VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Vyhledávání K nejbližších sousedů na základě filtru

Search K nearest neighbors based on a filter

Bc. Jan Jedlička

Vedoucí práce: Doc. Ing. Radim Bača, Ph.D.

Ostrava, 2022

Abstrakt

Techniky pro efektivní vyhledání K nejbližších sousedů (tzv. KNN problém) jsou základem pro mnoho dnešních aplikací. Velmi často se využívají i techniky pro přibližné KNN vyhledávání. Tyto techniky jsou založeny na grafech. Předmětem této práce rozšíření existující implementace pro přibližné KNN vyhledávání o možnost specifikovat filtr. Filtr bude podmínka, která stanoví, které vektory se při prohledávání vynechají.

Klíčová slova

KNN;HNSW;Filter

Abstract

Techniques for effective nearest neighbor search (so-called KNN problem) are the basis for many of today's applications. Techniques for approximate KNN searches are also very often used. These techniques are based on graphs. The subject of this work is to extend the existing implementation for approximate KNN searches with the ability to specify a filter. The filter will be a condition that determines which vectors are omitted when searching.

Keywords

KNN;HNSW;Filter

Poděkování

Rád bych na tomto místě poděkoval vedoucímu semestrálního projektu, kterým byl pan Doc. Ing. Radim Bača, Ph.D., za pravidelné konzultace a poskytnutí mnoha užitečných rad a nápadů pro řešení samotné práce.

Obsah

Seznam použitých zkratek a symbolů

KNN	–	K-nearest neighbors
HNSW	–	Hierarchical Navigable Small Worlds

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Cílem této semestrální práce bylo pochopit a následně naimplementovat HNSW algoritmus vyhledávající K nejbližších sousedů v prostoru n-diemnzionálních vektorů. Následně bylo zapotřebí tuto mou implementaci rozšířit o vyhledávání prvků na základě zadaných filtrů. Filtry jsou booleovsé podmínky omezující hodnoty jednotlivých atributů u prvků pro K vyhledaných sousedů. Veškerá práce byla napsána v jazyku C++. Tento jazyk je volen vzhledem k jeho nízké úrovni, což umožňuje maximální rychlosti, přímou práci s pamětí, a tato vlastnost je zvlášť u databázových operací a systémů podstatná.

Kapitola 2

KNN

KNN algoritmy slouží pro získání K nejbližších prvků od zvoleného prvku (QueryNode/TargetNode) v n-dimenzionální prostoru. Tyto techniky jsou základem pro mnoho dnešních aplikací a často se i využívají metody pro přibližné KNN vyhledávání. Tato hodnota K bývá relativně malá, nejčastěji získáváme 10 nejbližších prvků, případně hodnoty pod 100, s tím že K může nabývat i mnohonásobně vyšších hodnot což ale není příliš časté.

Pro určení vzdálenosti mezi jednotlivými prvky v n-dimenzionálním prostoru můžeme využít celou řadu metrik. Mezi ty nejznámější patří Euklidova, Hammingova, Minkovského případně Čebyševova. Ve vypracované implementaci se vždy pracuje s Euklidovou vzdáleností, nicméně by nebyl problém zaměnit definici metody pro určení vzdálenosti mezi prvky a tím pádem změnit použitou metriku.

Dimenze prostoru ve kterém vyhledáváme prvky nemusí být vůbec nízká. Dimenze prostoru se může pohybovat v rozmezí od jednotek (2,5,...) do stovek (sift vecdim = 128). A hodnoty jednotlivých atributů se mohou rovněž tak pohybovat v širokém spektru, s tím že v mé práci jsou všechny atributy datového typu float.

Kapitola 3

HNSW

Hierarchical Navigable Small Worlds [1], dále jen zkráceně HNSW, je jeden ze způsobů jakým řešit KNN problém. Tato metoda je založena na přibližném vyhledávání (ANN) za použití grafů. To znamená že výsledek je poskytován pouze s určitou přesností která je definována pomocí Recall, což je poměr relevantních výsledků které jsme získali.

Přesnost s jakou chceme získat výsledných K sousedů lze měnit. Metoda pro vyhledání KNN má parametr Ef udávající počet prvků které vyhledáváme, z nichž nakonec vracíme K nejbližších. Čím vyšší bude hodnota Ef tím s větší přesností proběhne vyhledání, zároveň tak poroste i čas vyhledávání a proto je potřeba najít vhodné Ef pro naše potřeby (3.2) (3.6).

