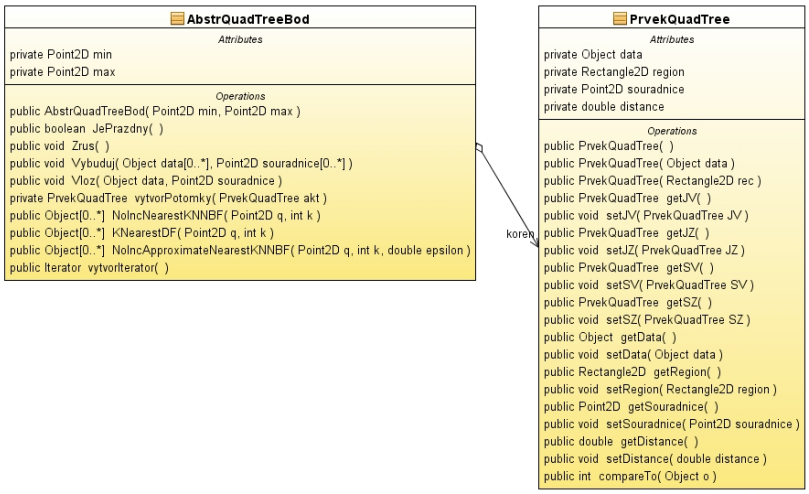
Regionální quad strom je realizován třídami *AbstrQuadTreeRegion* a *PrvekQuadTree*. Prvek obsahuje ukazatel na čtyři své potomky, které představují jednotlivé kvadranty. Použití zbylých atributů je ze stejného principu, jako u předchozích struktur. Třída obsahuje pouze dynamickou metodu *Vloz*, z důvodu popsaného v teoretické části tzn. nemožnost vybudování vyváženého stromu.



Třídy regionální quad strom, zdroj [autor]

Bodový quad strom

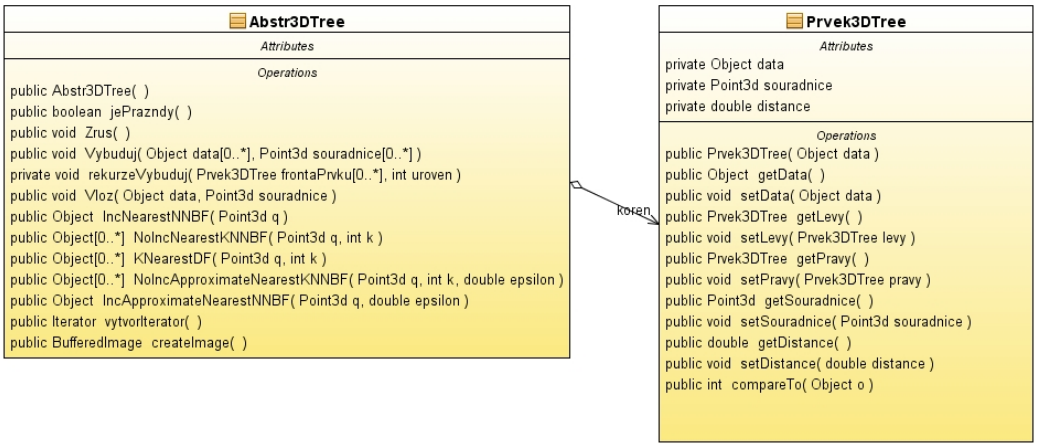
Datová struktura využívající téměř totožné třídy jako varianta regionální, jen obsahuje navíc metodu statického vybudování. Princip rozdílné stavby u obou struktur pak spočívá pouze v rozdílné implementaci metody *Vloz* respektive *Vybuduj*.



Třídy bodový quad strom, zdroj [autor]

3-D strom

3-D strom je totožná datová struktura jako 2-D strom, jen pracuje v trojrozměrném prostoru, z tohoto důvodu jsou u obou zmíněných struktur realizovány stejné třídy. Liší se pouze v metodě vybudování, která oproti 2-D stromu postupně dělí podle tří dimenzí.



Třídy 3-D strom, zdroj [autor]

