VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

AKCELERACE VYHLEDÁVÁNÍ V PROSTOROVÝCH STRUKTURÁCH

SEARCH ACCELERATION IN SPATIAL STRUCTURES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **JAKUB VLK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. MICHAL VLNAS**
SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

152771

Ústav:	Ústav počítačové grafiky a multimédií (UPGM)
Student:	**Vlk Jakub**
Program:	Informační technologie
Název:	**Akcelerace vyhledávání v prostorových strukturách**
Kategorie:	Počítačová grafika
Akademický rok:	2023/24

Zadání:

1. Nastudujte algoritmy prostorových struktur (octree, BSP, BVH, kD-tree apod.) se speciálním zaměřením na reprezentaci orientovaných bodů.
2. Nastudujte metody pro vyhledávání kNN v těchto strukturách.
3. Implementujte některé prostorové struktury, včetně algoritmů pro jejich průchod a vyhledávání.
4. Navrhněte vhodnou akceleraci zvolených algoritmů, např. pomocí vektorizace či paralelizace.
5. Změřte a vyhodnoťte efektivitu algoritmů a využitelnost na konkrétních prostorových problémech.

Literatura:

- M. de Berg, Computational Geometry: Algorithms and Applications. Springer, 2008.
- Drost, Bertram & Ilic, Slobodan. (2018). Almost constant-time 3D nearest-neighbor lookup using implicit octrees. Machine Vision and Applications. 29. 1-13. 10.1007/s00138-017-0889-4.

Při obhajobě semestrální části projektu je požadováno:
Bod 1, 2 a experimenty vedoucí k bodu 3 a 4.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Vlnas Michal, Ing.**
Vedoucí ústavu:	Černocký Jan, prof. Dr. Ing.
Datum zadání:	1.11.2023
Termín pro odevzdání:	9.5.2024
Datum schválení:	9.11.2023

Abstrakt

Tato práce představuje implementaci rychlých algoritmů pro nalezení nejbližšího souseda, které efektivně určují, který bod z dané množiny je nejblíže k zadanému bodu. Algoritmy jsou navíc škálovatelné pro hledání k-nejbližších sousedů. Součástí je i specializované vyhledávání bodů s podobnou orientací na základě specifických kritérií a většího množství přístupů pro hledání orientovaných bodů. Struktura využívá vlastností z Voronoi diagramu, Octree, ale i hašovací tabulky nebo binární vyhledávání. Složitost u vyhledání nejbližšího souseda dosahuje časů blížících se konstantním hodnotám, neboť celková složitost je logaritmicky logaritmická. Práce obsahuje podrobné testování, jak po stránce přesnosti, tak po stránce výkonnosti.

Abstract

This thesis presents the implementation of a fast nearest neighbor lookup algorithm, which identifies the closest point from a set of points to other points. The algorithm is scalable for searching for k-neighbors, and it supports the identification of oriented points according to selected criteria and various approaches. The structure offers various approaches and utilizes properties of structures such as Voronoi diagrams, Octree binary search, or hash tables. The complexity of the nearest neighbor search is nearly constant because the cost is logarithmically logarithmic. The thesis shows numerous benchmarks for accuracy and performance.

Klíčová slova

Prostorové struktury, akcelerace vyhledávání, oktalový strom, hašovací tabulka, nejbližší soused, k-nejbližší soused, vektorizace, paralelní zpracování, optimalizace algoritmů, strojový kód

Keywords

Spatial structures, search acceleration, octree, hash table, nearest neighbor, k-nearest neighbors, vectorization, parallel processing, algorithm optimization, machine code

Citace

VLK, Jakub. *Akcelerace vyhledávání v prostorových strukturách*. Brno, 2024. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Michal Vlnas

Akcelerace vyhledávání v prostorových strukturách

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Vlnase. Během práce jsem použil umělou inteligenci pro vyhledávání informací, objevování nových zdrojů a pře-frázování vět. Další informace mi poskytl pan doktor Rebenda. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Jakub Vlk
8. května 2024

Poděkování

Na tomto místě bych rád poděkoval panu inženýrovi Vlnasovi, který mě zásoboval po technické, vědomostní, pravopisné, ale i kulturní stránce po dobu tvorby této bakalářské práce. Rovněž bych chtěl poděkovat svým kamarádům, které jsem využil jako „gumové kachničky“ pro myšlenkové rozbory problémů, i když často nevěděli o čem mluvím. Rád bych poděkoval i své babičce, která moc dobře ví, že zákusky jsou základ a poskytla mi kvalitní zásobování takto základní potřeby, nehledě na pomoc s bojem se zákeřnými nástrahami jazyka českého. Už navždy v drobence jablkového koláče uvidím oktalový strom a v polevě větrníku nedokonalé rozložení prvků hašovací tabulky. Do smrti se neodpustím, že jsem binárním vyhledáváním zmrzačil nebohou koblihu, která stejně neobsahovala chtěný prvek.

Obsah

Kapitola 1

Úvod

Tato práce má za úkol vytvořit rychlou implementaci algoritmu nejbližšího souseda definovaného nad množinou bodů a bodem, kde samotný problém tkví v nalezení nejbližšího bodu k zadanému bodu z oné množiny. Tento algoritmus lze zobecnit na hledání k-nejbližších sousedů takového bodu. Tato práce rovněž implementuje vyhledávání podobně orientovaných blízkých bodů podle nastavených kriterií.

Tyto algoritmy se využívají v oblastech umělé inteligence, robotiky, počítačové grafiky a jiných. Podle aplikace jsou struktury optimalizovány pro různé parametry: doba stavby, rychlost získání nejbližšího souseda, paměťovou náročnost, chování v mnohodimenzionální prostorech a jiné. Základní struktury využívají pouze dělení prostoru na různě definované pod-části ať už binární, nebo jiné, které poté řadí do stromových struktur a značně zvyšují efektivitu vyhledávání. Jiné struktury využívají různých technik pro seskupení blízkých bodů a tím sníží rozsah prohledávaných oblastí, ale za to mohou snížit přesnost.

Prezentovaná struktura je náročná na výstavbu, ale principiálně nabízí nalezení nejbližšího souseda v téměř konstantním čase. Této vlastnosti je docíleno promyšleným použitím hašovací tabulky, binárního vyhledávání, Voronoi diagramu a vyhledávacího oktalového stromu. Jednotlivé vrstvy jsou uspořádány do pyramidové struktury, kde se jedna vrstva vystavuje na základě té předešlé, čímž vznikne struktura optimalizována primárně na rychlost vyhledávání. Tato varianta využívá, jak prostorové struktury, tak dokonce i techniku seskupování.

Cílem práce bylo najít a naimplementovat strukturu umožnující nejrychlejší možné vyhledávání nejbližších (orientovaných) sousedů. V kritických místech, kde to bylo možné a efektivní byly navrhnuty formy optimalizací pomocí vektorizace a paralelizace ať už pro stavbu nebo vyhledávání. Implementace byla podrobena testům, jenž odhalily její silné ale i slabé stránky a poskytly kontext v nichž je vhodné takovouto strukturu použít a jaké se dají očekávat výsledky.

V kapitole „Akcelerační, prostorové a optimalizační struktury a algoritmy nad nimi“ je rozebrán teoretický úvod většiny komplexnějších konceptů použitých při implementaci. Na tento úvod navazuje kapitola věnující se tomu, jakým způsobem byly jednotlivé teoretické koncepty zkombinovány do jedné struktury a jsou zde uvedeny koncepce toho, jak funguje vyhledávání a výstavba struktury. V této kapitoly je zmíněno jakým způsobem bude implementace testována, jak po výkonnostním, tak po správnostní stránce. Následující kapitola popisuje, jak byli koncepty z předešlé kapitoly přetvořeny na program. Popisuje jaké knihovny a jak byly použity a zmiňuje zajímavosti z implementační části této práce.

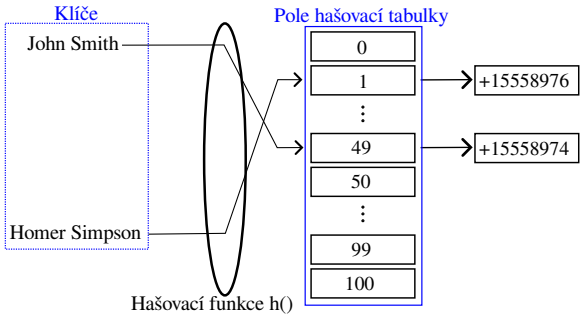
Kapitola 2

Akcelerační, prostorové a optimalizační struktury a algoritmy nad nimi

Efektivní implementace algoritmu nejbližšího souseda mohou pro zrychlení používat různé prostorové, ale i jiné struktury. Při optimalizacích implementace těchto struktur je možné použít speciální instrukce, které vektorizují nebo jinak zrychlují klíčové části implementace. V této sekci je možné nalézt příklady několika struktur, které se dají použít pro implementaci nejen algoritmu nejbližšího souseda, ale mohou se používat i jinde. Mimo to je níže možné najít i popis procesu vektorizace a použití vektorových optimalizací.

2.1 Hašovací tabulka

Hašovací tabulka, někdy je možné setkat se i s názvem slovník nebo mapa, je datová struktura umožnující přístup k datům pomocí klíče neboli vstupní hodnoty. Této vlastnosti je dosaženo pomocí dobře zvolené funkce, které se říká hašovací. Vlastnosti funkce jsou rozebrány v sekci níže. O vkládání a vyhledávání v této struktuře je možné se dočíst níže v sekcích o vkládání a kolizích.



Obrázek 2.1: Na obrázku lze vidět strukturu hašovací tabulky s příkladem vložení jmen „John Smith“ a „Homer Simpson“ do tabulky. Hašovací funkce $h()$ se používá k určení pozice v tabulce, přiřazuje jména do různých indexů v poli. „John Smith“ odkazuje na index 49, zatímco „Homer Simpson“ na index 1. Jména jsou použita jako klíče pro přístup k telefonním číslům.

Hašovací funkce

Hašovací funkce je definována jako $h(k) : K \to Y$, kde K je množina vstupních prvků, zpravidla struktur a Y je interval, na který h zobrazuje. Vlastnosti funkce h jsou dále rozebrány v sekci věnující o vlastnostem hašovací funkce. Tato funkce má na vstupu libovolný prvek k ze vstupní množiny K, obvykle se hovoří o klíčích. Tuto vstupní množinu mapuje na interval Y od nuly do $M - 1$, kde je $M > 0$. V praxi se kvůli minimalizaci počtu kolizí je možné se spíše setkat s $M > |K|$. Výstupu hašovací funkce se říká haš, nebo také otisk. M je velikost pole, které používá hašovací tabulka. Na obrázku 2.1 je znázorněno, jak hašovací tabulka funguje [19].

Někdy se hašovací funkce rozděluje na dvě – interní a externí [21]. Externí funkci je možné mezi jednotlivými konstrukcemi hašovacích tabulek měnit a slouží pro výpočet samotného haše na obecně libovolném intervalu. Tento interval je potřeba transformovat na $\langle 0, M - 1 \rangle$, kde M je velikost pole[1], které je k dispozici k použití pro hašovací tabulku. Jedná se tedy o vnitřní záležitost hašovací tabulky a pro hašovací funkci to není podstatné. V textu dále nebude rozlišováno mezi interní a externí hašovací funkcí, bude na ně nahlíženo jako na jeden celek.

Je důležité, aby hašovací funkci bylo možné spustit nad libovolnými vstupními daty. Pokud vstupem hašovací funkce bude řetězec, hašovací funkce musí zohlednit všechny znaky v tomto řetězci. Obdobně je tomu i u jiných vstupů.

Vlastnosti hašovací funkce

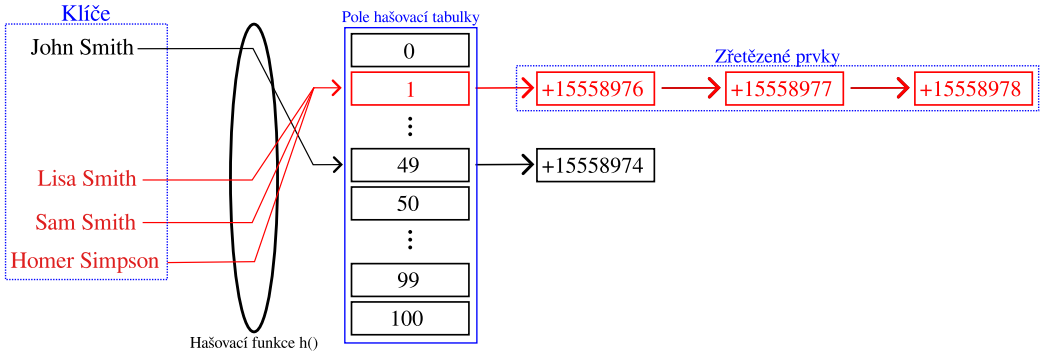
Pro správnou a efektivní činnost je klíčové, aby hašovací funkce splňovala jisté charakteristiky:

1. Deterministická – pro každý jeden vstup vždy vrátí stejný výstup.
2. Dodržení rovnoměrného rozdělení po celém intervalu – hašovací funkce nebude vracet jeden otisk neúměrně častěji než otisky jiné.
3. Rychlý výpočet – celý výkon hašovací tabulky stojí na rychlosti hašování.

První charakteristika je nutná z důvodu toho, aby při vyhledávání byla nalezena vždy stejná hodnota. Funkce musí pracovat jen na základě jediného vstupu – klíče. Zachování rovnoměrného rozdělení je nutné především, nejen však v případech, když je interval na který hašovací funkce zobrazuje menší než mohutnost vstupní množiny. Tím pádem musí několik klíčů vyprodukovat stejný otisk, nastane tedy kolize. O problémech, které mohou nastat, je víc napsáno v sekci o kolizích. Třetí charakteristika není nutná pro samotnou funkcionalitu, ale především pro reálnou použitelnost a nasazení hašovací tabulky. Někdy se dokonce používá i strojově akcelerované hašování [6].

Pokud je hašovací tabulka statická, tedy jednou se vytvoří a poté se už pouze používá k vyhledávání, může být funkce injektivní. Poté není nutné při vkládání ani při vyhledávání řešit kolize. Vyhledávání tak bude o to rychlejší. Pokud je kladen nárok na šetření paměti, je vhodné připravit hašovací funkci tak, aby interval, na který zobrazuje, byl stejně velký jako mohutnost vstupní množiny. V tomto případě je ale nutné vědět, která data konkretně nebo vstupní klíče budou na vstupu hašovací funkce. Dalším omezením může být také stále náročnější nalézání ideální funkce s rostoucí vstupní množinou [11].

[1] Předpokládá se, že pole bude indexované od nuly.



Obrázek 2.2: Hašovací tabulka používající strategii řetězení pro řešení kolizí. Hašovací funkce vygenerovala pro „Homer Simpson“, „Sam Smith“, „Lisa Smith“, stejný otisk, znázorněno červenou. Při pokusu o vyhledání posledního čísla bude nutné projít celou posloupnost čísel.

Z otisku zpravidla není možné sestavit původní klíč. Pro účely hašovací tabulky ovšem není důležité, aby mezi klíčem a otiskem neexistoval vztah. Existuje také řada dalších vlastností, které mohou být kladeny na hašovací funkce, ale pro účely hašovací tabulky není potřeba jejich dodržení.

Vkládání prvků

Před tím, než začne vkládání, je nutné mít připravené pole o velikost M – tedy o maximální velikosti intervalu. Data v hašovací tabulce jsou ukládána do tohoto pole. Při vkládání jsou vstupní data vložena jako vstup hašovací funkce. Výstupem je otisk, nebo-li index, na který budou data uložena. Problém může nastat v případě, že se ve vstupní množině nachází takové dva klíče, které mají stejný otisk. Tato problematika je dále rozebrána v části věnující se kolizím.

Při vyhledávání se hledaný klíč zahašuje. Podle určeného indexu se přejde na místo v poli a ověří se, zda byla nalezena pro správný klíč odpovídající data.

Kolize při vkládání prvků

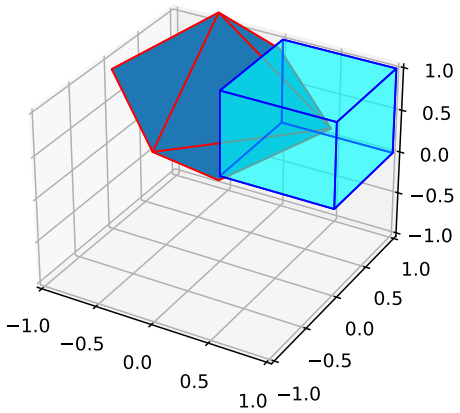
Kolize v hašovací tabulce je pojmenování pro situaci, kdy hašovací funkce pro dva různé klíče vrátí stejné otisky, tedy $h(k_1) = h(k_2)$, $k_1 \neq k_2$. Tato situace může nastat z několika příčin. Může se jednat například o nevhodně zvolenou hašovací funkci nebo příliš malý zobrazovaný interval hašovací funkce. Kolize lze řešit dvěma základními způsoby:

1. Řetězením

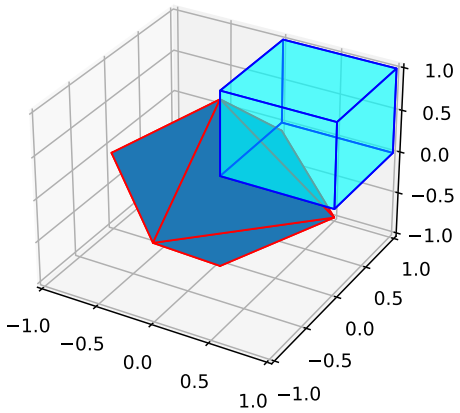
   V případě, že dojde ke kolizi, je prvek umístěn na stejné místo, jako již existující zařazený prvek. V praxi se na indexech hašovací tabulky nachází struktury, které umožňují skladování více prvků. Jeden z příkladů je použití lineárního seznamu.

2. Otevřená adresace

   V případě, kdy dojde ke kolizi, je použit alternativní způsob umístění prvku do pole. To umožňuje například použití druhé hašovací funkce, dvojitého hašování anebo připočtení k otisku konstanty a jiné.



(a) K_1 a K_2 zobrazeny v prostoru. Jeden z vrcholů K_2 je uvnitř krychle K_1.



(b) K_1 a K_2 zobrazeny v prostoru. Žádný z vrcholů K_2 není uvnitř krychle K_1 a přesto mají společnou podčást.

Obrázek 2.3: Porovnání dvou situací při průniku krychle a konvexního mnohostěnu.

Tyto strategie nejsou jediné možnosti, jakými se lze vypořádat s kolizemi. Přinejmenším je možné tyto způsoby kombinovat. Existují však i jiné pokročilejší strategie, jak se vypořádat s kolizemi [21].

Kolize mohou mít velmi negativní vliv na výkonnost hašovací tabulky. Jako nejhorší případ je možné si představit situaci znázorněnou na obrázku 2.2. Jde o situaci, kdy hašovací funkce bude pro mnoho prvků vracet stejný otisk. Z toho důvodu hašovací tabulka zdegraduje ze složitosti $O(1)$ až na složitost $O(n)$, kde n je počet prvků ve struktuře. Za předpokladu, že by všechny vložené prvky byly umístěny na stejný index v poli. V nejhorším případě je tedy při vyhledávání potřeba projít celou strukturu uchovávající prvky se stejným otiskem.

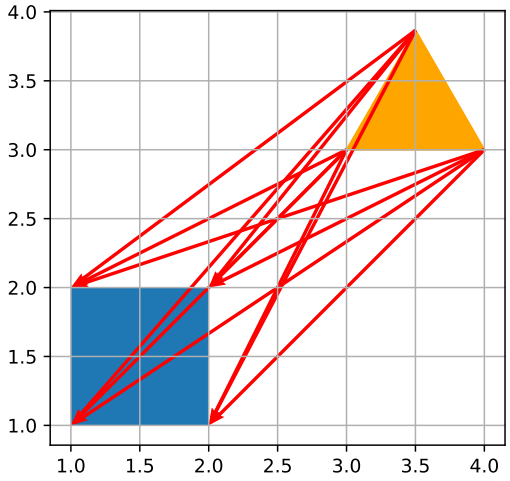
2.2 Detekce kolizí konvexních mnohostěnů

Nechť existují dva konvexní mnohostěny K_1 a K_2. Mnohostěny K jsou zadány vrcholy k_0 až k_n, kde $n \in \mathbb{N}$.

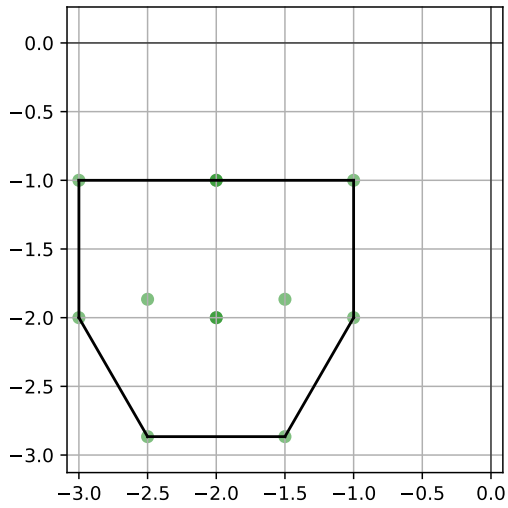
Aproximace detekce kolize osově zarovnané krychle a konvexního mnohostěnu

Nechť K_1 je konvexní mnohostěn a K_2 je pravidelná krychle zarovnaná se souřadným systémem. Krychli lze vyjádřit prostřednictvím dvou bodů U_l a U_r, určujících protilehlé vrcholy.. Potom stačí, aby alespoň jeden z bodů mnohostěnu K_1 ležel v krychli K_2, což ověřit je velmi snadné. Potom musí alespoň částečně existovat společná pod-část K_1 a K_2, což se dá velmi snad ověřit.