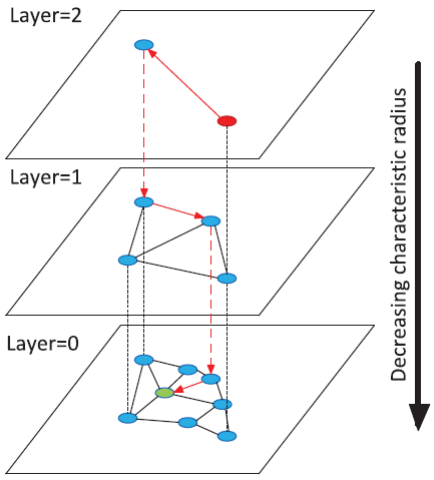
Důvod přibližného vyhledávání a ne procházení všech prvků je prostý, chceme co nejvyšší propustnost. Určování vzdálenosti mezi prvky nám zabírá mnoho času, okolo 90%, a proto chceme tuto operaci provádět co nejméně. V HNSW se vzhledem k přibližnému vyhledávání zhruba 90% prvků vůbec neprochází což ušetří velkou část času vykonání operace a proto tento způsob vyhledávání KNN dosahuje dobrých výsledků.

Jak již bylo zmíněno HNSW je založen na uložení dat pomocí grafů, konkrétně několikavrstvých grafů. Každý node v prostoru má Mmax sousedů, většinou 16/32, s tím že prvky mohou být a většinou i bývají sousedi navzájem. Vrstvy (layers) (3.1) grafů fungují tak že v nejvyšší vrstvě, v mé implementaci se jedná o vrstvu 3, se nachází poze zlomek prvků (okolo 500 z 1 000 000) a slouží poze k rychlému průchodu a nálezu nového entryPoint, tedy prvku z kterého se posuneme do nižší vrstvy a kde nakonec začně vyhledávání. Takto se prochází i nižší vrstvy, v mém případě vrstva 2 a hledá se v nich pouze 1 nový entryPoint. Když narazíme na nízkou vrstvu, vrstvu 1 která již obsahuje zhruba 10% všech prvků, a vstoupíme do ní pomocí dříve nalezeného entryPoint a vyhledáváme již EfC prvků. EfC je většinou 200 a EfC prvků se vyhledává ve vrstvě 1 pokud tuto vrstu procházíme z důvodu vložení nového prvku do grafu, pokud bychom do této vrstvy vstupovali z důvodu vyhledání K sousedů tak nám i vrstva vrací pouze 1 entryNode jako vyšší vrstvy. A nakonec vstupujeme do nejnižší vrstvy, vrstvy 0, buď s W o velikosti EfC v případě vkládání nového prvku nebo W o velikosti 1 v případě vyhledání KNN.

3.1 Implementace HNSW

Pro lepší pochopení HNSW se již naimplementované projekty [2] [3] využívaly pouze jako reference pro porovnání podobnosti výsledků a srovnání časů vykonání operací. Implemence HNSW se tedy vytvořila od základu nová dle referenčního projektu a pseudokódu v článku popisujícím principy HNSW. Vlastní implementace celého projektu přinesla výhodu v maximálním porozumění kódu a algoritmům použitým v HNSW. Na druhou stranu je tato implementace přibližně 2,5x pomalejší než referenční kód, výsledky ale vrací rovněž validní jako reference. Čas vkládání všech 1 000 000 prvků do grafu trvá přibližně 30 minut, u reference trvalo vytváření 15 minut lze tedy vidět dvojnásobné zpomalení.