Oktalový strom

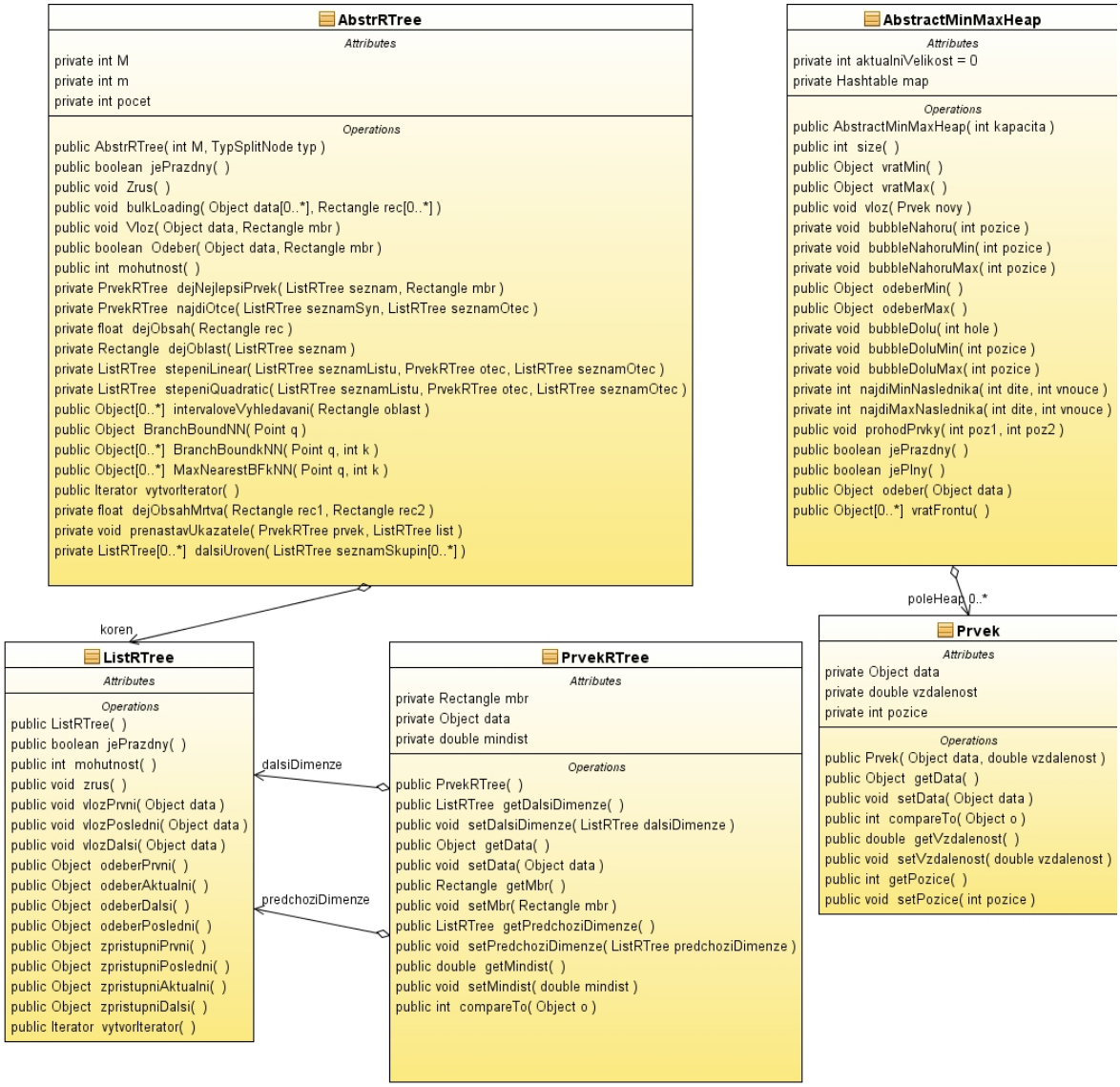
Poslední zástupce trojrozměrných datových struktur je oktalový strom implementován třídami *AbstrOctalTree* a *PrvekOctalTree*. Každý prvek obsahuje ukazatel na osm svých potomků, tzv. oktantů. Mimo jiné obsahuje také zapouzdřující region, který je datového typu *Box3D*. Jedná se o třídu využívající dvourozměrný obdélník rozšířený o třetí dimenzi tzn. reprezentuje trojrozměrnou krychli.

Třídy oktalový strom, zdroj [autor]

R-strom

Datová struktura R-strom je realizována v několika třídách. Třída *PrvekRTree* odráží jednotlivé prvky struktury, které jsou ukládány v uzlech. Uzel je representován třídou *ListRTree* a maximání počet prvků v každém uzlu, je dán hodnotou atributu *M*. Ten je předán v parametrickém konstruktoru spolu s typem dynamického štěpení. Jak ukazuje obrázek datová struktura disponuje, jak metodou statického vybudování *bulkLoading*, tak i dynamickými metodami *Vloz* a *Odeber*.

Prioritní fronta využívaná pro vyhledávací algoritmy je rozšířena nejen o sekundární strukturu *Hashtable*, definovanou jako atribut třídy, ale i o metodu *Odeber*.



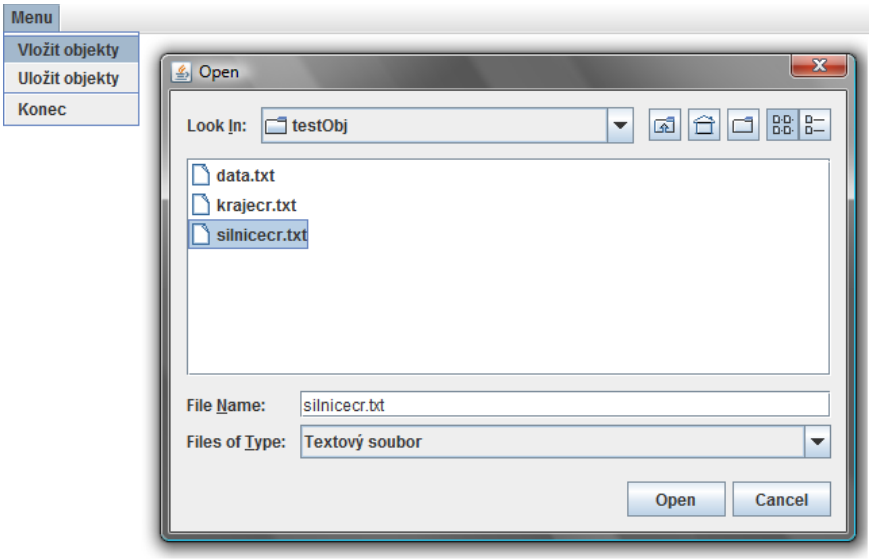
Třídy R-strom, zdroj [autor]