Při provedení popsaného postupu lze dojít k nesprávnému výsledku. Taková situace je zobrazena na obrázku 2.3. V některých aplikacích však tento rozdíl lze zanedbat. V případě, že K_1 i K_2 jsou osově zarovnané krychle, tak není možné, aby tato detekce někdy fungovala špatně [10].



(a) Grafické znázornění výpočtu množiny Minkowského rozdílů.



(b) Body z Minkowského rozdílů jsou vykresleny zelenou barvou. Body jsou obklopeny nejmenší konvexní obálkou pro tyto body.

Obrázek 2.4: Minkowského rozdíl.

Minkowského rozdíl

V angličtině je znám pod pojmem *Minkowski difference* [1], je množina bodů obecně v p-dimenzionálním prostoru, která vyjadřuje vztah mezi dvěma objekty A a B. Tyto objekty jsou zadány vrcholy. Její výpočet probíhá iterativně, kdy je nutné projít všechny kombinace vrcholů a nad každou dvojicí je vypočten jejich rozdíl, čímž vznikne nová množina C o mohutnosti $M = |A| \cdot |B|$, což zkomplikuje konstrukci pro objekty s mnoha vrcholy. Množina C obsahuje vektory, tak, jak jsou znázorněny na obrázku 2.4a.

Vzniklé body je možné uzavřít do konvexní obálky [1], čímž vznikne tvar, jak je možné vidět na obrázku 2.4b, jehož poloha reprezentuje vzájemnou pozici, tedy zdali jsou v kolizi nebo ne. Pokud se v této obálce nachází i počátek souřadného systému, znamená to, že jsou ony objekty v kolizi. Diskutovanou situaci a zobrazení výpočtu je možné vidět na obrázku 2.5. Formálně zapsaný vztah lze vyjádřit[2] [1]:

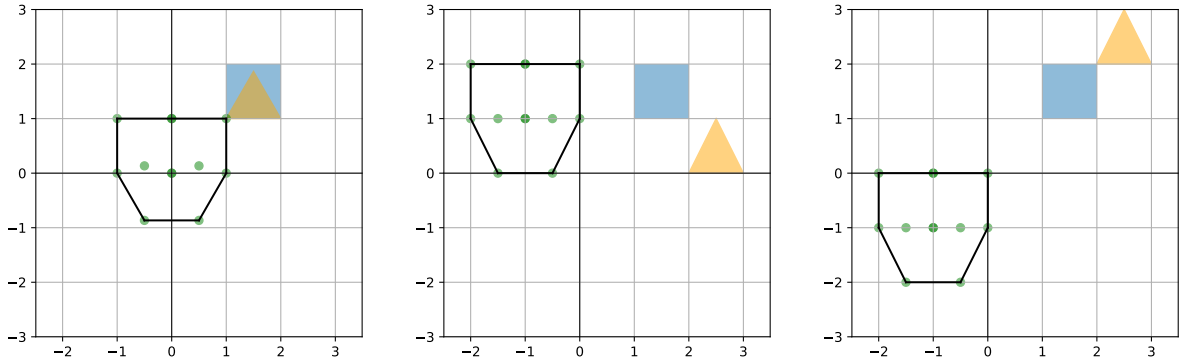
$$A - B = \{\mathbf{a} - \mathbf{b} \,|\, \mathbf{a} \in A,\ \mathbf{b} \in B\}$$

Algoritmus GJK

Algoritmus Gilbert–Johnson–Keerthi (GJK) [8] využívá vlastnosti získatelných z Minkowského rozdílu, ale vzhledem k tomu, jak roste výpočetní náročnost s množstvím dimenzí a vrcholů, je obvykle velmi náročné vypočítat tuto množinu. Je třeba tedy minimalizovat počet výpočtů, který je potřebný k detekci kolize.

Obecný GJK algoritmus má na výstupu vzdálenost dvou objektu. Existují i varianty, které jsou optimalizované jen pro odpověď ano/ne.

[2] https://www.youtube.com/watch?v=MDusDn8oTSE&



Obrázek 2.5: Z obrázku lze vyčíst, jakým způsobem se chová množina daná Minkowského rozdílem v různých pozicích, především pak při kolizi (vlevo).

Support funkce

Při pohledu na obrázek 2.5 lze vidět, že pro určení, zda jsou dva objekty v kolizi, stačí použít pouze okrajové body – tedy ty, které tvoří onu konvexní obálku a dokonce i snad nejsou potřeba všechny body, ale jen některé v blízkosti počátku souřadného systému. Tyto body lze poznat pomocí toho, že jsou nejvzdálenější od středu ve směru $D \in \mathbb{R}^p$:

$$\max(D \cdot (A \ - B)),$$

což nikterak nepomůže při optimalizaci složitosti, ta je stále $O(|A| \cdot |B|) = O(n^2)$. Nicméně při roznásobení a použití funkce max pro oba členy samostatně vznikne vztah[3]:

$$\max(D \cdot A) - \max(-D \cdot B)^{4},$$

který lze řešit v lineárním čase $O(|A| + |B|) = O(n)$. Popsaný postup se nazývá v angličtině *Support function* a vypočítává z Minkowského rozdílů pouze to nejnutnější. Výsledkem je jediný vrchol Minkowského rozdílů, jak lze vidět na obrázku 2.6. Pak už je otázkou několika iterací pro získání dostatečně velké části obálky, aby bylo možné rozhodnout, zda se počátek souřadného systému nachází uvnitř [8].

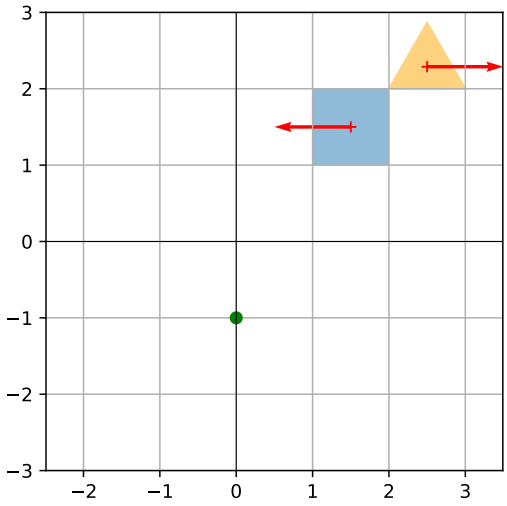
Simplex

Nejjednoduší tvar pro výběr oblasti v obecně p dimenzionálním prostoru se nazývá simplex. Tedy pro $p = 1$ se jedná o bod, pro $p = 2$ úsečku pro $p = 3$ trojúhelník a $p = 4$ čtyř stěn. Je potřeba tedy $p+1$ bodů. Tento tvar pro účely GJK algoritmu reprezentuje nějakou pod-část vzniklé obálky z Minkowského rozdílů. Postupnou iterací lze nalézt simplex, který je nejblíže počátku a má tedy největší šanci, že do něj bude patřit počátek souřadného systému, čímž se dosáhne tíženého výsledku a odpoví na otázku, zda jsou dva konvexní tvary v kolizi [8].

Otázkou může být jakým způsobem bude algoritmus aktualizovat směr D, protože pro stejný směr by funkce support vracela stále stejný bod. Simplex bude pro $p = 3$ tvořen jedním až čtyřmi body. Směr D aktualizuje funkce NearestSimplex(s), která má na vstupu simplex s a na základě něj se rozhodne, jak se vypočítá další směr. Tedy při každé iteraci

[3] https://winter.dev/articles/gjk-algorithm

[4] Pro zachování funkčnosti je potřeba pro B obrátit směr D.



Obrázek 2.6: Vypočítaný bod (označený zelenou barvou) pomocí funkce support pro směr (1,0) respektive (-1,0), označený červenou barvou.

algoritmu je výstupem funkce NearestSimplex(s) nový směr a případně odpověď, zda jsou v kolizi. Simplex může obsahovat jeden z těchto tvarů:

Pro přehlednost je simplex struktura s kapacitou $p + 1$ bodů, která umožňuje vkládat prvek na nulté místo a zbytek prvků posune o jedno a případně přebytečné body zahodí. Dále pak nultý prvek bude označován jako bod a, první b a tak dále.

Bod

Pokud je simplex jen jeden bod a, tak se směr D změní na směr k počátku souřadného systému. Nový směr je znázorněn na obrázku 2.7a.

Úsečka

V případě, že simplexem je úsečka, je dána dvěma body a a b. Nový směr bude vypočítán jako normálový vektor ve směru k počátku 2.7b.

Trojúhelník

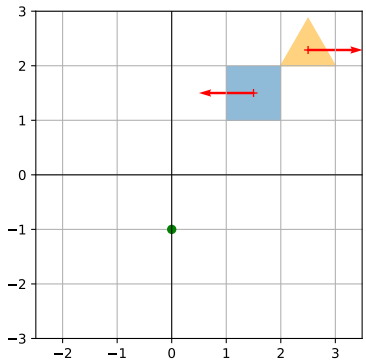
V případě, že simplexem je trojúhelníkem, tak už dokáže vybrat oblast, ve které by potenciálně mohl být počátek souřadného systému. Celá situace je znázorněna na 2.7c.

Čtyřstěn

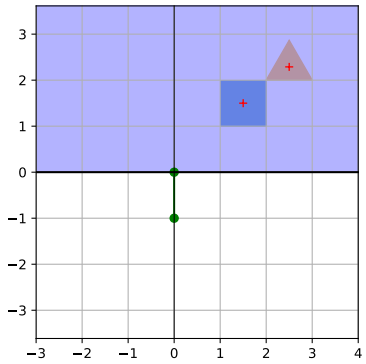
Pokud je potřeba detekovat kolize pro $p = 3$ prostor, tak je potřeba uvažovat o případě, kde je simplex zadán čtyřmi body. V takovém případě se jedná o čtyřstěn.

Hlavní cyklus algoritmu

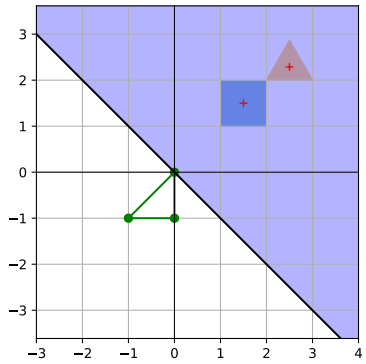
Nechť existuje simplex $s = \emptyset$ a je stanovena maximálně mohutnost, jako $p + 1$, dále pak existuje směr D, zvolený například jako konstanta $(1, 0)$. Poté existují dva konvexní tvary A a B, které jsou dány vrcholy. Příklad hlavního cyklu může vypadat a:



(a) Simplex sestavený z jednoho bodu. Nový směr pro, který bude vypočítán nový bod z Minkowského rozdílu (označený zelenou barvou).

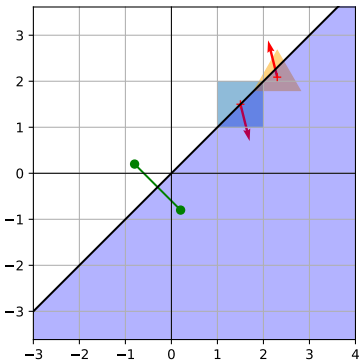


(b) Simplex sestavený ze dvou bodů (označený zelenou barvou). Je třeba provést kontrolu, zda nově vzniklý bod je v sekci (znázorněné modrou barvou), která by umožnila vznik trojúhelníku obsahující počátek souřadného systému. Na obrázku nově vzniklý bod je v této oblasti (nachází se na hranicích). Na obrázku není vidět směr D, neboť bude určen jako (0,0).

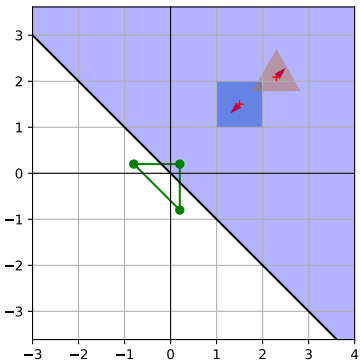


(c) Simplex sestavený ze tří bodů – označený zelenou barvou ve tvaru trojúhelníku. Je třeba zkontrolovat, zda se nově vzniklý bod nachází v sekci (znázorněné modrou barvou), která by umožnila vznik trojúhelníku obsahující počátek souřadného systém. Na obrázku nově vzniklý bod není v této oblasti. Algoritmus skončí a vrátí *false*.

Obrázek 2.7: Algoritmus GJK při ne-kolizi.

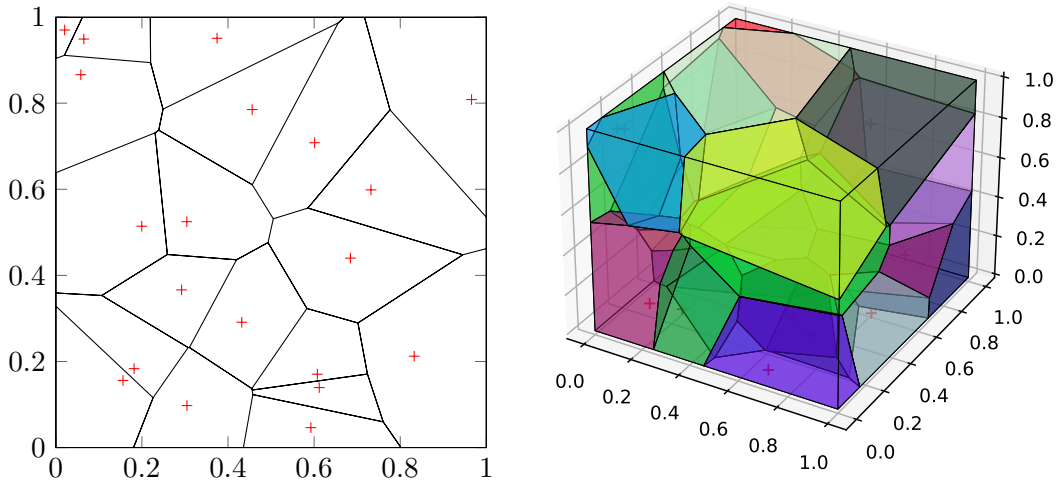


(a) Simplex sestavený ze dvou bodů – označený zelenou barvou. Je třeba zkontrolovat, zda se nově vzniklý bod nachází v sekci (znázorněné modrou barvou), která by umožnila vznik trojúhelníku obsahující počátek souřadného systém. Na obrázku nově vzniklý bod je v této oblasti.



(b) Simplex sestavený ze tří bodů (označený zelenou barvou ve tvaru trojúhelníku). Je třeba zkontrolovat zda se nově vzniklý bod nachází v sekci (znázorněné modrou barvou), která by umožnila vznik trojúhelníku obsahující počátek souřadného systém. Na obrázku nově vzniklý bod je v této oblasti. Mimo to vzniklý trojúhelník vytvořil obal kolem počátku, algoritmus končí a vrací *true*.

Obrázek 2.8: Algoritmus GJK při kolizi.



Obrázek 2.9: Každý z obrázků obsahuje 20 označených bodů (označených „+“) a jejich Voronoi diagram, pro $p = 2$ a $p = 3$.

1. Vypočtení nové hodnoty *support* z funkce *Support* pro tvary A, B a směr D.
2. simplex = $\{support\}$ a $direction = -support$.
3. Opakuj:
   (a) $support = Support(A, B, D)$.
   (b) Pokud $(dot(support, D) <= 0)$, tak nemůže dojít ke kolizi.
   (c) Do simplex množiny je vložen nový prvek a v případě, že množina dosáhla maximální velikosti, je poslední získaný bod vyloučen z množiny.
   (d) Pokud funkce $NextSimplex(s, D)$ vrátí *true*, tak je A a B v kolizi. Jinak opakuj.

2.3 Problém nejbližšího souseda a Voronoi diagramy

Funkce k-nejbližšího souseda je obvykle založena na Euklidovské vzdálenosti mezi vstupním bodem a vstupní množinou bodů. Euklidovskou vzdálenost lze obecně definovat pro body mající p dimenzí. Pro výpočet vzdálenosti mezi dvěma body x_i a x_j je možné použít následující vzorec [19]:

$$d(x_i, x_j) = \sqrt{(x_{i1} - x_{j1})^2 + (x_{i2} - x_{j2})^2 + \cdots + (x_{ip} - x_{jp})^2}.$$

Přirozený způsob grafického znázornění myšlenky problému nejbližšího souseda je pomocí Voronoi diagramu [5]. Toto lze vidět na obrázku 2.9. V diagramu se nachází 20 označených bodů pomocí „+“ a příslušné Voronoi buňky R_i, které obklopují každý bod. Každá Voronoi buňka R_i je množinou všech bodů, pro které platí, že má: nejmenší vzdálenost k bodu x_i. Tento vztah lze definovat [19]

$$R_i = \{x \in \mathbb{R}^p : d(x, x_i) \leq d(x, x_m), \forall i \neq m\},$$

kde R_i je Voronoi buňka pro bod x_i. Při pohledu na obrázek 2.9 lze vidět, že ať už je vybrán jakýkoliv bod z oblasti Voronoi buňky R_i, tak jeho nejbližší sousedem bude vždy bod x_i. Další zkoumání prozradí, že pokud je potřeba najít pro vstupní bod jeho nejbližší bod náležící do množiny vstupních bodů, bude tento bod určen nejbližší hranou Voronoi buňky. Druhou z vlastností je patrný problém k-nejbližších sousedů. Tento problém je možné dále definovat, jako přiřazení testovanému bodu k nejbližších bodů ze vstupní množiny bodů. V praxi je obvykle hodnota k zvolena jako liché číslo, aby se předešlo remízám. Obvykle pro $k = 1$ se nehovoří o problému prvního-nejbližšího souseda ale přímo o problému nejbližšího souseda [19].

2.4 Prostorové struktury

Prostorové struktury jsou velmi často využívány v mnoha oborech informatiky, jakožto například ve strojovém učení, robotice, geografických informačních systémech a také v počítačové grafice. Častou aplikací prostorových struktur v počítačové grafice je problém nejbližšího souseda [4].

Octree a quadtree

Oktalové stromy, octree nebo quadtree jsou relativně jednoduché prostorové datové struktury, které pracují na principu rozdělování podprostoru na poloviny v každé dimenzi. Pokud jsou zadány body v 3D prostoru, tak se hovoří o octree, protože když je prostor v každé dimenzi rozdělen na polovinu, tak vznikne 8 krychlí. Pokud jsou použity pouze 2 dimenze, vzniknou 4 čtverce – tedy quadtree. Vzhledem k nevýhodám, o kterých se hovoří níže, se tato struktura obvykle používá především pro nižší počet dimenzí [5].

Konstrukce Octree

Prostor je tak dlouho rekurzivně dělen v každé rovině na polovinu, dokud není splněna podmínka zastavení, zpravidla mohutnost množiny bodů v uzlu. Rekurzivní sestupem vznikne stromová struktura, kdy každý prvek může mít 0 až 2^p synů [5].

Vyhledávání nejbližšího souseda

Algoritmus postupně, rekurzivně vylučuje uzly stromu. Začne v uzlu listovém, do kterého by náležel bod ke kterému je hledán nejbližší soused a pak postupuje ostatními uzly dokud není vyloučeno, aby se nacházel nejbližší soused jinde [7]:

1. Algoritmus na každé úrovni rozhodne, do kterého oktetu (čtvrtiny) by připadl vyhledávaný bod, dokud nedojde až do listového uzlu S_{leaf}.
2. V listovém uzlu nalezne nejbližší bod b se vzdáleností d.
3. Vznikne hyperkoule[5] k s průměrem d a středem b.
4. Pokud tato kružnice k:
   (a) zasahuje i mimo listový uzel, tak je vyhledávání znovu spuštěno nad otcem S_{leaf} a pokračuje se bodem 2.

[5] Pro $p = 2$ se jedná o kružnici, pro $p = 3$ se jedná o kouli.

(b) zasahuje pouze do oblasti dané S_{leaf}, takže bod b je nejbližším sousedem hledaného bodu.

Výhody a nevýhody

Velkou výhodou octree je, že jeho implementace je velmi jednoduchá. Například knižní implementace na webu geeksforgeek[6] má méně než 300 řádků.

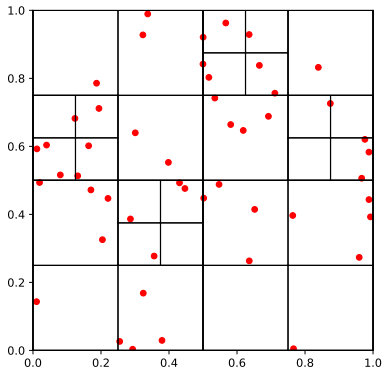
Nevýhodou ale může být i například to, že tyto datové struktury mohu vytvářet velmi hluboké stromy i pro velmi malé datové sady a jestliže se v sadě nachází velké množství bodů u sebe. Při dělení prostoru může hloubka stromu značně narůst a strom bude velmi řídký. Na obrázku 2.10 lze vidět takovou situaci. Pak strom defakto zdegraduje na lineární seznam, ve kterém je značně náročnější vyhledávání než u stromových algoritmů.

Další nevýhodou je to, že množství synů roste exponenciálně s počtem dimenzí p. Počet synů je roven 2^p, což u mnoho dimenzionálních prostor může značně ovlivnit nejen dobu stavby stromu, ale i vyhledávání. Mimo zmíněné je octree zatížen prokletím dimensionality.