Metoda Search (3.3) pro vyhledání K nejbližších prvků v grafu od queryNodu s tím že máme zadaný 1 nebo více entryPoint prvků (W) se využívá s drobnými úpravami jak pro operaci vložení prvku do grafu (3.2) tak i pro vyhledávání KNN prvků (3.5). Pokud chceme prvek vložit tak musíme najít pozici kam do grafu ho dát a s kým bude sousedit což nám Search vrátí. Pokud vyhledáváme KNN prvků tak nám Search vrací K nejbližších prvků. Tato metoda funguje na jednoduchém prinicpu. Na začátku máme W již nalezených prvků které následně prohlásíme za ty nejbližší (entryPoint/s), V již navštívených prvku, a C potencionálních kandidátů na nejbližší prvky tedy prvky v W. Před začátkem průchodu jsou v C i V uloženy všechny prvky co jsou v W. Následně začíná průchod, který se opakuje dokud je $C > 0$. Získáme si prvek c, nejbližší prvek z C (vzdálenosti prvků, nejbližší/nejvzdálenější jsou vždy vohodnoceny jako vzdálenost daného prvku ku zadaném queryNodu) a f, nejvzdálenšjší prvek z W. Pokud je vzdálenost c vetší než vzdálenost f tak průchod končí a našli jsme W s námi hledanými prvky. V opačném případě jdeme dál. Procházíme všechny sousedy e od prvku c. Pokud e není mezi navšívenými prvky V tak jeji přidáme do V, do f opět uložíme nejvzdálenější prvek z W a pokud je vzdálenost mezi e a q menší než vzdálenost mezi f a q a nebo je velikost W menší než zadané Ef tak pokračujeme dál. Do C i W vložíme prvek e a pokud je velikost W větší než Ef tak z W odstraníme nejvzdálenější prvek. Takto procházíme všechny prvky c z C a následně všechny sousedy e z c dokud není $C == 0$ nebo není vzdálenost mezi nejbližšího prvku z C (c) větší jak vzdálenost nejvzdálenějšího prvku z W (f).



Obrázek 3.1: Vrstvy grafu

Algorithm 1. INSERT($hnsw, q, \mathrm{M}, M_{max}, efConstruction, m_L$)

Input: multilayer graph *hnsw*, new element q, number of established connections M, maximum number of connections for each element per layer M_{max}, size of the dynamic candidate list *efConstruction*, normalization factor for level generation m_L
Output: update *hnsw* inserting element q

1 $W \leftarrow \emptyset$ // list for the currently found nearest elements
2 $ep \leftarrow$ get enter-point for *hnsw*
3 $L \leftarrow$ level of ep // top layer for *hnsw*
4 $l \leftarrow \lfloor -\ln(unif(0..1)) \cdot m_L \rfloor$ // new element's level
5 **for** $l_c \leftarrow L \ldots l+1$
6 $W \leftarrow \mathrm{SEARCH-LAYER}(q, ep, ef = 1, l_c)$
7 $ep \leftarrow$ get the nearest element from W to q
8 **for** $l_c \leftarrow \min(L, l) \ldots 0$
9 $W \leftarrow$ SEARCH-LAYER(q, ep, *efConstruction*, l_c)
10 *neighbors* $\leftarrow$ SELECT-NEIGHBORS(q, W, M, l_c) // Algorithm 3 or Algorithm 4
11 add bidirectionall connectionts from *neighbors* to q at layer l_c
12 **for** each $e \in$ *neighbors* // shrink connections if needed
13 *eConn* $\leftarrow$ *neighbourhood*(e) at layer l_c
14 **if** *eConn* $> M_{max}$ // shrink connections of e // if $l_c = 0$ then $M_{max} = M_{max0}$
15 *eNewConn* $\leftarrow$ SELECT-NEIGHBORS(e, *eConn*, M_{max}, l_c) // Algorithm 3 or Algorithm 4
16 set *neighbourhood*(e) at layer l_c to *eNewConn*
17 $ep \leftarrow W$
18 **if** $l > L$
19 set enter-point for *hnsw* to q

Obrázek 3.2: Pseudokód metody Insert Node

Algorithm 2. SEARCH-LAYER(q, ep, ef, l_c)

Input: query element q, enter-points ep, number of nearest to q elements to return ef, layer number l_c
Output: ef closest neighbors to q

```
1  v ← ep // set of visited elements
2  C ← ep // set of candidates
3  W ← ep // dynamic list of found nearest neighbors
4  while |C| > 0
5    c ← extract nearest element from C to q
6    f ← get furthest element from W to q
7    if distance(c, q) > distance(f, q)
8      break // all elements in W are evaluated
9    for each e ∈ neighbourhood(c) at layer l_c // update C and W
10     if e ∉ v
11       v ← v ∪ e
12       f ← get furthest element from W to q
13       if distance(e, q) < distance(f, q) or W < ef
14         C ← C ∪ e
15         W ← W ∪ e
16         if |W| > ef
17           remove furthest element from W to q
18 return W
```

Obrázek 3.3: Pseudokód metody Search Layer

Algorithm 4. SELECT-NEIGHBORS-HEURISTIC(q, C, M, l_c, $extendCandidates$, $keepPrunedConnections$)