Příloha C – uživatelská dokumentace

Příloha popisuje všechny možnosti ovládání ve vytvořených podpůrných aplikacích, které slouží především pro demonstraci principů implementovaných algoritmů.

Vložení souboru s daty

Pro načtení souboru s daty slouží položka *Menu*. Po kliknutí na položku *Vložit objekty* se zobrazí formulář pro výběr daného souboru. Po načtení nových dat dojde automaticky k vybudování dané datové struktury a tím i k jejímu korektnímu zobrazení.



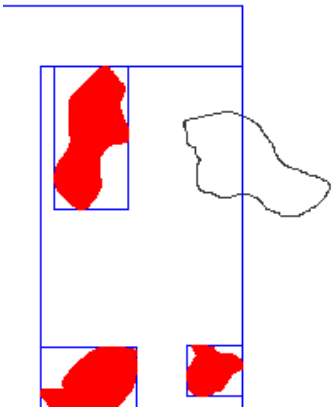
Soubor s daty, zdroj [autor]

Uložení souboru s daty

Analogií pro načtení souboru s daty je jejich uložení. Slouží k tomu položka *Uložit objekty*. Uživateli je opět zobrazen formulář, tentokrát pro definování jména souboru a jeho umístění. V případě ukládání se ve výsledném souboru objeví všechny objekty, ze kterých je daná datová struktura vybudována.

Vložení nového objektu do struktury

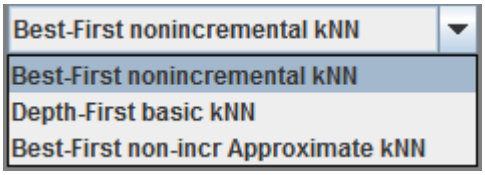
Většina aplikací umožňuje i interaktivní vkládání nového objektu a to pouhým stlačením případně tažením levého tlačítka myši v panelu vyzualizace datové struktury. Například u datové struktury R-strom můžeme navíc definovat, zda chceme nový objekt vkládat dynamicky či nikoli.



Příklad vkládání nového objektu do R-stromu, zdroj [autor]

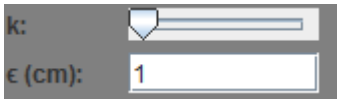
Konfigurace typu vyhledávání

Nejdůležitější částí každé aplikace je možnost výběru vyhledávacího algoritmu. Slouží k tomu komponenta *combobox* ve spodní části aplikace. Komponenta obsahuje všechny algoritmy, které byly u dané struktury implementovány.



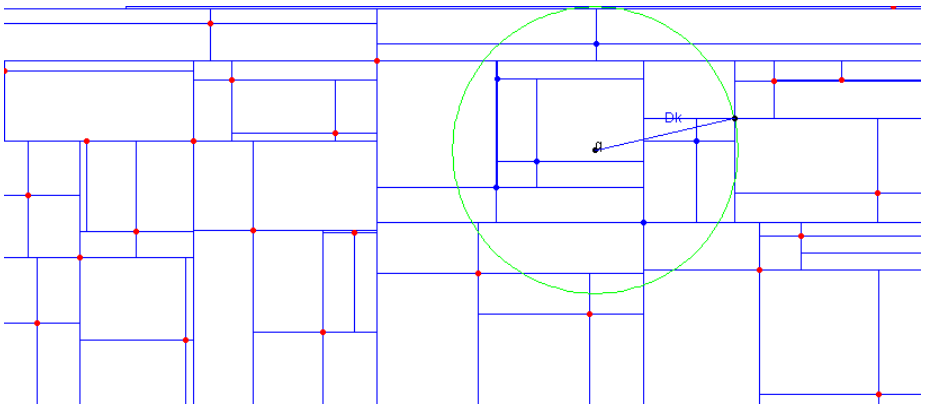
Výběr algoritmu, zdroj [autor]

U algoritmů, které hledají více nejbližších sousedů, lze definovat jejich počet. Posledním nastavením je chybový faktor ε, který se zadává do textové komponenty a to v centimetrech.



Dodatečná nastavení, zdroj [autor]

Posledním krokem k zobrazení výsledku vyhledávání stačí levým tlačítkem myši kliknout do panelu vizualizace a tím určit bod q. Ve výsledku je pak vidět odlišná barva nalezených nejbližších sousedů než ostatních objektů datové struktury. Dále je zobrazen i minimální vyhledávací region jehož poloměr udává výslednou velikost D_k.



Vizualizace výsledku algoritmu, zdroj [autor]