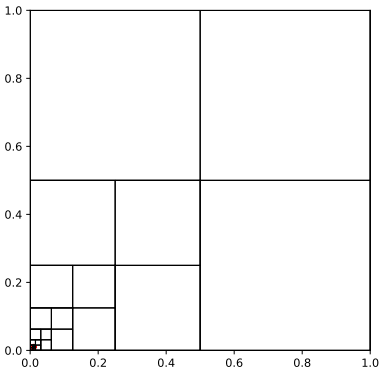
KD-tree

KD-tree je binární stromová struktura, která pracuje na principu postupného dělení jedné dimenze tak, aby množinu bodů vždy rozdělila na půl [17]. Tímto způsobem vznikne struktura, která se daleko lépe přizpůsobuje nepravidelnosti vstupní množiny bodů [7].

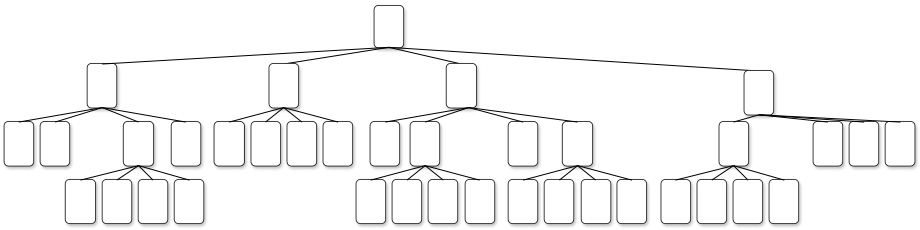
[6] https://www.geeksforgeeks.org/octree-insertion-and-searching/



(a) Rovnoměrně náhodně rozložených 50 bodů. Maximální hloubka stromu je 3, pro $m = 3$.



(b) Nerovnoměrně rozležení bodů, v okolí počátku souřadného systému. Maximální hloubka stromu je 10, pro $m = 3$



(c) Stromové zobrazení quadtree pro stejné body, jako na obrázku 2.10a.

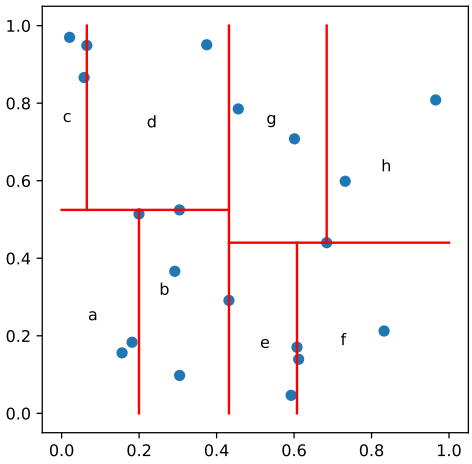
Obrázek 2.10: Porovnání hloubky quadtree pro stejné množství bodů s jiným rozmístěním v prostoru.

Konstrukce KD-tree

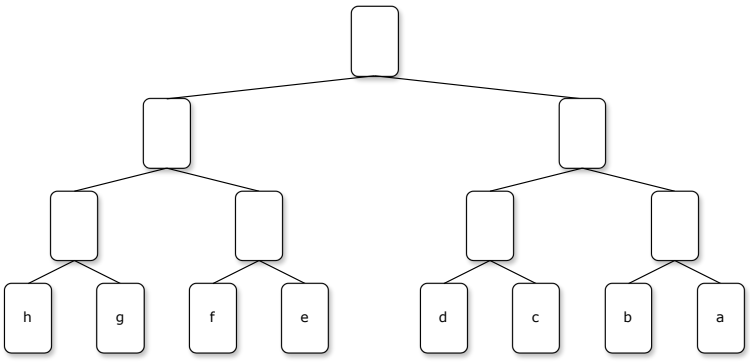
Vstupem pro algoritmus konstrukce KD-tree je množina bodů $S \subseteq \mathbb{R}^p$ a konstantu M_{max}, která určuje kolik bodů maximálně se má nacházet v listovém uzlu [17]. Na obrázku 2.11 lze vidět vystavěný KD-tree.

KD-tree je velmi často využíván a existuje mnoho způsobů, jak může být sestaven. Zde je například uveden jednoduchý způsob, jak může být tato struktura vytvořena [17]:

1. Je vybrána jedna z dimenzí $d \in \{1, ..., p\}$.
2. Je nalezen medián m d-té dimenze ze všech bodů S.
3. Vznik dvou množin $S_l \in \{S[d] < m\}$ a $S_r \in \{S[d] \geq m\}$, kde S_l a S_r jsou synové S.
4. Algoritmus je znovu spuštěn nad takovým S_l nebo S_r, kde mohutnost je větší než konstanta M_{max}. V opačném případě je ona množina označena za listový uzel.



(a) Vybudovaný KD-tree pro 20 bodů v rovině.



Obrázek 2.11: Vybudovaný KD-tree pohled na stromovou strukturu.

Existuje mnoho variant, jak vystavět KD-tree. Jedna ze základních možností je popsána v následujících bodech:

Pro výběr dimenze d je možné použít heuristiku jakožto například vybrání té dimenze, jejíž rozptyl je největší[7]. Další možností je postupovat od 1 až do p, a poté znovu začít od jedničky.

Vyhledávání nejbližšího souseda

Algoritmus pracuje úplně stejným způsobem, jako u octree [7]:

1. Algoritmus na každé úrovni rozhodne zda by bod připadl do S_l nebo S_r dokud nedojde až do listového uzlu S_{leaf}.
2. V listovém uzlu nalezne nejbližší bod b se vzdálenosti d.
3. Vznikne kružnice[8] k s průměrem d a středem b.
4. Pokud tato kružnice k:
   (a) zasahuje i mimo listový uzel, tak je vyhledávání znovu spuštěno nad otcem S_{leaf} a pokračuje se bodem 2.
   (b) zasahuje pouze do oblasti dané S_{leaf}, tak bod b je nejbližším sousedem hledaného bodu.

Nevýhody a výhody

Struktura je jednoduše implementovatelná. KD-tree je taktéž zatížen prokletím dimenzionality z toho pramenící neefektivita s větším počtem dimenzí a další problémy spjaté s prokletím [7].

Ball Tree

Ball tree, balltree, nebo česky koulový strom je stromová struktura, která rozděluje prostor na podprostory. Tyto podprostory jsou charakterizovány maximální euklidovskou vzdáleností rozebrané v sekci 2.3 pro obecně p-dimenzioální prostor $\mathbb{R}^p$, kterému se říká hyperkoule (v angl. ball). Každá taková koule bude obsahovat alespoň dva body. Hyperkoule k je tedy vlastně množina, která sdružuje blízké body. Každá takováto hyperkoule může mít právě dva syny nebo žádného syna, jejímž sjednocení je získána množina otcovské koule. Rekurzivním způsobem lze vytvořit binární strom, kde zanoření bude pokračovat dokud není splněna předem zadaná podmínka, zpravidla maximální mohutnost každého listového uzlu [18, 4, 15].

Sestrojení struktury

Existuje mnoho způsobů jakými lze sestavit balltree strukturu [18, 4]:

Nechť existuje množina bodů $S \subseteq \mathbb{R}^p$ v prostoru. Dále pak existuje konstanta M_{max}, která specifikuje cílovou mohutnost listových hyperkoulí.

1. Je vybrán náhodný bod r z množiny S.
2. Je nalezen nejvzdálenější bod od náhodného bodu r, f_1 z množiny bodů S.

[7] https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote16.html
[8] Pro $p = 2$ se jedná o kružnici, pro $p = 3$ se jedná o kouli.

3. Je nalezen bod f_2, který je nejvzdálenější od f_1.

4. Všechny body jsou promítnuty na přímku danou bodem f_1 a f_2, vznikne tedy nová množina bodů Z.

5. Nalezení mediánu z množiny Z bodů m.

6. Získání množin S_r a S_l kde $S_l = \{x \in Z | z < m\}$ a $S_r = \{x \in Z | z \geq m\}$

7. Pokud $|S_r| > M$ je tento algoritmus spuštěn na S_r je pravým synem S. Pokud $|S_l| > M$ je tento algoritmus spuštěn na S_l je levým synem S.

8. Pokud S_l a S_r splňuje maximální mohutnost, tak strom je kompletně vystavěn.

Pro získání středu koule, je třeba spočítat průměrnou hodnotu všech bodů v množině S. Průměr kruhu lze získat určením středu a spočítáním nejvzdálenějšího bodu od středu koule, čímž se získá poloměr.

Na obrázku 2.12 je znázorněna konstrukce balltree v několika klíčových krocích a na obrázku 2.13 lze vidět vystavěný strom.

Vyhledávání bodu

Pro bod k, bude nalezení nejbližšího souseda probíhat následovně [4]:

1. Na aktuální úrovni stromu jsou vypočteny vzdálenosti d_l a d_r mezi k a středy koulí S_l a S_r.

2. Je vybrána ta menší vzdálenosti z d_l a d_r a pokud odpovídají koule S_i ($i \in \{r, l\}$) není listovou, tak algoritmus pokračuje nad množinou S_i.

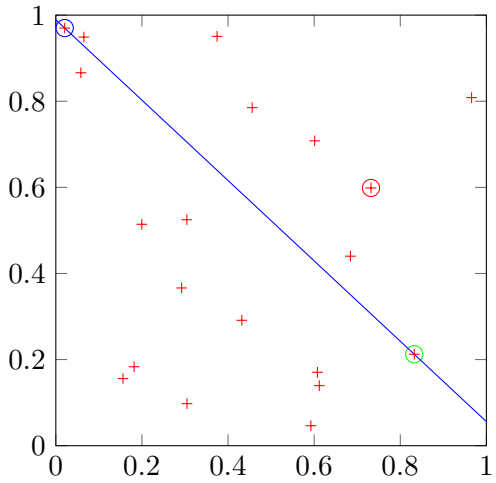
3. V opačném případě je z množiny S_i vypočítán bod $x \in S_i$, jenž má nejmenší vzdálenost mezi x a k. Bod x je nejbližším sousedem bodu k. Pro nalezení k-tého nejbližšího souseda je nalezen k-tý nejvzdálenější bod v listové kouli. Pokud je k větší jak mohutnost listové koule, tak je potřeba procházet body v otcovské kouli.

Výhody a nevýhody

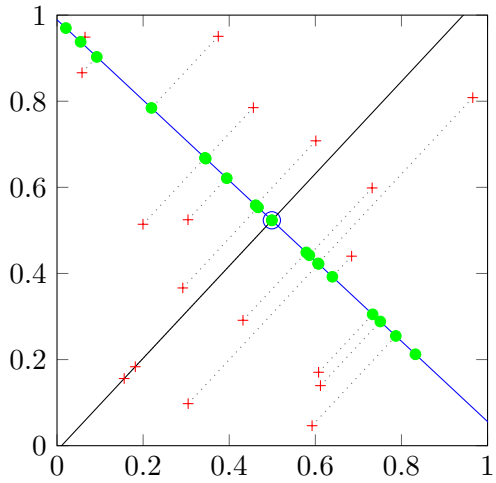
Velkou výhodou je, že struktura se nemění s rostoucím množstvím dimenzí p. Pro nižší počet (např. $p = 3$) dimenzí je balltree pomalejší než např. KD-tree, ale u mnoho dimenzionálních prostor je balltree rychlejší [2]. Balltree je rovněž zatížen prokletím dimensionality.

Další prostorové struktury

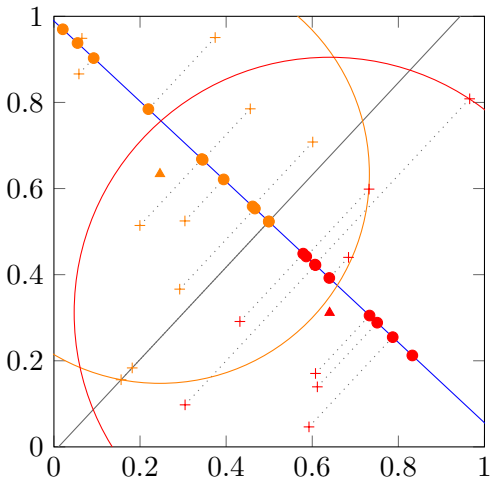
Mimo zmíněné struktury existuje i mnoho dalších struktur, které mohou, ale i nemusejí být založeny na výše jmenovaných. Některé různými způsoby pozměňují výše jmenované, jako například přidání různých heuristik aplikovaných na výstavbu. Struktury je možné optimalizovat na paměťovou náročnost, dobu potřebnou k výstavbě struktury, dobu potřebnou pro vyhledání prvního, ale obecně jakéhokoliv souseda. Dále pak může být struktura zaměřena na mnoho dimenzionálních prostor, kde se řeší efektivita výstavby, ale i vyhledávání v ní [3].



(a) 20 bodů je umístěno do prostoru. Je vybrán jeden náhodný bod (označený červeným kroužkem) a k němu je nalezen nejvzdálenější f_1 (označený modrý kroužkem) a k němu je nalezen nejvzdálenější bod f_2 (označený zeleným kroužkem). Vznikne přímka dána body f_1 a f_2.

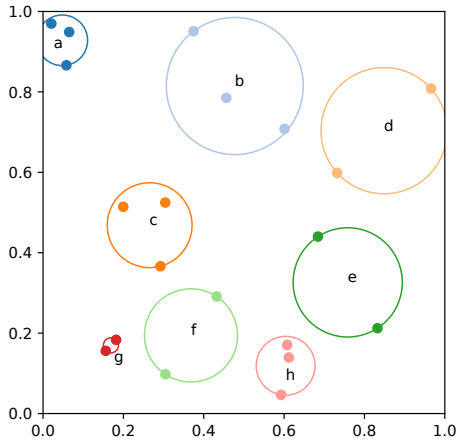


(b) Body jsou promítnuty na přímku (zelené body) a vznikne množina bodů Z. Z množiny Z je určen medián m (označený modrým kroužkem).

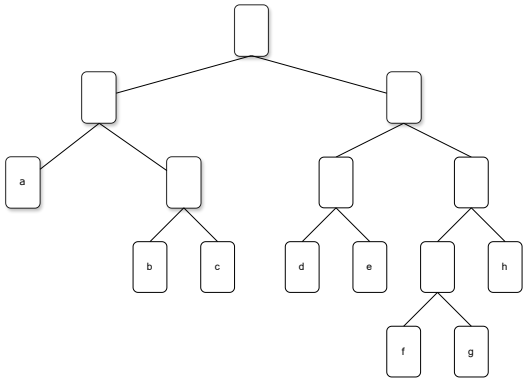


(c) Body jsou rozděleny do množiny S_l (oranžové „+“) a S_r (červené „+“), podle toho, zda jejich promítnutí je větší nebo menší než medián. Vzniknou dvě koule (kružnice) obepínající všechny body dvou množin (červené a oranžové kružnice).

Obrázek 2.12: Znázornění konstrukce balltree pro $p = 2$.



(a) Grafické znázornění listových uzlů vybudovaného stromu pro $m = 3$.



(b) Grafické znázornění stromu balltree.

Obrázek 2.13: Grafické znázornění balltree, jak v prostorovém zobrazení (pouze listové uzly), tak ve formě stromového diagramu.

2.5 Optimalizace na strojové úrovni

V prezentovaných strukturách je často potřeba vypočítávat hodnotu podle nějakého konkrétního vztahu hned pro několik hodnot. V těchto případech se jedná o smyčku, která se postupně vykonává nad všemi daty, u kterých to je potřeba. Moderní procesory však umožňují používat instrukce typu *Single Instruction Multiple Data* (SIMD) česky, jedna instrukce více dat, což jsou instrukce rozšiřující možnosti procesoru o další instrukce, které umožňují některé operace vykonávat najednou paralelně nad více daty. Obvykle přidávají i další registry, aby byla práce s více daty vůbec možná. Existuje více instrukčních sad, které umožňují různé a různě datově široké operace, jako například SSE a různé verze AVX. Například na architektuře ARM se mluví o SVE, Helium nebo Neon [12, 13, 14]. Porovnání těchto architektur je v tabulce 2.1.

Vektorizace

Jak je uvedeno výše, tak existuje mnoho instrukčních sad, které mezi sebou nemusejí být kompatibilní. Stěžejní je velikost registrů, se kterou dokážou pracovat a operace co dokážou vykonávat.

Vektorizace pomocí překladače

Dnešní překladače jsou natolik inteligentní, že samy o sobě provádějí vektorizaci pokud dokážou detekovat, že to má smysl a že to jde. Obvykle však překladače neprovádějí tuto optimalizaci v základním nastavení. Je třeba si jej vynutit přepínačem jako například u překladače *GCC* (GNU Compiler Collection) pomocí:

- -ftree-vectorize: tento flag povoluje vektorizaci kódu.
- -march=native: tento flag informuje překladač o cílové instrukční sadě procesoru.

- -ffast-math: tento flag povoluje agresivnější optimalizace, které mohou zahrnovat vektorizaci.
- souhrný přepínač *-o3*, který aktivuje další optimalizace.

Bohužel, překladače neumí vektorizovat všechno. Obecně se dá říci, že čím složitější kód, tím je menší šance, že překladač rozpozná možnost k vektorizaci. Programátor může překladači pomoci a dát mu tzv. nápovědy přímo do kódu např. ohledně aliasingu[9], očekávaného počtu opakování smyčky, zarovnání v paměti nebo zakázání vektorizace pro nějakou sekci [22].

Ruční vektorizace

Pokud je kód více komplexní nebo programátor nechce spoléhat na překladač a nebo věří, že dokáže vektorizaci udělat lépe, je možné provést vektorizaci ručně [23], což může být docela náročná činnost. Nejnáročnější může být příprava dat. Data je potřeba správně připravit do správných registrů příslušné velikosti. Je potřeba mít na paměti, že tato příprava dat musí být co možná nejefektivnější.

Je tedy logické a vhodné, aby kód byl strukturován už s myšlenkou na to, že bude použita optimalizace vektorizací. Bohužel často může dojít k tomu, že je potřeba implementovat, jak SIMD verzi výpočtu tak i sériovou verzi, protože obvykle je potřeba, aby byly hodnoty se kterými se počítá násobkem osmi, čtyř podle typů hodnot a verze SIMD. U AVX s 256 bitovými registry a počítání s 4 bytovými floaty je nutné, aby počet hodnot odpovídal násobkům osmi.

Pro příklad je možné si představit pole hodnot, kde je potřeba vynásobit konstantou každou hodnotu. V sériovém provedení bude for cyklus procházet prvek po prvku a každou hodnotu vynásobí, což jde vidět na kódu 2.1.

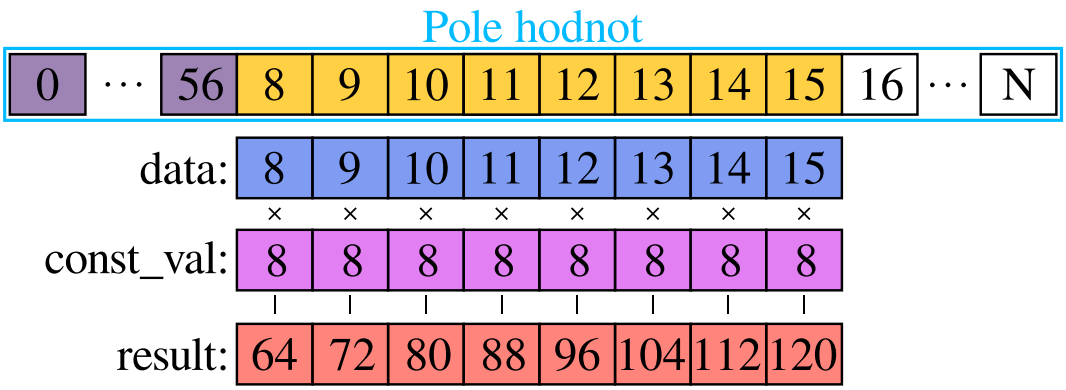
U implementace používající AVX bude potřeba si připravit registr, kde bude mít osmkrát vedle sebe konstantu, kterou se budou násobit všechny prvky. Dále pak je nutné zkopírovat prvky, které je potřeba vynásobit. A následně je potřeba hodnoty zkopírovat zpět do pole odkud jsou data brána. V jeden okamžik je vypočítáno osm hodnot. Je tedy teoreticky 8x rychlejší. Násobení je znázorněno na obrázku 2.14. V realitě tomu tak obvykle však není kvůli přípravám. Další zpomaleni může nastat pokud vstupní pole neobsahuje násobek osmi prvků. S tím je možné se vypořádat tzv. „dokročením“ sériovou alternativou nebo kopírováním 8 míst z paměti (i když už neobsahují validní data) a následně zkopírovat do výsledku pouze hodnoty, které nesou význam. Jak by mohlo takové násobení pole konstantou vypadat, je možné vidět v kódu 2.2.

Vhodné je, aby struktury byly uloženy jako struktury polí, nikoliv jako pole struktur. Pokud tomu tak není, může to vést k značnému zpomalení, jenž může dosti zneefektivnit takto naimplementovaný výpočet.

Je zjevné, že u některých algoritmů není možné nebo není výhodné použít SIMD zpracování dat. Může to být buď způsobeno nekompatibilitou algoritmu nebo nepřipraveným návrhem uložení dat pro tento styl zpracování.

```
void multiplyByConst(float *f, float c, int n){
    for (size_t i = 0; i < n; i++){
        f[i] = f[i]*c;
    }
```

[9] Označení pro situaci, kdy mají respektive nemají dva ukazatele plně nebo částečně stejné cíle [9].



Obrázek 2.14: Ilustrace fungování násobení velkého pole konstantou 8. Na obrázku je vidět vstupní pole, které zároveň slouží jako výstupní. Pole obsahuje čísla od nuly po N uspořádaná zleva doprava. Na obrázku je znázorněna situace, kdy prvních 8 prvků pole bylo již vynásobeno. Zkopírované prvky (označené žlutou barvou) se nacházejí v registru pojmenovaném „data“, který je označen modrou barvou. Další registr, „const_val“ (označený růžovou barvou), obsahuje osmkrát opakovanou konstantu 8. Ve finálním registru „result“ je uložen výsledek násobení (znázorněn červenou barvou). Tento výsledek by poté byl kopírován zpět do původního pole, do žlutě označené sekce pole. Proces násobení by následně pokračoval posunem o 8 prvků doprava, tedy na hodnoty od 16 do 23.

```
}
```

Výpis 2.1: Ukázka funkce pro násobení pole konstantou v jazce C.

```
void multiplyByConstAVX(float *f, float c, int n){
    __m256 const_val = _mm256_set1_ps(c);
    size_t i;
    for (i = 0; i + 7 < n; i += 8) {
        __m256 data = _mm256_loadu_ps(&f[i]);
        __m256 result = _mm256_mul_ps(data, const_val);
        _mm256_storeu_ps(&f[i], result);
    }
    multiplyByConst(f+i, c, n-i);

}
```

Výpis 2.2: Ukázka funkce pro násobení pole konstantou pomocí AVX instrukcí. Ilustrace takového násobení je znázorněna na obrázku 2.14.

Parametr	**AVX/AVX2**	**ARM Helium**	**AVX-512**	**ARM Neon**
Šířka vektoru	256-bit	128-bit	512-bit	64-bit až 128-bit
Počet registrů	16 YMM registrů	32	32 ZMM registrů	16-32
Datové typy	Int (8,16,32,64), Float, Double	Int, Float,	Int (8,16,32,64), Float(32,16), Double	Int (8,16,32,64), Float
Architektura	x86_64	ARMv8.6-A a novější	x86_64	ARMv7 a novější

Tabulka 2.1: Srovnání SIMD instrukcí: AVX/AVX2, ARM Helium, AVX-512 a ARM Neon.

Kapitola 3

Koncept struktury řešící problém nejbližšího souseda

Tato práce reimplementuje verzi algoritmu nejbližšího souseda, popsanou v práci „Almost constant-time 3D nearest-neighbor lookup using implicit octrees“ [5]. Zmíněná implementace kombinuje použití struktur octree a hašovací tabulky spolu s využitím vlastností Voronoi diagramu s tím, že tato implementace je rozšířena pro vyhledávání orientovaných bodů.

Octree je vystavěný klasickým způsobem popsaným v sekci 2.4 nad Voronoi diagramem při dodržení kriteria pro maximální množství zasahujících Voronoi buněk do jednoho oktetu. Zde přichází ta nejzajímavější část této struktury. Postupně jsou procházeny jednotlivý uzly vystavěného stromu a ty jsou umístěny do hašovací tabulky.

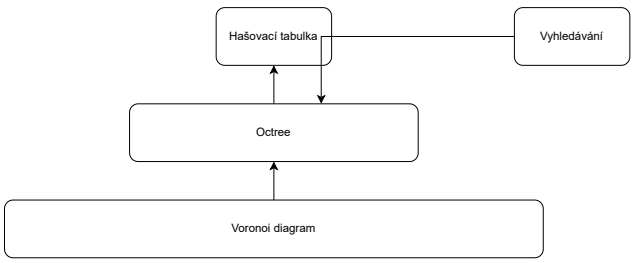
3.1 Model struktury

Struktura se staví ve 3 fázích, které je možné vidět na obrázku 3.1. Fáze probíhají postupně a po vybudování celé struktury se interaguje pouze s hašovací tabulkou a skrz ní se stromem.

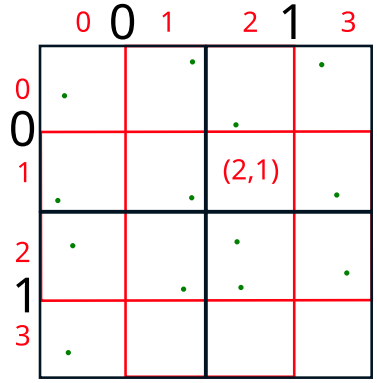
Index hašovací tabulky

Index se skládá ze dvou částí. První částí je identifikace sektoru. Identifikace sektoru může být trojice[1], kde každá složka odpovídá souřadnici v jedné dimenzi. Druhou složkou je úroveň ve stromu. Tato dvojice jednoznačně identifikuje jakýkoliv uzel ve stromu. To jakým

[1] Pro $p = 3$ pro $p = 2$ stačí dvojice.



Obrázek 3.1: Ilustrační schéma prezentované struktury. Na první úrovni je vystavěn Voronoi diagram, nad ním je vystavěn octree a nad ním je vystavěna hašovací tabulka.



Obrázek 3.2: Identifikace sektorů v quadtree (pro octree by situace vypadala podobně, jen by indexy nebyly dvojice ale trojice.), černě jsou znázorněny indexy na vyšší úrovni. Červeně pak ty na nižší úrovni. Na obrázku lze vidět i dokonce, jak by byl indexován jeden z červených sektorů.

způsobem funguje souřadnice je zachyceno na obrázcích 3.2 a 3.3, na nichž je znázorněn vztah mezi hašovací tabulkou, indexy v ní a octree.

Úroveň stromu určuje limity, které souřadnice mají, přesněji řečeno od nuly[2] po $2^l - 1$ je maximální hodnota, která se může objevit, kde l je úroveň.

Složitost

Pokud by vyhledávání probíhalo jen ve stromu, byla by složitost logaritmická. Nicméně prezentovaná struktura nad tímto stromem vystaví hašovací tabulku, ve které probíhá binární vyhledávání, čímž se složitost sníží na logaritmicky logaritmickou [5].

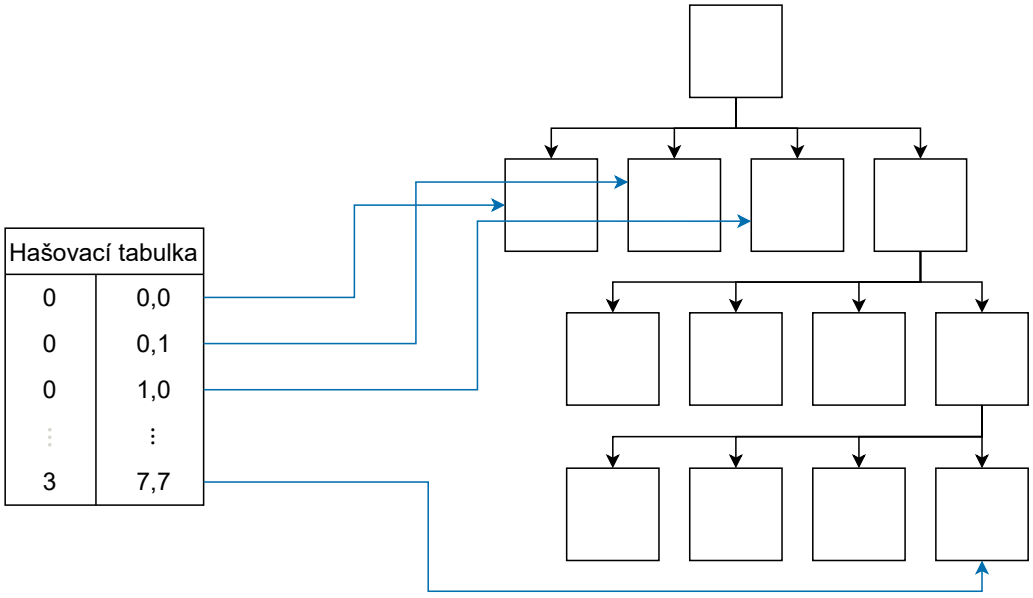
Vyhledávání

Vyhledávání využívá princip binárního vyhledávání, které má logaritmickou složitost. Proces nalezení zahrnuje binární prohledávání prostoru určeného souřadnicemi uzlu v hašovací tabulce, což umožňuje efektivně získat přístup k odpovídajícím uzlům. Tento přístup je efektivní v kombinaci s hašovací tabulkou, která umožňuje rychlé získání dat bez nutnosti procházet celý strom.

Nechť p je vyhledávaným bodem, $l_{\min}$ je minimální úroveň ve stromu a $l_{\max}$ je maximální úroveň. Potom existuje i l_c. Dále pak existuje hašovací tabulka H, která pro index vrací buď null v případě, že neexistuje prvek pod takovým indexem, nebo vrátí uzel stromu. Potom také existuje funkce find_idx(), která pro vstupní bod a úroveň ve stromu vrátí souřadnice sektoru, ve kterém se bod p nachází.

1. Pokud $l_{\max} - l_{\min}$ je roven 1, vyhledávání se zastaví a nejbližší soused se nachází v $l_{\max}$. Jinak se pokračuje dále.
2. $l_c = \frac{l_{\max} - l_{\min}}{2}$, idx = find_idx$(l_c, p)$.
3. Pokud $H(l_c, \text{idx})$ je roven null, vyhledávání pokračuje s $l_{\max} = l_c$ a $l_{\min}$ zůstane nezměněný. V opačném případě vyhledávání pokračuje s $l_{\min} = l_c$ a $l_{\max}$.

[2] Předpokládá se indexování od nuly.



Obrázek 3.3: Zobrazení souvislosti mezi hašovací tabulkou, indexy a octree (quadtree). Modrou barvou jsou znázorněny odkazy z hašovací tabulky do quadtree. Indexy hašovací tabulky jsou ve dvou sloupcích, kde první sloupec je úroveň znázornění a druhý je identifikátor sektoru (viz obrázek 3.2). V obrázku se nachází jen část podstromu a hašovací tabulky.

Pomocí tohoto způsobu je nalezen uzel, ve kterém se nachází nejbližší soused. Nalezení nejbližšího souseda je už jen otázkou iterace skrz všechny prvky tohoto uzlu.

Vyhledání k-nejbližších sousedů

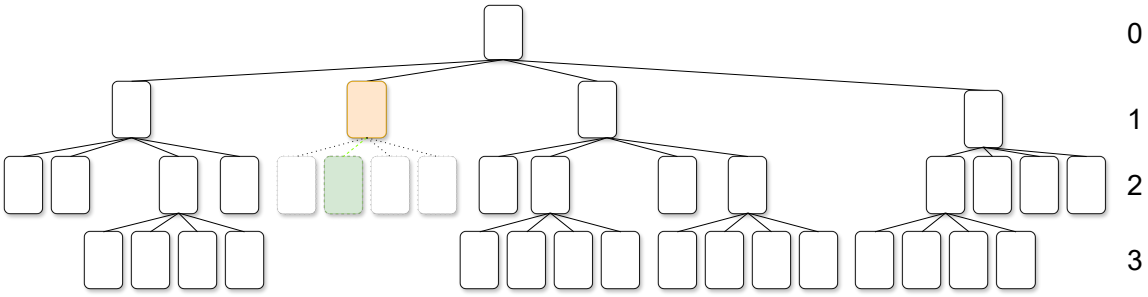
Algoritmus pro nalezení k-nejbližšího souseda je velmi podobný jako u nejbližšího souseda. Po nalezení uzlu hašovací tabulkou jsou prvky seřazeny a je vrácena seřazená sekvence. Pokud je potřeba víc prvků než je v uzlu, algoritmus provede stejný postup nad otcovským uzlem.

Vyhledávání podle normály

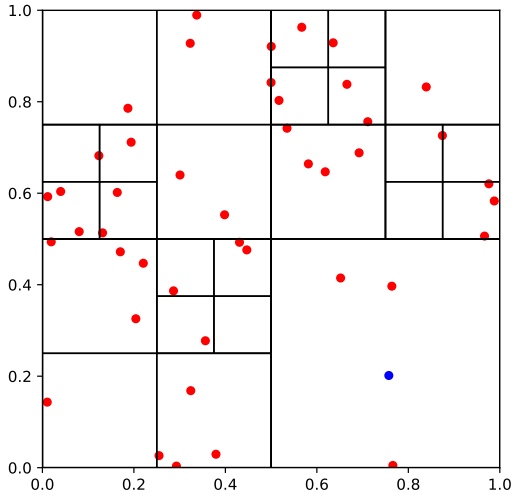
Vyhledání je typicky omezeno vzdáleností. Po určení uzlu hašovací tabulkou je nalezen nejvzdálenější bod v seznamu bodů. Pokud je maximální vzdálenost menší než nastavená maximální vzdálenost, a pokračuje se ve vyhledávání nejvzdálenějšího bodu v otcovském uzlu. Je nutné pokračovat dokud není nalezen prvek, co je dál než je požadováno. V takovém případě je v tomto uzlu i nejvzdálenější prvek, který je potřeba vzít v potaz. Nejvzdálenější prvek splňujícím zadání je nalezen a určí množinu prvků. Z této množiny je vybrán ten s nejpodobnější normálou.

Orientovaný bod

Tato práce implementuje bod jako orientovanou strukturu. Takový bod krom své souřadnice obsahuje i další trojici, která určuje jeho směr. Tento směr je jednotkový vektor a slouží pouze k určení směru. Pomocí goniometrie je možné porovnávat takové směry.



(a) Grafické znázornění stromu. Přerušovaně jsou znázorněny neexistující uzly, jejichž neexistence ovlivňuje vyhledávání (ten hlavní je znázorněn zelenou barvou). Oranžovou barvou je znázorněn uzel, ve kterém se nachází nejbližší soused. Po pravé straně jsou k nalezení úrovně stromu.



(b) Grafické znázornění stromu v podobně diagramu. K modrému bodu je vyhledáván nejbližší soused.

Obrázek 3.4: Ilustrační obrázek demonstrující vyhledávání.

3.2 Konkrétní příklad vyhledávání

Nechť existuje bod E, pro který je potřeba nalézt nejbližšího souseda. Předpokládá se existence vybudované struktury tak, jak je na obrázku 3.4, ve které se bude vyhledávat.

1. $l_{\min} = 0$ a $l_{\max} = 3$, tím pádem $l_c = 2$. V hašovací tabulce se pokusí nalézt uzel pod indexem $(2, \text{find\_idx}(E, 2))$, což vyústí v navrácení hodnoty null, tento neexistující uzel je znázorněn ve stromu zelenou barvou.

2. $l_{\min} = 0$ a $l_{\max} = 2$, tím pádem $l_c = 1$. V hašovací tabulce se pokusí nalézt uzel pod indexem $(1, \text{find\_idx}(E, 1))$, což skončí úspěchem v podobě nalezeného uzlu (v obrázku 3.4 znázorněno oranžovou).

3. $l_{\max} - l_{\min} = 1$, nejbližší soused se nachází v tomto uzlu.

4. Jsou postupně otestovány všechny prvky na vzdálenost k E a tím je nalezen skutečně nejbližší soused.

3.3 Testovací kritéria

Pro účely ověření funkčnosti a rychlosti struktury je potřeba ji podrobit testům. V této kapitole jsou definovány metriky, které byly při testech aplikovány a zkoumány.

Testování přesnosti

Pro testování přesnosti byla vytvořena co možná nejjednodušší implementace, která využívá počítání hrubou silou. Ta dostane jako vstup množinu R náhodných bodů pro množinu sousedů a T náhodných bodů jako množinu testovaných bodů. Dále pak se předpokládá existence K, což je počet sousedů, který bude implementovaný algoritmus vyhledávat. Následně budou vypočítány seznamy pro každý z testovaných bodů obsahující seřazené posloupnosti vzhledem ke vzdálenosti k jednomu bodu z T. Výpočet for cyklem prochází body z testovací množiny a pomocí *std::sort()* ze standardní knihovny C++ je seřazen podle vzdálenosti k testovanému bodu a seznam je uložen do souboru tak, aby nebylo potřeba jej počítat pro každý test. Tímto způsobem budou spočítány seřazené posloupnosti pro všechny body z T.

V další fázi se jen porovnají prvky vygenerované z naprogramované implementace a implementace hrubou silou. Je potřeba mít na paměti, že všechny body včetně vzdáleností jsou počítány na přesnosti datového typu float. V případě, že prvky z množin jsou rozdílné, tak jsou ještě porovnány vzdálenosti k testovanému bodu (na přesnosti float). Pokud jsou stejné, předpokládá se, že algoritmus pracuje správně.

První nejbližší soused

Pro kontrolu fungování nejbližšího souseda bude využita vypočtená množina tak, jak je popsaná výše s tím, že se porovnává jen první prvek.

K-tý nejbližší soused

Ačkoliv autoři původního článku vůbec tuto strukturu netestují pro k-NN, tak by struktura měla fungovat i pro tento případ. Pro ověření bude využita množina tak, jak je popsána výše.

První nejbližší soused s nejpodobnější normálou

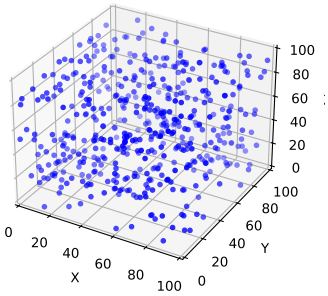
Pro vyhledávání podle normály bude použit podobný přístup. S tím rozdílem, že bude vypočítáno jen nejlepších 10 bodů. Těchto 10 bodů se spočítá hrubou silou. Jedná se o 10 nejlépe orientovaných z množiny bodů s fixní vzdáleností.

Výstupy testů

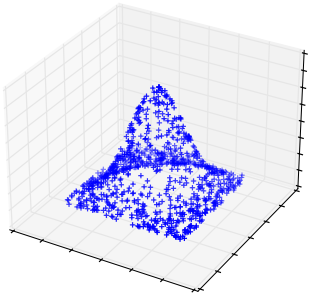
Ve všech testech se bude sledovat množství úspěšných vyhledání (nalezen stejný prvek jako v kontrolní implementaci) a v případě, že vyhledání nalezne jiný bod než ten správný podle kontrolní implementace, tak je spočten rozdíl vzdáleností od hledaného bodu a ten je zprůměrován pro získání průměrné odchylky, kterou struktura vytváří.

Testování výkonu

Při testování struktury určené k vyhledávání nejbližšího souseda je potřeba sledovat následující metriky:

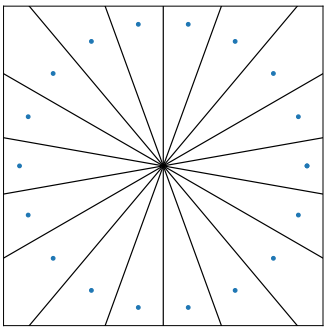


(a) Náhodné rozdělení bodů.

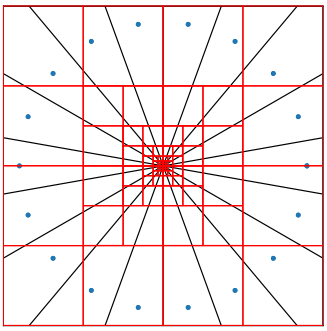


(b) Plocha tvořena body.

Obrázek 3.5: Ukázka náhodného rozdělení bodů použitých v testech. Charakteristiky jsou převzaty ze [5].



(a) Body uspořádané do kruhu a Voronoi diagram nad nimi.



(b) Rekurzivní dělení se nikdy nezastaví protože, podmínka je množství zasahujících Voronoi buněk do čtverce, což u čtverců v blízkosti středu nemusí nikdy nastanou.

Obrázek 3.6: Ukázka nevýhodného rozdělení bodů v $p = 2$ prostoru. Všechny Voronoi buňky mají jedno místo, kde se dotýkají.

1. Doba výstavby stromu.
2. Množství potřebných uzlů stromu.
3. Doba potřebná pro vyhledání prvního souseda a vyhledání n-tého souseda.
4. Kriterium M_{max}, tedy kolik Voronoi buněk maximálně zasahuje do uzlu.

Následující metriky bude potřeba sledovat v několika různých podmínkách, především pak vstupní rozdělení bodů. Výkon algoritmů nejbližšího souseda může značně ovlivnit právě rozdělení bodů. Jmenovitě: rovnoměrné náhodné rozdělení a body tvořící plochy.

Degenerativní případy

Autoři studie zmiňují, že struktura může mít problémy se zpracováním oblastí, které vytvářejí Voronoi buňky tak nešťastně, že spolu sousedí hned několik v jednom místě. Jak to může vypadat lze vidět na obrázku 3.6. Tyto případy nebudou testovány, protože nejsou pro tuto strukturu vhodné.

Kapitola 4

Implementace rychlé varianty algoritmu nejbližšího souseda

Výstavba struktury je rozdělena do několika fází, které na sobě závisí. Výsledkem je struktura skládající se z octree a hašovací tabulky, která se odkazuje na uzly této tabulky.

4.1 Struktury použité v implementaci

Přehled vybraných klíčových použitých struktur při implementaci je možné vidět na obrázku 4.1.

OctreeNode

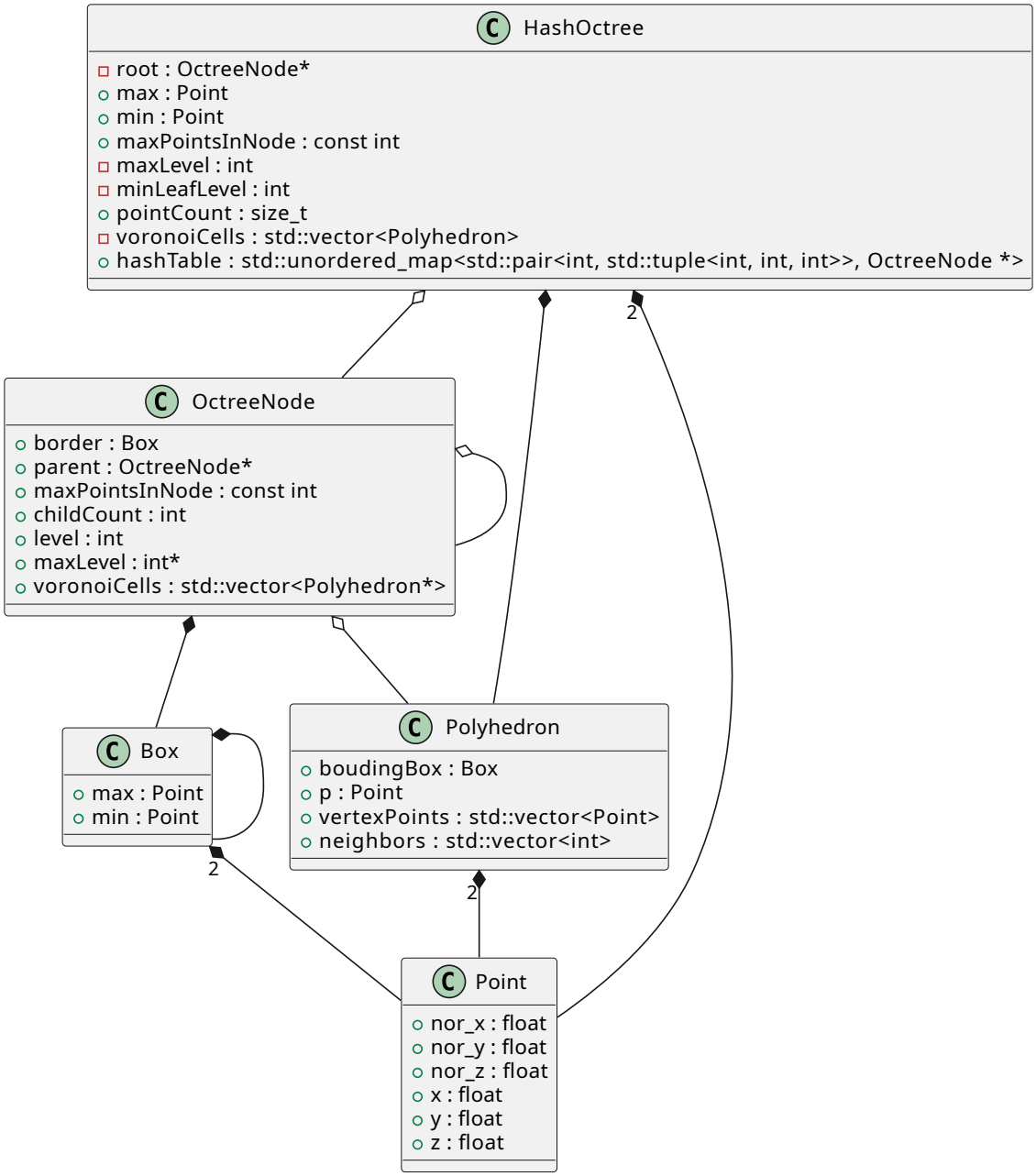
Tato struktura slouží k skladování informací a potomků uzlů stromu. Každý uzel může mít až osm poduzlů, které jsou skladovány v poli ukazatelů, kde *null* indikuje neexistenci syna. Každý uzel má odkaz na svého otce, což usnadňuje navigaci ve stromu od listových uzlů. Důležité je také pole skladující všechny Voronoi buňky zasahující do prostoru vymezujícího tento uzel. V neposlední řadě jsou zde čísla ukládající informace používané při výstavbě stromu, hašovací tabulky nebo při vyhledávání. Vše je shrnuto v obrázku 4.1 včetně závislosti na jiných třídách.

Hashtree

Třída Hashtree obaluje celou strukturu. Obsahuje všechny Voronoi buňky, na které se odkazují všechny uzly stromu. Dále obsahuje kořenový uzel struktury, hašovací tabulku a další.

Polyhedron

Třída reprezentuje mnohostěn a protože Voronoi buňky jsou mnohostěny, jsou použity pro jejich reprezentaci. Obsahuje seznam vrcholů a středový bod, který určuje tento mnohostěn (Voronoi buňku). Údaje pro konstrukci jsou získány ze struktury vypočítané pomocí knihovny voro++ (viz sekce zabývající se Voronoi diagramem 4.2), protože struktura z voro++ nese zbytečně mnoho dalších informací, které jsou irelevantní pro použití v této aplikaci.



Obrázek 4.1: UML diagram klíčových členů, klíčových struktur.

Box

Box je jednoduchá struktura pro ukládání krychlí. Je definována dvěma souřadnicemi. Klíčové u této třídy jsou především metody zpřehledňující implementaci.

Point

Struktura pro skladování bodu společně s normálou. Je implementována pro zpřehlednění zápisu.

4.2 Implementace výstavby stromu

Veškerá implementace je skryta pod třídou *Hashtree*. Hashtree reprezentuje kompletní strukturu jako celek a nabízí metody pro vyhledávání. Všechny body jsou nahrány najednou jako vstupní pole hodnot, třída mimo to očekává hranice, kde se body nacházejí, tedy vlastně krychli ohraničující všechny body. Další parametr konstruktoru je počet jader, který má být použit pro výstavbu a v neposlední řadě i cílové kritérium, které určuje maximální množství Voronoi buněk, zasahujících do listového uzlu.

Konstruktor potom přistoupí k první fázi a to je výpočet Voronoi diagramu. Druhou fází je stavba octree nad Voronoi diagramem. Poslední fází je umístění prvků do hašovací tabulky.

Voronoi diagram

Pro výstavbu octree je nezbytné vypočítat Voronoi diagram. Pro výpočet Voronoi diagramu je využitá knihovna voro++ [20, 16], která umožňuje počítat Voronoi rozdělení paralelně na CPU. Knihovna nejdříve vyžaduje inicializaci prostoru podle krychle ohraničující prostor a dále vyžaduje body na kterých vytváří diagram. Následně jsou ve smyčce vypočítány všechny Voronoi buňky, ze kterých jsou extrahovány jen ty důležité informace do struktury *Polyhedron*.

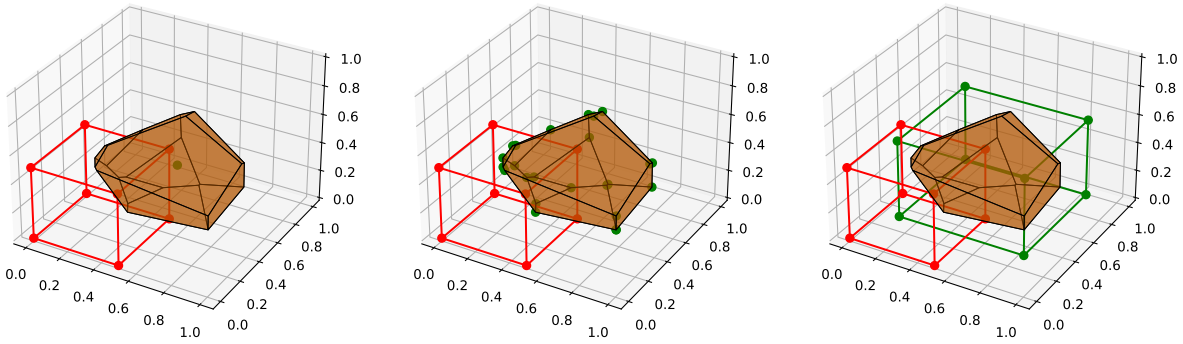
Autoři původní implementace nezmiňují, jakým způsobem počítají Voronoi buňky. Vzhledem k časů uvedených ve studii potřebných pro stavbu struktury, ale pravděpodobně používají nějakou přesnější metodu výpočtu. Knihovna voro++, počítá voronoi buňky na datovém typu double, ten je však přetypován při vkládání do struktury *Polyhedron* na float kvůli rychlejším výpočtům později.

Výstavba stromu

Pro výstavbu stromu je implementovaná speciální struktura *OctreeNode*, která reprezentuje jeden uzel stromu. Po dokončení výpočtu Voronoi buněk je vytvořen kořenový uzel, jenž dostane na vstupu všechny mnohostěny a meze celého zadaného prostoru. Rekurzivním přístupem je vybudován strom, který o každém z mnohostěnů rozhodne, zda se nachází v právě prozkoumávané krychli[1]. Při rozdělení prostoru na osminy jsou zkontrolovány všechny Voronoi buňky, které náležely otci. Rekurzivní výstavba stromu bude pokračovat až do okamžiku, kdy počet Voronoi buněk zasahujících do sledované krychle nebude menší nebo roven jako stanovené maximum.

Nejnáročnější část výstavby stromu je kontrola náležitostí mnohostěnu do uzlu stromu. V implementaci se kombinuje několik přístupů. Využívá se toho, že detekovat, zda je nějaký

[1] Zkoumají se jen ty mnohostěny, co patřily do nadřazeného uzlu.



(a) Kontrola středového bodu Voronoi buňky zda se nachází uvnitř krychle.

(b) Kontrola zda jeden z vrcholů Voronoi buňky je uvnitř krychle. Vrcholy Voronoi buňky jsou znázorněny zelenou barvou.

(c) Kontrola zda jeden z vrcholů blending boxu Voronoi buňky je uvnitř krychle. Bounding box je znázorněn zeleně.

Obrázek 4.2: Grafické znázornění různých přístupů při zjišťovaní kolizí. Červenou barvou je znázorněna právě kontrolovaná krychle.

bod konvexního tvaru uvnitř krychle, je snadné. Je zjevné, že pokud alespoň jeden bod konvexního tvaru je uvnitř krychle, musí to nutně znamenat, že ona krychle a mnohostěn jsou v kolizi (obrázek 4.2b). Poslední detekční mechanismus je algoritmus GJK, který se však používá pouze, pokud bounding box zkoumané Voronoi buňky je v kolizi se zkoumanou krychlí. Postup je znázorněn algoritmem 1 a taktéž je možné jednotlivé přístupy vidět na obrázku 4.2.

Nechť existuje funkce *isInside()*, která má na vstupu krychli a dále pak množinu nebo jeden bod a vrátí *true*, pokud alespoň jeden z bodů nebo samotný bod je uvnitř dané krychle. Algoritmus použitý v implementaci vypadá takto:

Algorithm 1 Detekce kolize mezi krychlí a mnohostěnem

Input: Množina vrcholů mnohostěnu M, středový bod mnohostěnu S, vrcholy krychle K
Output: True, pokud je detekována kolize, jinak False

```
if isInside(K, S) then
    return True
end
if isInside(K, M) then
    return True
end
if isInside(K, boundingBox(M)) then
    return GJK(M, K)
end
return False
```

Tímto způsobem je vystavěn celý octree. Sledovanou vlastností je množství Voronoi buněk zasahujících do právě řešené krychle. Výstavba je paralelizována maximálně na osmi vláknech, kdy se 8 podstromů kořenového uzlu staví současně.

GJK algoritmus

Implementace GJK algoritmu je převzata z webové stránky[2]. Implementace je optimalizována pro odpověď ano/ne a upravena tak, aby co nejlépe odpovídala strukturám použitým v této práci.

Hašovací tabulka

Pro implementaci hašovací tabulky je využita implementace *std::unordered_map* ze standardní knihovny C++. Této implementaci stačí poskytnout hašovací funkci, která umožní zahašovat typ použitý pro indexování. V prezentované implementaci se používá dvojice tvořená celým číslem a trojicí. Po dokončení výstavby stromu jsou do hašovací tabulky umístěny všechny uzly stromu.

Výstavba stromu je paralelizována a probíhá paralelně, kdy každé vlákno má svou hašovací tabulku a zabezpečuje vkládání prvků z určené části stromu. Následně je použita metoda umožnující spojování hašovacích tabulek *merge*.

Implementace rychlého výpočtu souřadnice

Nechť existuje 3D prostor ohraničený ve všech směrech od 0 do 1[3]. Úkolem je nalezení diskrétní souřadnice[4] zadané od 0 do 2^l, kde l je parametr. Dále existuje bod p, náležící do mezí prostoru (definovaný nad datovým typem *float*).

Každá složka bodu je vynásobena hodnotou (2^l) a následně je každá ze složek zaokrouhlena dolů na celé číslo[5]. Výsledkem jsou tři čísla dávající souřadnici v každé z dimenzí. Výpočet je demonstrován na konkrétním příkladu pro vstupní bod p a parametr 3:

$$round((0.23, 0.45, 0.99) \times (2^3)) = round((1.84, 3.6, 7.92)) = (1, 3, 7)$$

V implementaci jsou body normalizovány na interval 0 až 1. Tato normalizace je vektorizována společně s výše popsaným postupem pro získání souřadnice.

4.3 Implementace vyhledávání

Vyhledávání probíhá v octree přes hašovací tabulku. Využívá rozhraní nabízející implementace hašovací tabulky.

Nejbližší soused

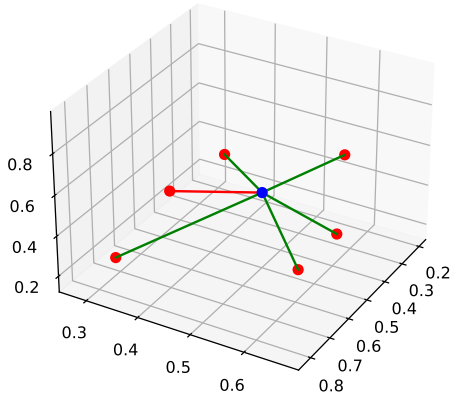
Vyhledávání probíhá ve dvou fázích. První fází je vyhledávání v hašovací tabulce a druhou fází je určení skutečně nejbližšího souseda z bodů ve vyhledaném uzlu.

První fázi lze vidět v algoritmu 2. Souhrnně by se tato fáze dala popsat jako binární vyhledávání v hašovací tabulce. Algoritmus rozdělí prohledávaný prostor na dvě části: spodní a vrchní, rozdělené hranicí l_c. Pokud neexistuje ve stromu uzel na úrovni l_c, je jisté, že nemůže existovat ani na úrovni l_c až l_{max}. Je tedy zřejmé, že vyhledávání se může dále soustředit pouze na interval od l_{min} do l_c. V případě, že takový uzel existuje, je jisté, že bude existovat přinejmenším na úrovni l_c a vyhledávání pokračuje v intervalu od l_c do l_{max}.

[2] https://winter.dev/articles/gjk-algorithm
[3] Body jsou v implementaci normalizovány do tohoto intervalu.
[4] Jedná se o souřadnici v octree zobrazenou např na obrázku 3.2.
[5] V implementaci je použita konverze na datový typ *int*.

Algorithm 2 Algoritmus pro nalezení nejbližšího uzlu stromu.

Input: Vstupní bod p
Output: Uzel obsahující nejbližší bod k p
$l_{min} = 0$, l_{max} = tree.maxDepth(),
while $l_{max} - l_{min} <= 1$ **do**
    $l_c = (l_{max}$ - $l_{min})/2$
    if *hashtable[*l_c*, find_idx(p,*l_c*)] == null* **then**
        $l_{max} = l_c$
    else
        $l_{min} = l_c$
    end
end
return *hashtable[*l_c*, find_idx(p,*l_c*)]*



Obrázek 4.3: Ilustrace vyhledávání nejbližšího souseda z vybraného uzlu stromu. Modrý bod je vstupním bodem. Červené body jsou body uzlu získaného v první fázi. Vzdálenosti jsou porovnány a je vybrána ta nejmenší (znázorněna červeně).

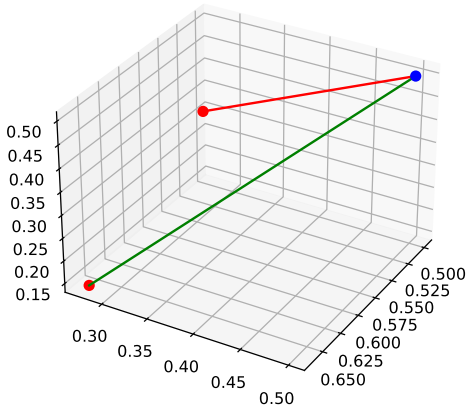
Druhou fází je nalezení nejbližšího souseda ve vyhledaném uzlu. Algoritmus 2 vždy vrátí listový uzel, který bude obsahovat jeden až M_{max} bodů. Iterací přes všechny body je nalezen ten s nejmenší vzdáleností k vstupnímu bodu. Na obrázku 4.3 je vidět ilustrace takového porovnání.

V implementaci je výpočet vzdáleností vektorizován a počítá vzdálenosti po osmi.

K-nejbližší soused

Vyhledávání k-nejbližších sousedů probíhá velmi podobně, jako při hledání prvního souseda. Tentokrát by se hledání dalo rozdělit na tři fáze. První fáze nalezení nejbližšího listového uzlu je stejná jako u hledání nejbližšího souseda (algoritmus 2). V druhé fázi je potřeba ve stromu vystoupat od listových uzlů výše do takového uzlu, který obsahuje alespoň k prvků. Následně stačí prvky seřadit podle vzdálenosti k bodu p.

Řazení prvků je realizováno pomocí *std::sort* ze standardní knihovny C++. Této implementaci je dodána funkce pro porovnání prvků, což je vlastně jen operátor větší než, který porovná vzdálenost k bodu p od dvou vstupních hodnot. Není tedy porovnána hodnota bodů, ale jejich vzdálenost od bodu p. Ilustrace je na obrázku 4.4.



Obrázek 4.4: Ilustrace operátoru porovnání vzdálenosti k bod p (znázorněného modře), a dvou porovnávaných bodů (znázorněno červeně) a jejich vzdáleností (červená a zelená).

Po seřazení je navrácen seznam bodů. Implementace ignoruje, zda se v uzlu nachází více, než požadované množství bodů a vrátí seznam všech, které nalezl ve vybraném uzlu.

Nejbližší soused orientovaný zadaným směrem

Pro maximální výkon se předpokládají následující možnosti použití:

1. První splňující zadaná kriteria.
2. Nejbližší s nejpodobnější normálou.
3. Iterátor se zadanou maximální vzdáleností a vracející postupně ty nejvíc podobné body až po ty méně podobné.

Všechny tyto přístupy budou mít na vstupu jeden bod s normálou. Podle potřeby mohou obsahovat tyto další parametry:

- Míra snažení

  Zadává metodě míru snahy, kterou má při vyhledávání projevit. Tímto parametrem se dá nastavit zda-li má vyhledávání probíhat pouze v listovém uzlu (nastavení 0), nebo v úrovni o jedno větší jak listové případně o dvě a tak dále.

- Tolerance

  Jedná se o číslo na intervalu od -1 do 1. Určuje o jak moc se může lišit normála bodu. Jedná se o cosinus úhlu udávající rozdíl normál.

- Maximální vzdálenost

  Udává maximální vzdálenost pro vyhledávání.

První splňující zadaná kriteria

Implementace tohoto vyhledávání je nejjednodušší, ale zároveň by mělo být nejrychlejší. Předpokládá se však značná nepřesnost. Začátek vyhledávání bude jako u všech popsaných

hledání vyhledáním listového uzlu v hašovací tabulce. Poté budou všechny tyto prvky seřazeny podle vzdálenosti k zadanému bodu. Takto seřazené pole je potřeba projít prvek po prvku a v okamžiku, kdy první bod splňuje kriterium, tak je navrácen jako správné řešení. V případě, že žádný z prvků nedosahuje kvalit požadovaných vstupem, je vrácena ta nejlepší možnost. Celý postup jde vidět v algoritmu 3.

Tato varianta potřebuje na vstupu parametr zadávající maximální chybu a případně volitelný parametr udávající očekávanou snahu.

Algoritmus využívá metodu popsanou v algoritmu 2, pojmenovanou jako find_node. Dále pak existuje metoda *tree_climbing*, která postoupí o zadaný počet úrovni výše ve stromu a vrátí tento uzel. Funkce *sort* řadí prvky podle vzdálenosti k bodu p. Funkce *dotN* počítá skalární součin normál dvou bodů.

Algorithm 3 Algoritmus pro nalezení prvního bodu splňující zadané kriteria.

Input: Vstupní bod p, $effort$, $tollerance$
Output: Aproximace bodu s nejpodobnější normálou

```
node = find_node(p)
node = tree_climbing(node, effort)
sort(node.points, p)
Lsimilarity = −1, Lpoint = ∅
for u ∈ node.points do
    similarity = dotN(u,p)
    if Lsimilarity < similarity then
        if similarity >= (1 - tollerance) then
            return u
        end
        Lsimilarity = similarity
        Lpoint = u
    end
end
return Lpoint
```

Nejbližší soused s nejpodobnější normálou

Tento způsob vyhledávání je velmi podobný jako ten předešlý. Rozdíl je v tom, že vždy projde všechny prvky v zadaném uzlu a najde tu nejlepší možnost. Celý postup je znázorněn v algoritmu 4.

Iterátor se zadanou maximální vzdáleností

V tomto případě je vyhledávání zahájeno metodou, která vytvoří nový objekt speciálního typu. Tento objekt obsahuje informace potřebné k vyhledávání podle normály: potenciálně zajímavé body, bod, ke kterému jsou vyhledáváni sousedé a maximální vzdálenost od bodu. Bod a maximální vzdálenost jsou brány z uživatelského vstupu.

Vyhledávání začíná nalezením listového uzlu. Následně je potřeba procházet strom od listů ke kořenu, dokud nebudou vyfiltrovány všechny body v požadované vzdálenosti. Tento proces je však aproximován. Aproximace využívá toho, že prostor je rozdělen na krychle o jednoduše vypočitatelné velikosti z octree. Velikost se odvíjí od celkové velikosti prostoru S a úrovně zanoření. Pokud je jako listový uzel určen uzel na úrovni l, celý prostor je

Algorithm 4 Algoritmus pro nalezení nejbližší bodu s nejpodobnější normálou.

Input: Vstupní bod p, $effort$,
Output: Aproximovaný bod s nejpodobnější normálou
node = find_node(p)
node = tree_climbing(node, effort)
sort(node.points, p)
$Lsimilarity = -1, Lpoint = \emptyset$
for $u \in node.points$ **do**
    similarity = dotN(u,p)
    if $Lsimilarity < similarity$ **then**
        $Lsimilarity = similarity$
        $Lpoint = u$
    end
end
return $Lpoint$

rozdělen celkem na 8^l kostek. Hrana každé kostky je rozdělena na 2^l dílů. Pokud požadovaná vzdálenost je D, pak $\frac{S}{2^l}$ udává počet dílků, který je potřeba. Z toho plyne, že úroveň lze získat jako $\log_2\left(\frac{S}{D}\right)$, což informuje o tom, jak vysoko je potřeba vystoupat.

Tento postup nevyžaduje iterování skrz body. Do struktury jsou nahrány body a již zmíněné údaje.

Poté je možné nad vytvořeným objektem volat metodu *next*, která s každým zavoláním vrátí další bod s nejpodobnější normálou. Pokud zbývá už jen jeden bod, vrací pořád dokola tentýž. O prázdnosti iterátoru se lze dotázat metodou *isEmpty*.

4.4 Testování přesnosti

Při testování přesnosti bylo pomocí kontrolní implementace vygenerováno 50 různých množin náhodných bodů o mohutnosti 100 až 10^6. Celkem tedy 250 různých množin bodů, dále pak bylo vygenerováno 1000 náhodných bodů jako vstupy pro algoritmus nejbližšího souseda. Kontrolní implementace seřadí tyto body podle vzdálenosti k vstupnímu bodu. Poté je spuštěna testovaná implementace a porovnají se výsledky.

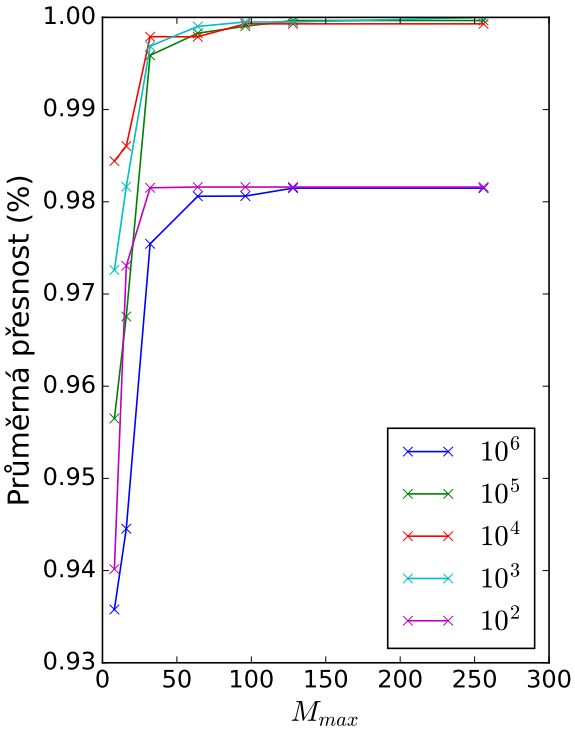
Nejbližší soused

Implementovaná struktura není 100 procentně přesná, jak je možné vidět na grafech 4.5, tak je přesná a nepřesnost nikdy nepřekročila 5 procent. Z naměřených výsledků lze vyčíst, že kriterium M_{max} má minimální vliv na přesnost. Daleko větší vliv má mohutnost vstupní množiny bodů, což jde vidět na grafu 4.5d.

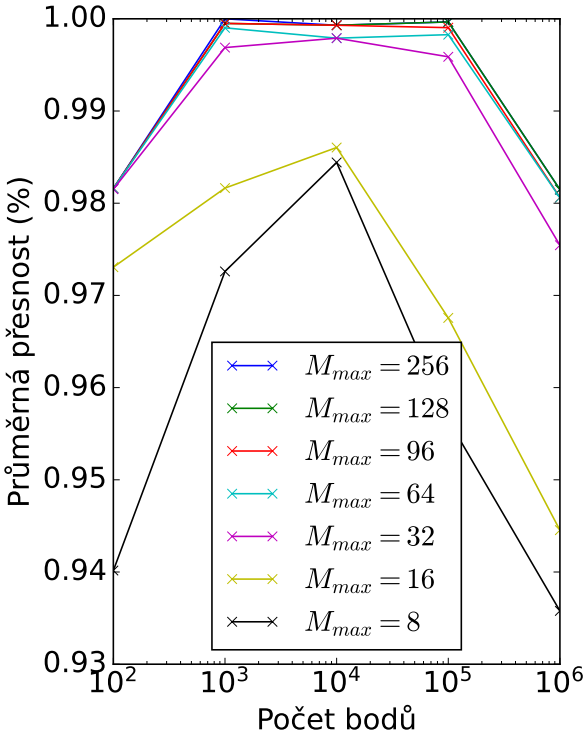
Na grafu 4.5c jde vidět, jak s rostoucím počtem bodů roste přesnost, která se u větších sad bodů blíží ke 100 procentům. Určitě bude platit, že s rostoucím počtem M_{max} bude růst přesnost, neboť je strom čím dál tím méně hluboký a výpočet se přibližuje výpočtu hrubou silou.

K-nejbližší soused

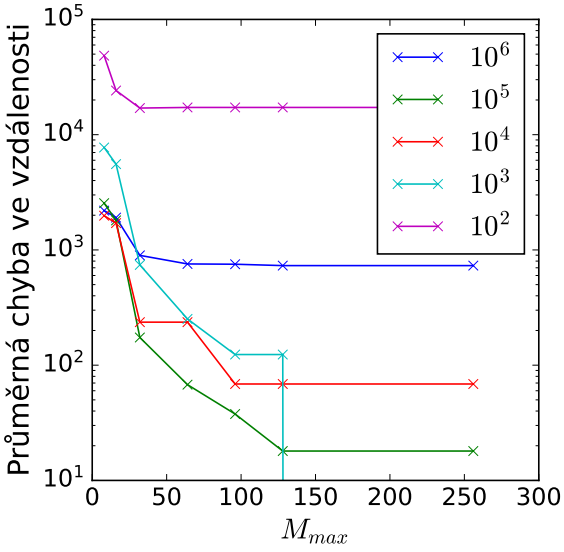
Testování probíhalo výše popsaným způsobem pro body na intervalu $-(10^6)$ až 10^6. Po algoritmu bylo požadováno, aby navracel nejbližší sousedy pro $k \in \{8, 16, 32, 64, 128, 256, 512,$



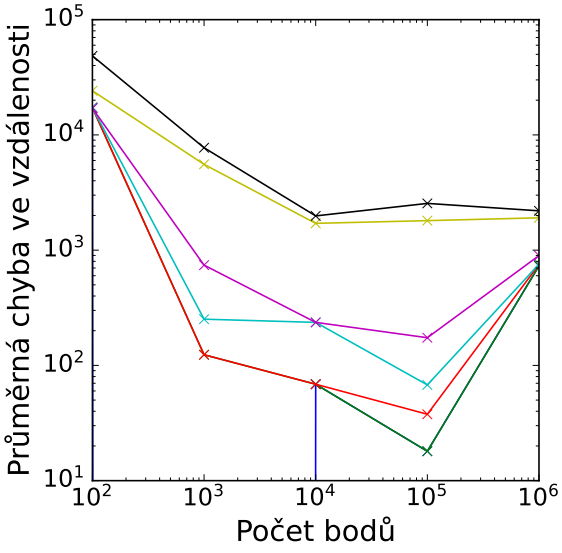
(a) Porovnání průměrné úspěšnosti nalezení nejbližšího správného souseda vzhledem ke kriteriu M_{max}.



(b) Porovnání průměrné úspěšnosti nalezení nejbližšího správného souseda vzhledem k počtu bodů.



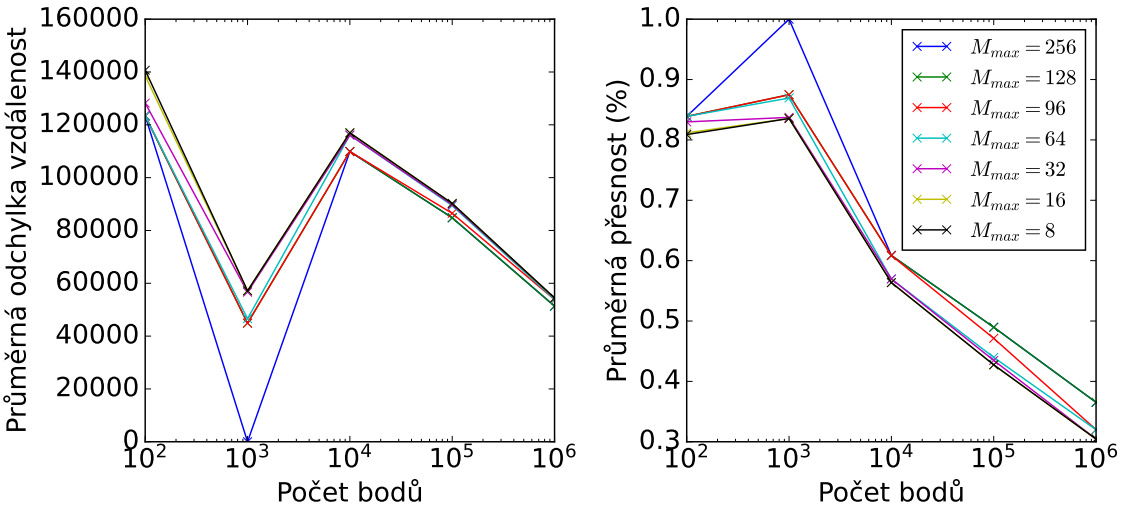
(c) Porovnání průměrné úspěšnosti nalezení nejbližšího správného souseda vzhledem ke kriteriu M_{max}.



(d) Porovnání průměrné odchylky vzdálenosti od správného nejbližšího souseda vzhledem k počtu bodů[a].

[a] Legenda pro tento graf se nachází na na 4.5b

Obrázek 4.5: Grafy ukazující přesnost pro náhodné body na intervalu $-(10^6)$ až 10^6



Obrázek 4.6: Přesnost algoritmu k-NN nad náhodnými body na intervalu $-(10^6)$ až 10^6.

1000}. Všechny tyto množiny byly vyhledávány ve stromech s $M_{max} \in \{8, 16, 32, 64, 96, 128, 256\}$. Hodnoty byly zvoleny tak, aby každý strom vybudovaný s jakýmkoliv parametrem byl nucen vyhledávat i jinde než v listovém uzlu.

Jako přesnost je uváděna hodnota, kde je podělen celkový počet možnosti k výskytu chyby (tzn. například u $k = 16$ to je 16) a uskutečněných chyb. Například pokud by v seznamu 16 prvků bylo 8 špatně, tak by přesnost byla stanovena jako 0.5.

Z grafů 4.6 je patrné, že přesnost značně klesá s počtem bodů. Nicméně průměrná chyba paradoxně klesá. Struktura tedy s přibývajícím počtem bodů nedává absolutně nejbližší prvky, ale vypočítává přibližně správné sousedy.

Nejbližší soused orientovaný zadaným směrem

Pro testování této funkcionality byly použity jiné množiny než u testování předešlých dvou funkcionalit. Na obrázcích 4.10 jsou zobrazeny množiny bodů, které budou testovány. Kompletní plocha, ze které byly body spočteny se nachází na obrázku 4.11a. Tato plocha je vlastně normální rozdělení pro 2D prostor.

Mimo souřadnice je potřeba vygenerovat i normály. Normály jsou generovány jako směrový vektor ke středu normálního rozdělení. Tyto vektory jsou normalizovány.

Kontrolní implementace

Kontrolní implementace pracuje se dvěma množinami bodů. První množinou je množina P. Tato množina je určená vstupem a vyhledávají se v ní sousedé. Druhou množinou je množina T, obsahující množiny množin bodů. Každá z těchto množina reprezentuje vstupy pro test. Aby byly testy směrodatné, bylo vygenerováno 30 takových množin. Tyto množiny obsahují body, které rovněž leží na této ploše, každá s počtem 100 bodů. Pro účely testování přesnosti se tedy uskuteční 3000 různých vyhledání nad všemi vstupními množinami. Všechny tyto body jsou na intervalu -(10^6) až 10^6.

Implementace shromažďuje všechny body se vzdáleností menší jak sto tisíc. Potom jsou tyto body seřazeny podle normály a 10 nejlepších je uloženo pro porovnání.

Přesnost

Přesností u algoritmů vracející pouze jeden bod (hrubé aproximace) je myšleno procento případů, které vrátily stejný bod jako kontrolní implementace. Pokud nevrátily stejný bod, tak implementace počítá průměrnou odchylku normály a vzdálenosti. Pokud jde o přesnost u implementace s iterátorem, rozumí se tím kolik z 10 bodů vrátil iterátor správně včetně pořadí.

Na obrazcích 4.7 jde vidět, že u hrubých aproximací klesá přesnost s rostoucím počtem bodů. Stejně, tak to je i s chybou, která roste, ale postupně se růst zpomaluje. Iterátor nevrací exaktně přesné sousedy. Vzhledem k tomu, jak aproximuje iterátor vzdálenost nedává příliš smysl měřit exaktní přesnost. Mnohem lepší představu o kvalitě výsledků lze vyčíst z průměru odchylek vzdáleností a normály. Vývoj těchto údajů si lze prohlédnout na obrázku 4.8 a 4.9. Při pohledu na obrázek 4.8, lze vidět, že chyby ve vzdálenosti s rostoucí množinou vstupních bodů sice rostou, ale růst se postupně zpomaluje.

Přesnost u podobnosti normály je velmi dobrá u všech algoritmů. U malých množin je nepřesnost větší kvůli aproximacím týkající se vzdáleností. Vývoj přesností u všech hrubých akceleracích je stejný, jak ukazuje graf 4.9. U poslední implementace se přesnost normály blíží ke sto procentům už u malých množin.

Z výsledků je patrné, že největší problém pro tyto aproximace je správné určení zkoumaných bodů pro vyhledání podobné orientace. I přesto však všechny implementace přinášejí výsledky, které jsou uvažovatelné pro nasazení u některých aplikací. Kriterium M_{max}, tentokrát významně neovlivňuje téměř nic. Z testů je taky patrné, že nastavit *effort* na hodnotu 1, minimálně na testovaných datech nemá žádný smysl z pohledu přesnosti.

4.5 Testování výkonu a paměťové náročnosti

Testování probíhalo na procesoru *AMD Ryzen 9 7950X3D* s 32 vlákny, spolu s 64 GB paměti RAM. Jako operační systém bylo použito *Ubuntu 22.04.4 LTS* s *5.15.0-102-generic* verzí kernelu. Pro překlad bylo použito *gcc 13.1.0* (Ubuntu 13.1.0-8ubuntu1~22.04). Překlad probíhal s překladovými flagy:

- `-Ofast -DNDEBUG -std=c++20 -march=native -fpic -ftree-vectorize -ffast-math -flto -o3`,

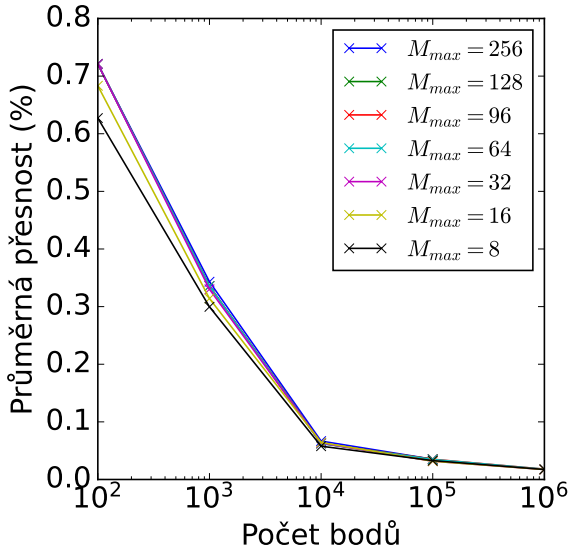
pokud není uvedeno jinak. Každý test byl spuštěn padesátkrát a všechny uváděné hodnoty jsou průměry z 50 běhů. Implementace měla k dispozici všech 32 vláken procesoru. Pro výkonnostní benchmarky byl použit framework *Google Benchmark*[6].

Pseudonáhodné body byly generovány pomocí *default_random_engine* ze standardní knihovny C++.

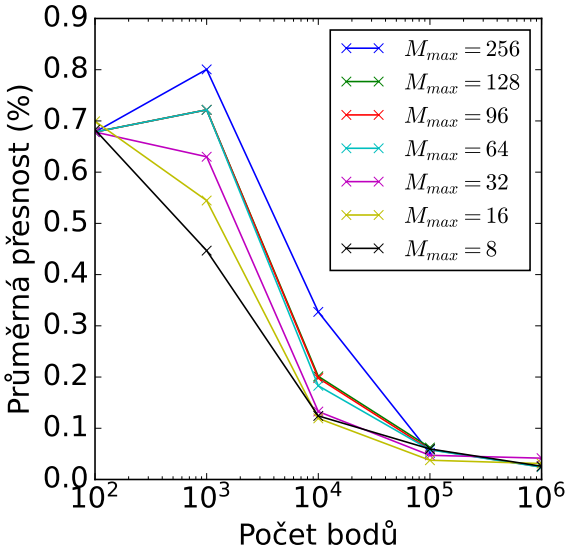
Výstavba struktury

Struktura se staví umístěním všech bodů najednou a není vhodné v průběhu přidávat další body, neboť to vyžaduje kompletní přepočítání. Předpokládá se užití struktury v režimu, kdy je předem počítána a následně používána v komplexnějším nasazení, kde se s ní bude intenzivně pracovat po dlouhou dobu.

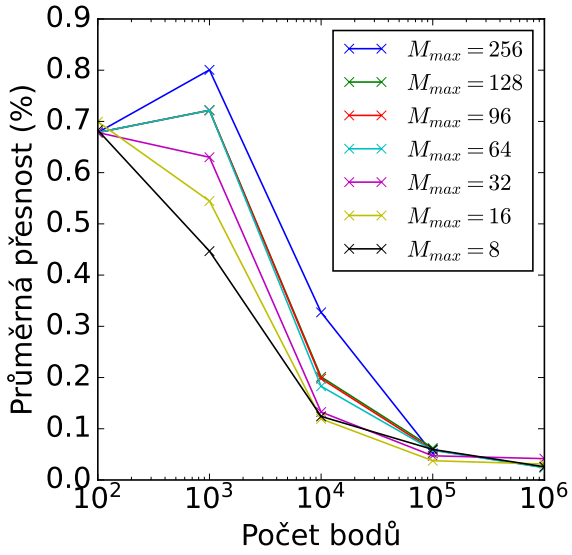
[6] https://github.com/google/benchmark



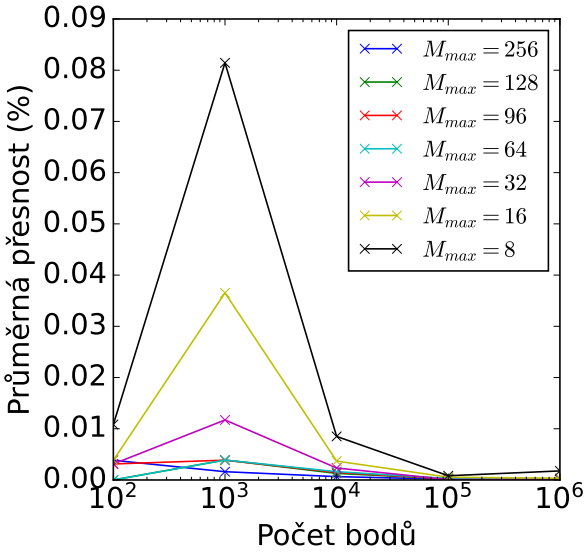
(a) Přesnost algoritmu „První splňujícími zadané kriteria“.



(b) Přesnost algoritmu „Nejbližší s nejpodobnější normálou“.

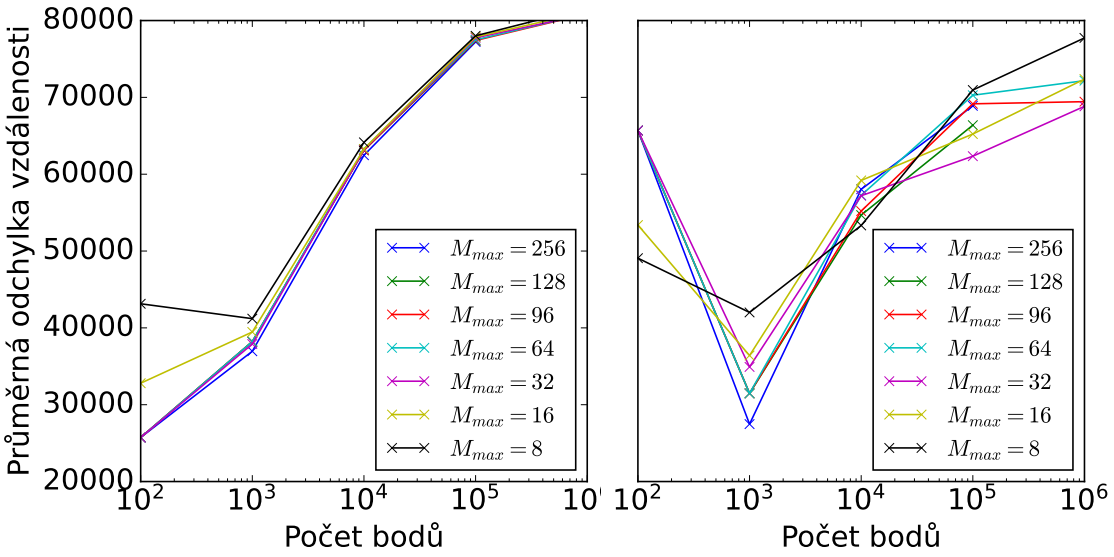


(c) Přesnost algoritmu „Nejbližší s nejpodobnější normálou“ s nastaveným parametrem *effort* na 1.



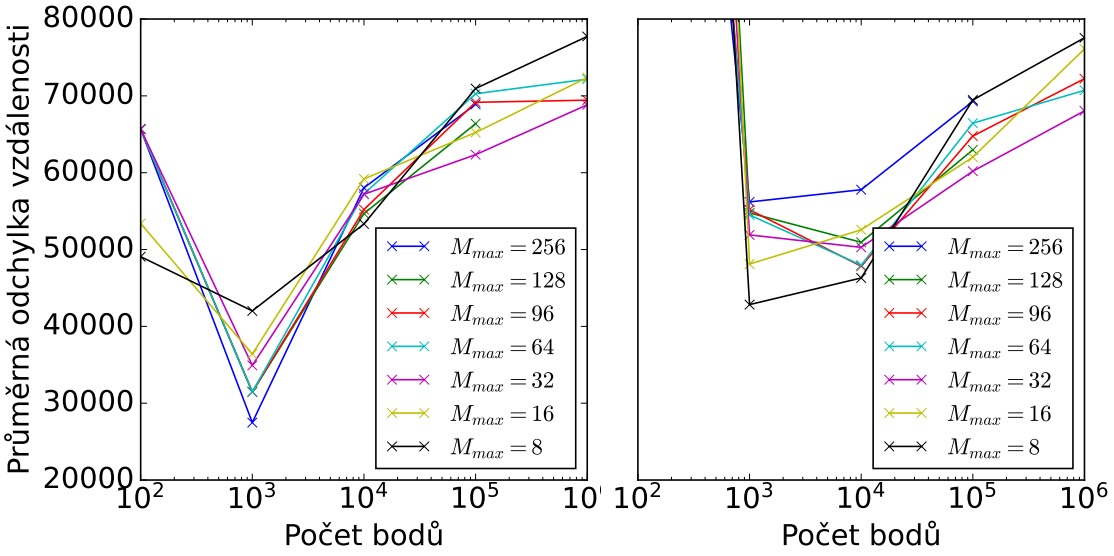
(d) Přesnost vypočtu pomocí iterátoru s aproximací vzdálenosti. Přesnost v úspěšnosti uhodnuti deseti nejpodobnějších bodů.

Obrázek 4.7: Přesnost ve výpočtech nejbližších sousedů s nejpodobnější normálou.



(a) Odchylka vzdálenosti při výpočtu pomocí „První splňujícími zadané kriteria v uzlu“.

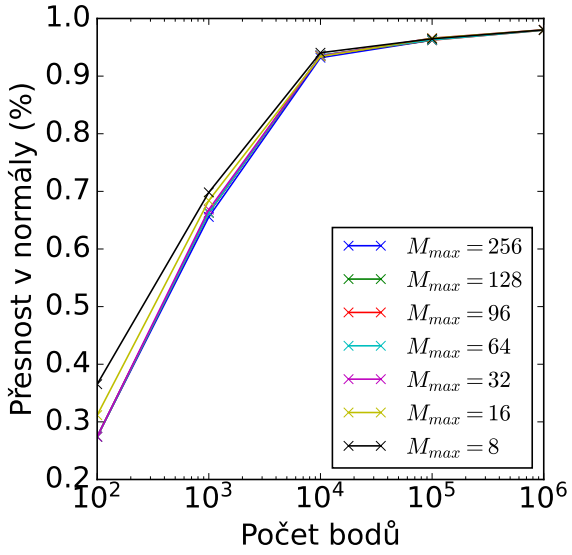
(b) Odchylka vzdálenosti při výpočtu pomocí „Nejbližší s nejpodobnější normálou“.



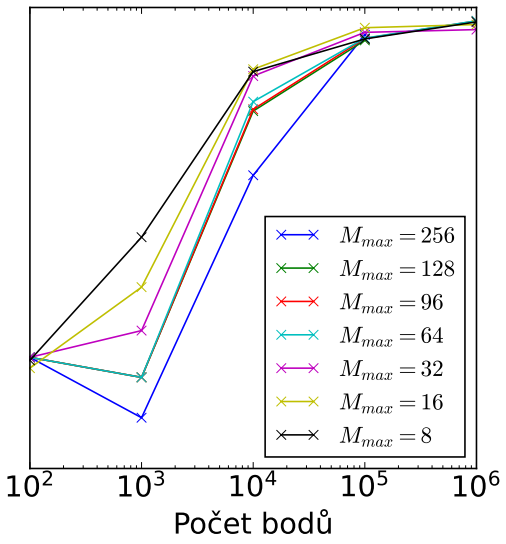
(c) Odchylka vzdálenosti při výpočtu pomocí algoritmu „Nejbližší s nejpodobnější normálou“ s nastaveným parametrem *effort* na 1.

(d) Odchylka vzdálenosti při výpočtu pomocí iterátoru s aproximací vzdálenosti. Přesnost v úspěšnosti uhodnuti deseti nejpodobnějších bodů.

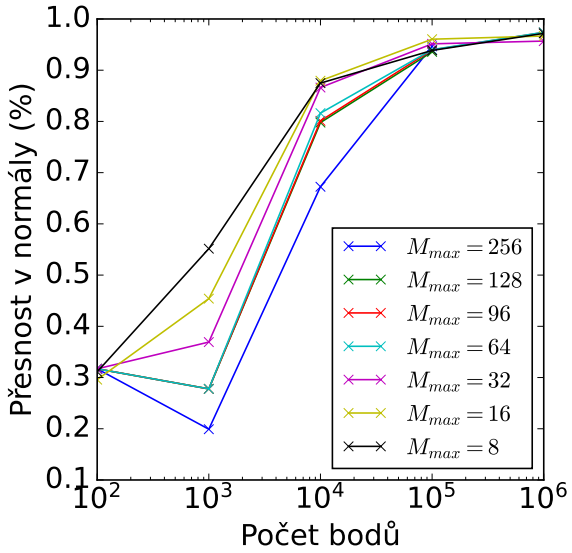
Obrázek 4.8: Chyby vzdálenosti ve výpočtech nejbližších sousedů s nejpodobnější normálou.



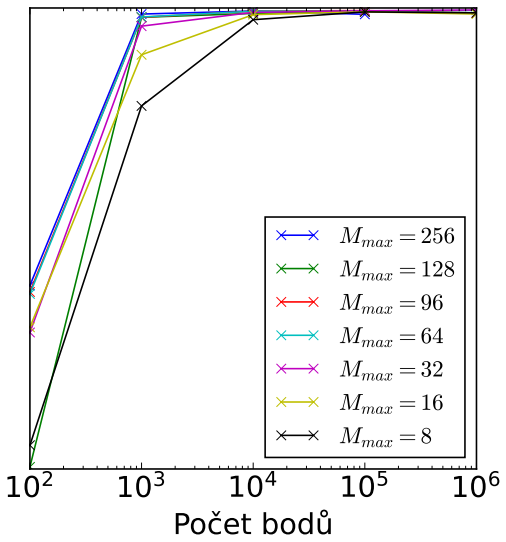
(a) Odchylka normály při výpočtu pomocí „První splňujícími zadané kriteria v uzlu“.



(b) Odchylka normály při výpočtu pomocí „Nejbližší s nejpodobnější normálou“.

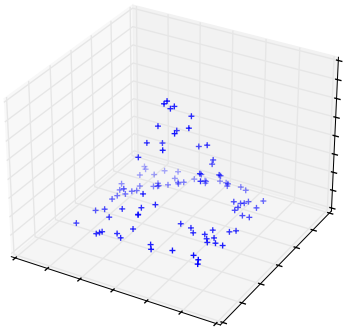


(c) Odchylka normály při výpočtu pomocí algoritmu „Nejbližší s nejpodobnější normálou“ s nastaveným parametrem *effort* na 1.

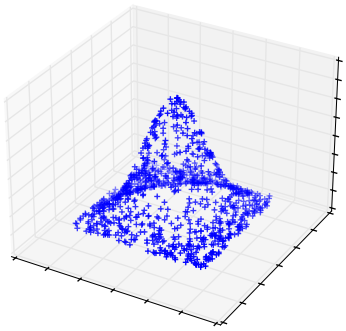


(d) Odchylka normály při výpočtu pomocí iterátoru s aproximací vzdálenosti.

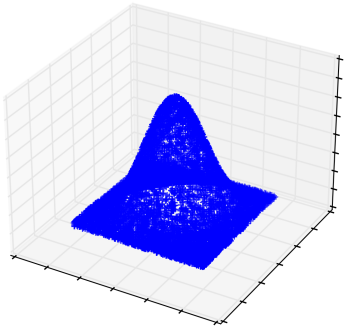
Obrázek 4.9: Chyby v normále ve výpočtech nejbližších sousedů s nejpodobnější normálou.



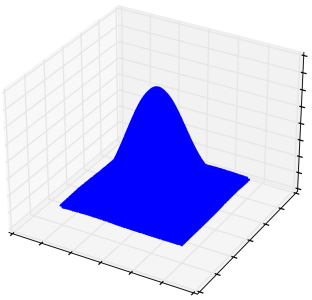
(a) 10^2 bodů



(b) 10^3 bodů

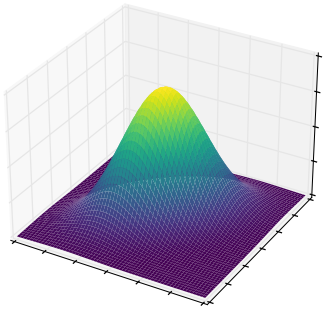


(c) 10^4 bodů

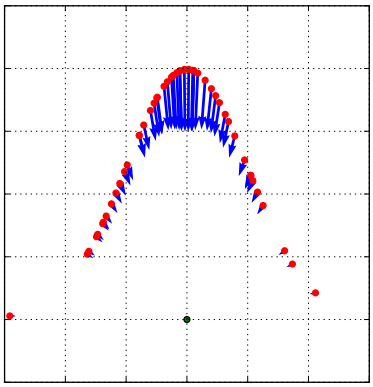


(d) 10^6 bodů

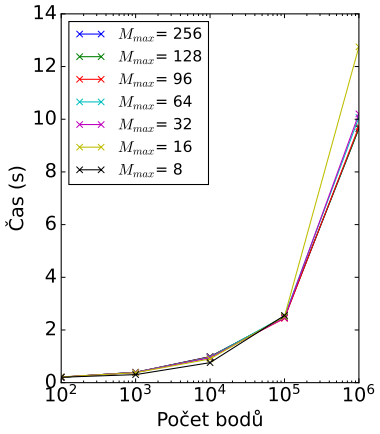
Obrázek 4.10: Grafické znázornění rozložení bodů, na kterých byl testován algoritmus nejbližšího souseda s porovnáním normály. Množství bodu na obrázcích je od 10^2 až 10^6.



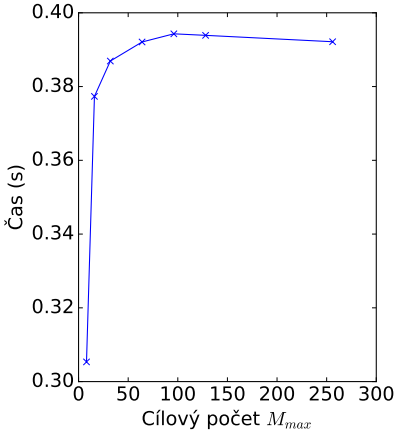
(a) Grafické znázornění plochy jenž byla použita pro generování bodů pro testování algoritmu nejbližšího souseda a s porovnáním normály.



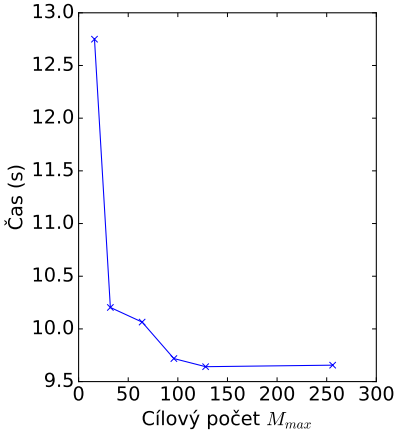
(b) Grafické ilustrace průřezu plochy a bodů procházející, spolu s jejich normálou směřující k bodu (0,0).



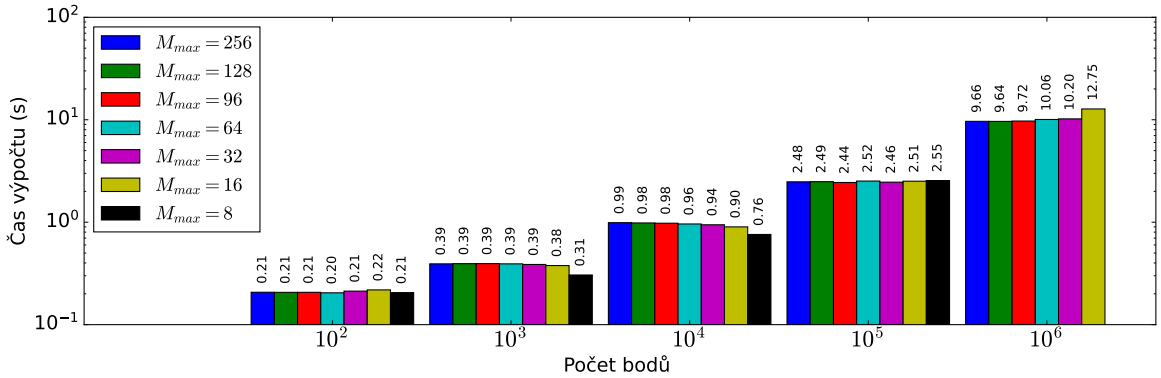
(a) Růst časové složitosti vzhledem k počtu bodů pro zvolená kriteria M_{max}.



(b) Růst časové složitosti vzhledem k počtu bodů pro vstupní počet bodů 10^3.



(c) Růst časové složitosti vzhledem k počtu bodů pro vstupní počet bodů 10^6.



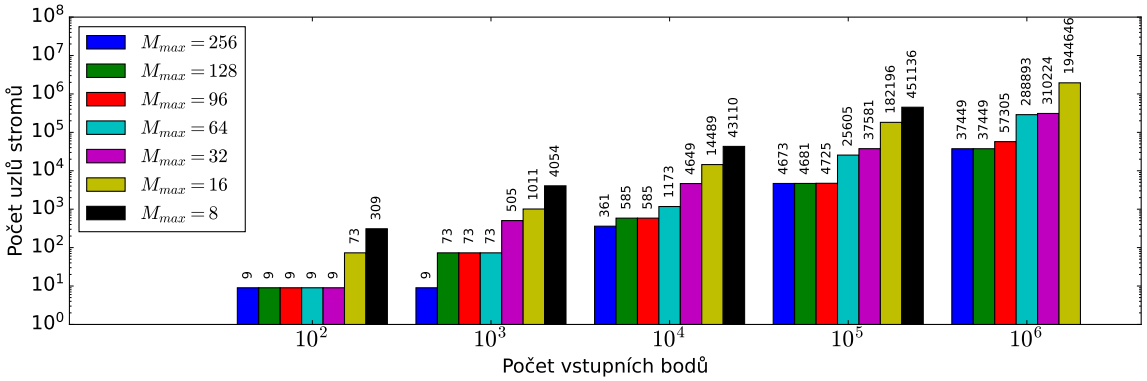
(d) Růst časové složitosti vzhledem k počtu bodů pro kriterium $M_{max} = 5$.

Obrázek 4.12: Grafy zachycující závislost mezi počtem bodů M_{max} a dobou výstavby struktury.

Vliv kriteria M_{max} na výstavbu stromu

Jak je možné vidět na grafech 4.12, M_{max} je velmi znatelný parametr, který ovlivňuje čas potřebný k výstavbě struktury. Na grafu lze vidět, jak pro malé množiny trvá budování stromu téměř stejný čas, ale s rostoucí množinou se ukazuje, kdy je vhodné zvolit jaké kriterium M_{max}. Například u velikosti 10^6 je nejrychlejší výstavba stromu pro kriterium $M_{max} = 128$. Jak je vidět na obrázcích 4.12b a 4.12c, růst má velmi rozdílnou charakteristiku pro různě velké vstupní množiny.

Růst počtu uzlů je možné vidět na grafu 4.13. Je potřeba si uvědomit, že mimo samotného stromu je potřeba mít v paměti i hašovací tabulku, jenž obsahuje všechny uzly stromu, čímž roste celkové využití paměti o to rychleji. Na grafu jde vidět, že používat nízká kriteria M_{max} se pro větší množství bodů absolutně nevyplatí. Na druhou stranu u k-nejbližšího souseda zase algoritmus benefitu-je z nižšího M_{max}, ale u nejbližšího souseda to vyhledávání spíš zpomaluje.



Obrázek 4.13: Růst počtu uzlů stromu pro náhodné body vzhledem k počtu bodů pro zvolená kriteria M_{max}.

Výkonnost algoritmu nejbližšího souseda

Při optimalizacích vyhledávání nejbližšího souseda se ukázalo, že nejužším hrdlem je implementace hašovací tabulky ze standardní knihovny[7]. Při profilování byl zjištěno, že vyhledání prvku v hašovací tabulce zabírá více, než 70 procent času vyhledávání. Vyhledání nejmenšího prvku je vektorizováno, a proto je vhodné kriterium M_{max} zvolit jako násobky osmi, aby se maximálně využilo potenciálu AVX.

Vzhledem k na měřeným datům lze konstatovat, že je vhodné zvolit kriterium M_{max} ze znalostí mohutnosti množiny vstupních bodů. Pak lze dosáhnout nejoptimálnějšího výsledku. Graf 4.14 ukazuje že, větší M_{max} je vhodný pro větší vstupní množiny. Například si je možné porovnat $M_{max} = 32$ u 10^4 a 10^5, kde se začíná projevovat toto zrychlení.

U malých vstupních množin se rychlost NN téměř nemění napříč mnoha kriterii M_{max}. Především to platí pro $M_{max} \geq 64$. Tento fakt lze vysvětlit tím, že všechny body se vejdou přímo do kořene nebo jen do první úrovně stromu.

Vyhledávání pro malé množiny je u této struktury značně neefektivní a pomalé. Zbytečnému binárnímu vyhledávání nad stromem, který má jen velmi malou hloubku a tím pádem i minimální počet uzlů, způsobuje značné zpomalení. Jedná se v podstatě o zpomalené vyhledávání hrubou silou.

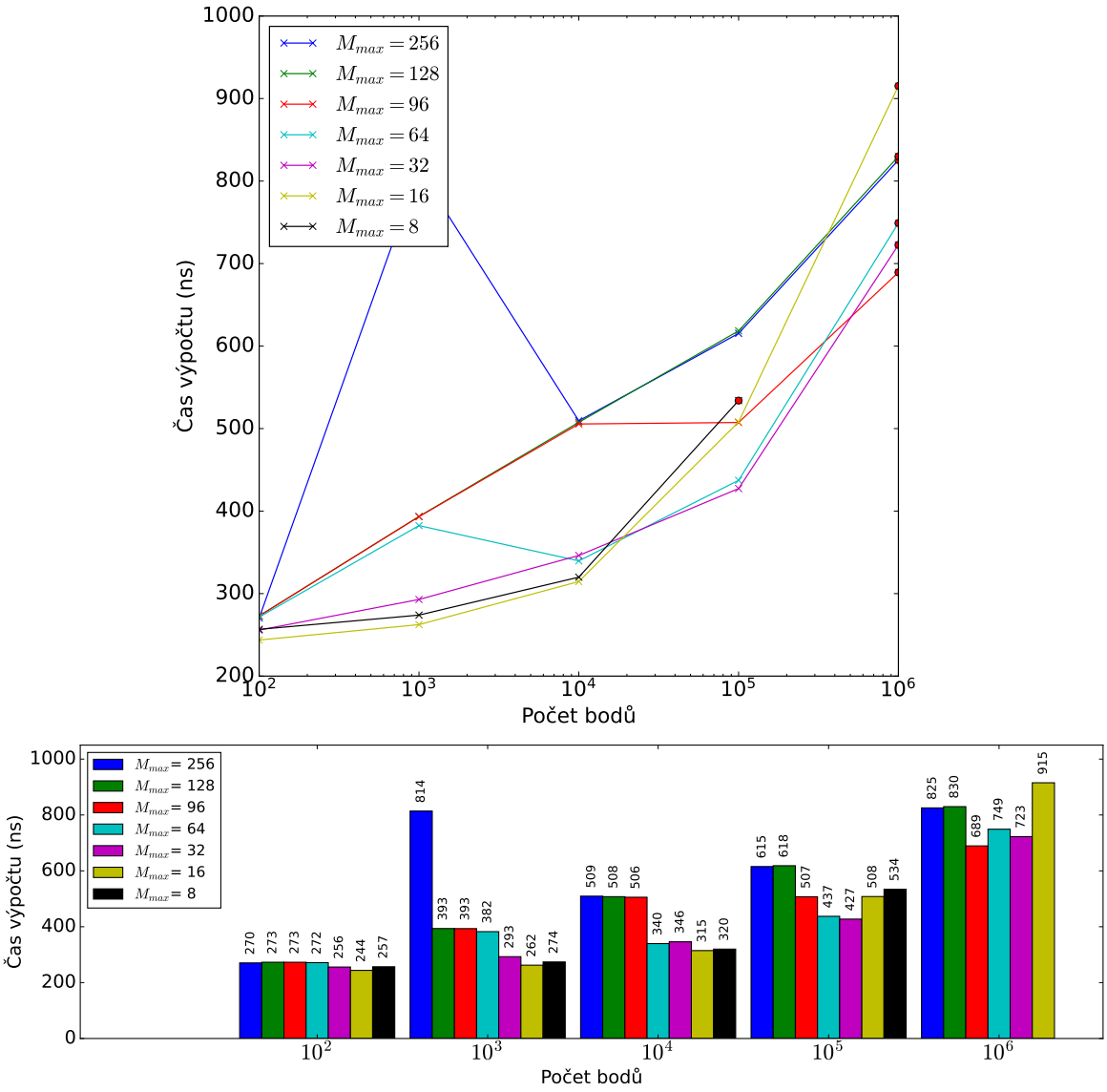
Graf 4.14 ukazuje, pro jak velké vstupní množiny je vhodné zvolit které kriterium M_{max}.

Přímé srovnání s původní implementaci není možné. Autoři nezveřejnili zdrojové kódy a není možné provést benchmarky na stejném stroji. Zdá se však, že tato varianta je asi 2x až 3x pomalejší, což je velmi pravděpodobně způsobeno nevhodně zvolenou implementací hašovací tabulky. Je potřeba dodat, že obě varianty naleznou nejbližší sousedy během několika málo stovek nanosekund.

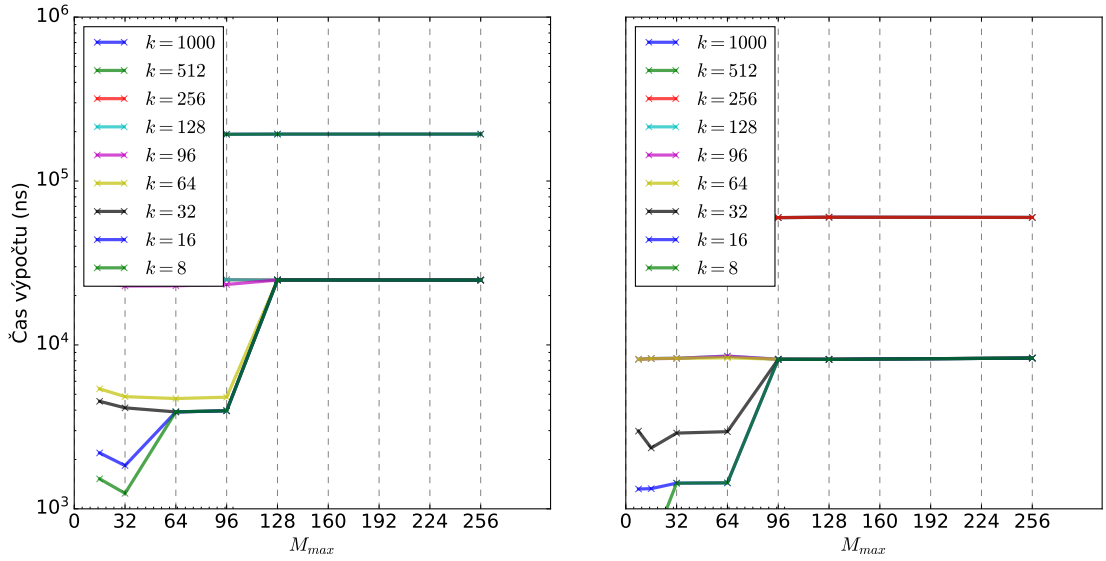
Výkonnost algoritmu k-nejbližšího souseda

Obrázek 4.15 představuje struktura, která byla použita pro vyhledávání k-nejbližších sousedů z hlediska rychlosti. Z toho vyplývá, že se vyplatí používat co možná nejmenší kriterium M_{max}. Zajímavostí může být to, že u všech testovaných k bylo rychlejší vyhledávat větší k než nastavené kriterium M_{max}. Vysvětlením tohoto úkazu je to, že vybudovaný strom obsahuje v listových uzlech jeden až M_{max} bodů. Pokud vyhledaný listový uzel obsahuje bodů

[7] https://www.youtube.com/watch?v=M2fKMP47slQ



Obrázek 4.14: Porovnání rychlosti NN pro různé nastavení M_{max} a různé počty bodů.



(a) Graf pro vstupní množinu bodů o mohutnosti 10^6

(b) Graf pro vstupní množinu bodů o mohutnosti 10^3

Obrázek 4.15: Grafy pro velikosti vstupních množin 10^3 a 10^6 k-NN algoritmu. Podobné charakteristiky mají i ostatní testované mohutnosti bodů. U $k \in \{256, 512, 1000\}$ se hodnoty překrývají.

méně, jak požadované množství, tak algoritmus vystoupá do otce, kde může v nejhorším případě nalézt až osm krát tolik uzlů[8]. Tento jev vytváří znatelné zpomalení, protože je potřeba seřadit mnohem více prvků než pouze k respektive M_{max}. Vyhledávání v nejlepším případě v 1 milionů bodů trvá přibližně 1550 nanosekund.

Výkon je stejně jako u NN limitována hašovací tabulkou. Znatelný hendikep je inicializace a vkládání hodnot do vektoru, který má být navrácen. Další drahou operací je řazení prvků podle vzdálenosti. Jak řazení tak inicializaci by se dalo vyhnout tak, že by algoritmus vracel odkaz na interní pole bodů nacházející se v listovém uzlu. V takovém případě by, ale body nebyly seřazeny a bylo by potřeba se smířit s chybou.

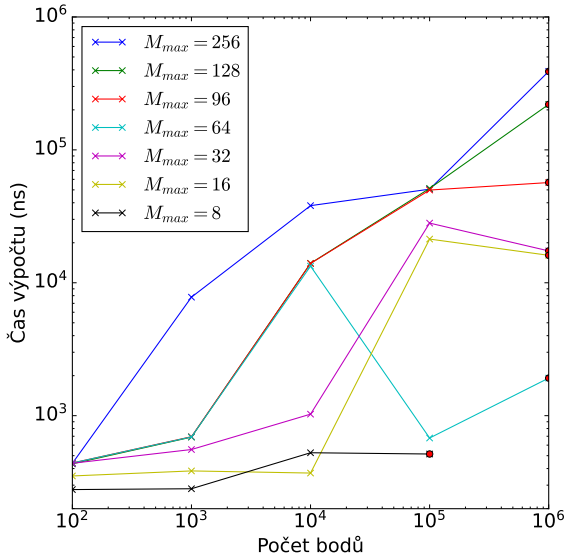
Výkonnost algoritmu nejbližšího souseda s porovnáním normály

V obrázcích 4.16, si lze povšimnout toho, že varianta pro prohledávání všech bodů náležející do listového uzlů je rychlejší, než ta která vrací první bod splňující kritéria. Ačkoliv při návrhu se počítalo s přesně opačným výsledkem. Vzhledem k tomu, že tato varianta je i přesnější je vhodné použít tuto variantu, pro získání prvního bodu. Nečekaný výsledek je způsoben lepší vektorizací. Pro M_{max} jde vidět, jak je růst složitosti s rostoucí množinou vstupních bodů velmi pozvolný. Je tedy vhodné použít malé M_{max}, ale je potřeba počítat s vysokou paměťovou náročností.

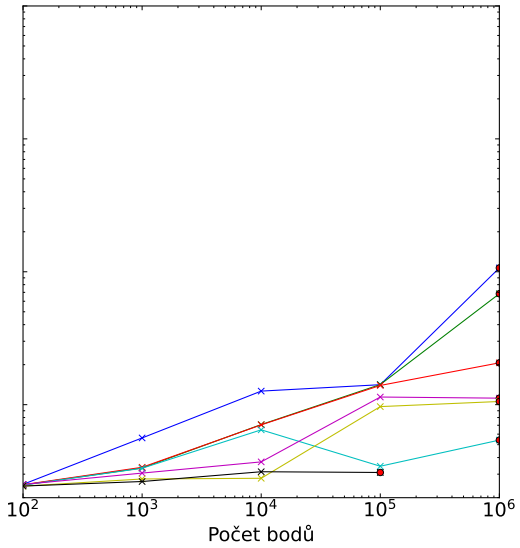
Další věc, které si lze všimnou z grafů, se týká implementace vracející první vyhovující bod. Jde vidět, jak s rostoucím počtem bodů se potenciálně vyhovující bod může nacházet až na konci pole, což způsobuje čím dál tím markantnější zpomalení.

[8] $8 \cdot M_{max} - 1$ je největší počet uzlů

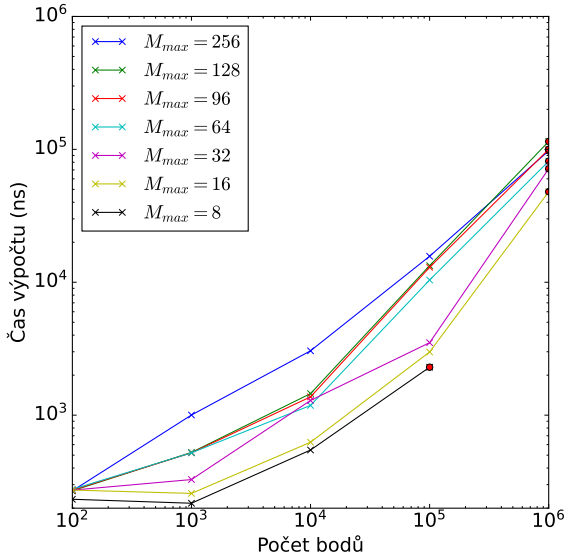
Pro iteratorovou implementací není rychlost vyhledání jediný sledovaný parametr. Je potřeba i přihlédnout k době, za kterou se vystaví tato iterační struktura, znázorněna na grafu 4.16d. Tento čas je velmi variabilní. Záleží totiž na, jak vysoko je potřeba vystoupat ve stromu a taky na M_{max}. Pro rychlost je vhodné zvolit kriterium M_{max}, kvůli přesnějším aproximacím vzdáleností a taky kvůli menší mohutnosti stromových uzlů, způsobené větší hloubkou stromu.



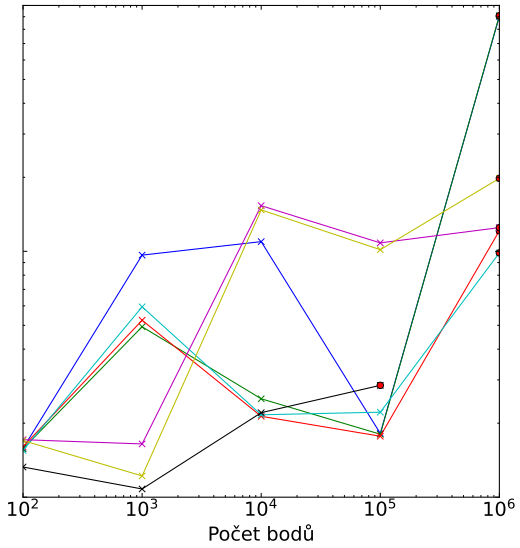
(a) Výkonost při hledání prvního vyhovujícího bodu.



(b) Výkonnost při hledání nejlepšího kandidáta uzlu.



(c) Výkonosti iterátoru pro jednu iteraci.



(d) Čas potřebný k inicializací struktury iterátoru.

Obrázek 4.16: Výkon při výpočtech nejbližších sousedů s nejpodobnější normálou.

Kapitola 5

Závěr

Cílem tento práce bylo analyzovat možnosti vyhledávání nejbližších sousedů a k-nejbližších sousedů pro orientované body. Práce má vytvořit akcelerovanou implementaci ať už pomocí vektorizace nebo paralelizace. Akcelerace je aplikována ať už na vyhledávání, budování struktury nebo na další operace. Poslední úkolem v rámci zadání této práce jsou měření ukazující výkonnost implementované implementace. Cíle této práce byli splněny.

Nastudování různých prostorové struktur pro algoritmus nejbližšího souseda a další teoretické poznatky dokládají znalosti v kapitole 2. Z těchto znalostí byla vybrána implementace pokročilé hybridní prostorové struktury, které je vysoce akcelerována pro NN. Tato práce znovu implementovala tuto struktur pouze z technického popisu a obohatila tuto implementaci o další algoritmy nad ní jako k–nejbližší soused, a vyhledávání bodů s nejpodobnější normálou. Výstavba stromu byla akcelerována, paralelizací a v úzkých hrdlech i vektorizací. Stejně tak byla vektorizována i vyhledávání. Tuto pokročilou strukturu vylepšuje především po stránce rychlosti sestavování struktury paralelizací, jak je popsaná v kapitole 3. Algoritmy v této struktuře byly důkladně otestovány v sekci 4.4 a 4.5 jak už po výkonnostní, tak po exaktnostní stránce a bylo vyhodnoceno, jaké nastavení struktury je vhodné pro jaké použití. Je zřejmé, že všechny body zadání se podařilo splnit.

Implementovanou strukturu a původní strukturu není možné porovnat z důvodu nedostupné implementace a nebo procesoru, který byl původními autory použit. Z naměřených dat se však zdá, že prezentovaná varianta je pomalejší především pak z důvodu použití nevhodné implementace hašovací tabulky. U algoritmu k–NN, algoritmus nevrací exaktně správný seznam, ale přijatelnou aproximaci, u které navíc s rostoucím počtem bodů roste i přesnost. U nově vymyšlených implementací algoritmů s porovnáním normály není přesnost exaktní, ale obvykle dostačují.

Tato práce mi dala hluboké povědomí o prostorových strukturách využitelných v různých aplikacích napříč různými obory informatiky. Na práci jsem si vyzkoušel, praktické řešení různých problémů spojených s implementací výkonově orientovaných algoritmů, paralelizací a vektorizací. V neposlední řadě jsem si v práci měl možnost zkusit i různé nové funkcionality moderního C++.

Vhodným rozšířením této práce souvisí s limitacemi nalezenými při měření výkonu. Nejvíc limitující se zdá pomalá implementace hašovací tabulky, jejíž optimalizace by mohla vést až k zrychlení o 60 procentu u NN a obdobnému u ostatních algoritmů. Vhodným rozšířením by mohla být i jiná implementace výpočtu Voronoi buněk šitá na míru této aplikaci, především pak pokud by umožňovala počítat přesné sousedy každé Voronoi buňky a tím by došlo k zrychlení budování struktury. Další návrhem by mohl být i paralelní vyhledávání u větší kriterií M_{max} pro orientované body.

Literatura

[1] Barki, H., Denis, F. a Dupont, F. A New Algorithm for the Computation of the Minkowski Difference of Convex Polyhedra. In: *2010 Shape Modeling International Conference*. 2010, s. 206–210. DOI: 10.1109/SMI.2010.12.

[2] Bhatia, N. et al. Survey of nearest neighbor techniques. *ArXiv preprint arXiv:1007.0085*. 2010.

[3] Dhanabal, S. a Chandramathi, S. A review of various k-nearest neighbor query processing techniques. *International Journal of Computer Applications*. Citeseer. 2011, sv. 31, č. 7, s. 14–22.

[4] Dolatshah, M., Hadian, A. a Minaei Bidgoli, B. Ball*-tree: Efficient spatial indexing for constrained nearest-neighbor search in metric spaces. *ArXiv*. 2015, abs/1511.00628. Dostupné z: https://api.semanticscholar.org/CorpusID:14162909.

[5] Drost, B. a Ilic, S. Almost constant-time 3D nearest-neighbor lookup using implicit octrees. *Machine Vision and Applications*. Únor 2018, sv. 29, s. 1–13. DOI: 10.1007/s00138-017-0889-4.

[6] Fairouz, A. A., Abusultan, M., Fedorov, V. V. a Khatri, S. P. Hardware Acceleration of Hash Operations in Modern Microprocessors. *IEEE Transactions on Computers*. 2021, sv. 70, č. 9, s. 1412–1426. DOI: 10.1109/TC.2020.3010855.

[7] Friedman, J. H., Bentley, J. L. a Finkel, R. A. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)*. ACM New York, NY, USA. 1977, sv. 3, č. 3, s. 209–226.

[8] Gilbert, E., Johnson, D. a Keerthi, S. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal on Robotics and Automation*. 1988, sv. 4, č. 2, s. 193–203. DOI: 10.1109/56.2083.

[9] Intel Corporation. *Compiler Auto-Vectorization Guide*. Intel Corporation, 2023. PDF document. Dostupné z: https://www.intel.com/content/dam/develop/external/us/en/documents/31848-compilerautovectorizationguide.pdf.

[10] Jiménez, P., Thomas, F. a Torras, C. 3D collision detection: a survey. *Computers & Graphics*. 2001, sv. 25, č. 2, s. 269–285. DOI: https://doi.org/10.1016/S0097-8493(00)00130-8. ISSN 0097-8493. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0097849300001308.

[11] Knuth, D. E. *The art of computer programming / Vol. 3, Sorting and serching*. 3rd ed. Upper Saddle River: Addison-Wesley, 1998. ISBN 0-201-89685-0.

[12] Limited, A. *Helium*. Arm Developer, 2024. Accessed on 2024-04-10. Dostupné z: https://developer.arm.com/Architectures/Helium.

[13] Limited, A. *Neon*. Arm Developer, 2024. Accessed on 2024-04-10. Dostupné z: https://developer.arm.com/Architectures/Neon.

[14] Limited, A. *SVE*. Arm Developer, 2024. Accessed on 2024-04-10. Dostupné z: https://developer.arm.com/Architectures/Scalable%20Vector%20Extensions.

[15] Liu, T., Moore, A. W. a Gray, A. New Algorithms for Efficient High-Dimensional Nonparametric Classification. *Journal of Machine Learning Research*. 2006, sv. 7, č. 41, s. 1135–1158. Dostupné z: http://jmlr.org/papers/v7/liu06a.html.

[16] Lu, J., Lazar, E. A. a Rycroft, C. H. An extension to Voro++ for multithreaded computation of Voronoi cells. *Computer Physics Communications*. 2023, sv. 291, s. 108832. DOI: https://doi.org/10.1016/j.cpc.2023.108832. ISSN 0010-4655. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0010465523001777.

[17] Moore, A. W. *An Introductory Tutorial on Kd-Trees*. Computer Laboratory, University of Cambridge, 1991. PhD Thesis. Carnegie Mellon University. Extract from Andrew Moore's PhD Thesis: Efficient Memory-based Learning for Robot Control.

[18] Omohundro, S. M. *Five balltree construction algorithms*. 1989.

[19] Peterson, L. E. K-nearest neighbor. *Scholarpedia*. 2009, sv. 4, č. 2, s. 1883. DOI: 10.4249/scholarpedia.1883. revision #137311.

[20] Rycroft, C. H. VORO++: A three-dimensional Voronoi cell library in C++. *Chaos: An Interdisciplinary Journal of Nonlinear Science*. Říjen 2009, sv. 19, č. 4, s. 041111. DOI: 10.1063/1.3215722. ISSN 1054-1500. Dostupné z: https://doi.org/10.1063/1.3215722.

[21] Tapia Fernández, S., García García, D. a García Hernandez, P. Key Concepts, Weakness and Benchmark on Hash Table Data Structures. *Algorithms*. 2022, sv. 15, č. 3. DOI: 10.3390/a15030100. ISSN 1999-4893. Dostupné z: https://www.mdpi.com/1999-4893/15/3/100.

[22] Team, G. *GCC Tree SSA Vectorization*. 2024. Dostupné z: https://gcc.gnu.org/projects/tree-ssa/vectorization.html.

[23] Team, G. *Using Vector Instructions through Built-in Functions*. 2024. Dostupné z: https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html.

Příloha A

Obsah přiloženého paměťového média

Na přiloženém paměťovém mediu se nachází kompletní projekt připravený k překladu. Na mediu se také nachází externí závislosti ve verzích, které byly použity (Google Benchmark, Google Test, Voro++). Jedinou externí závislostí, kterou je potřeba mít nainstalovanou, je OPENMP ve verzi alespoň 4.5.

Na přiloženém mediu se nachází:

složka src Obsahuje všechny soubory nezbytné pro funkci knihovny, včetně hlavičkových a zdrojových souborů.

složka benchmark Obsahuje benchmarky, které byly použity pro testování výkonu.

složka accuracy_test Obsahuje zdrojové soubory pro testování přesnosti a generování souborů, které obsahují nejbližší sousedy získané hrubou silou. Testy v této složce byly napsány co možná nejvíc přímočaře, aby se vyloučily chyby.

složka test Obsahuje testy některých komponent, u kterých byly během implementace pochybnosti o funkčnosti.

složka picture Obsahuje skripty, které byly použity pro generování většiny obrázků pro text práce.

složka deps Obsahuje externí závislosti.

CMakeLists.txt Hlavní CMake soubor celého projektu, umístěný přímo v kořenovém adresáři.

Dockerfile Dockerfile pro jednoduché spuštění projektu na různých systémech, bez ohledu na nainstalované knihovny.

readme.md Návod na kompilaci a seznam závislostí.

složka docs zdrojové soubory latexu pro vygenerování tohoto souboru.