Input: base element q, candidate elements C, number of neighbors to return M, layer number l_c, flag indicating whether or not to extend candidate list $extendCandidates$, flag indicating whether or not to add discarded elements $keepPrunedConnections$
Output: M elements selected by the heuristic

```
1  R ← ∅
2  W ← C // working queue for the candidates
3  if extendCandidates // extend candidates by their neighbors
4    for each e ∈ C
5      for each e_adj ∈ neighbourhood(e) at layer l_c
6        if e_adj ∉ W
7          W ← W ∪ e_adj
8  W_d ← ∅ // queue for the discarded candidates
9  while |W| > 0 and |R| < M
10   e ← extract nearest element from W to q
11   if e is closer to q compared to any element from R
12     R ← R ⋃ e
13   else
14     W_d ← W_d ∪ e
15 if keepPrunedConnections // add some of the discarded// connections from W_d
16   while W_d > 0 and R < M
17     R ← R ∪ extract nearest element from W_d to q
18 return R
```

Obrázek 3.4: Pseudokód metody Select Nearest Neighbors

Algorithm 5. K-NN-SEARCH(*hnsw*, *q*, *K*, *ef*)

Input: multilayer graph *hnsw*, query element *q*, number of nearest neighbors to return *K*, size of the dynamic candidate list *ef*
Output: *K* nearest elements to *q*

```
1 W ← ∅ // set for the current nearest elements
2 ep ← get enter-point for hnsw
3 L ← level of ep // top layer for hnsw
4 for l_c ← L ... 1
5     W ← SEARCH-LAYER(q, ep, ef = 1, l_c)
6     ep ← get nearest element from W to q
7 W ← SEARCH-LAYER(q, ep, ef, l_c = 0)
8 return K nearest elements from W to q
```

Obrázek 3.5: Pseudokód metody KNN

3.2 Měření závislosti času operace KNN na přesnosti

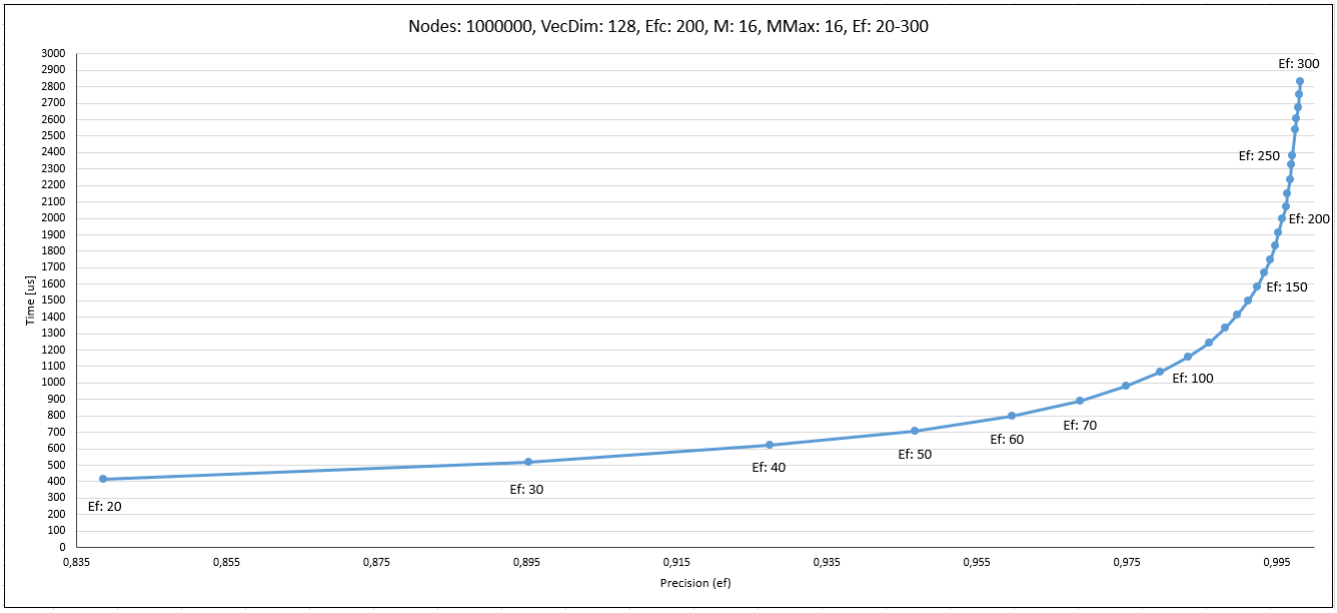
Testování a měření probíhalo nad datovou kolekcí Sift1M. Tato kolekce obsahuje jak data prvků v prostoru tak i dotazy a očekávané množiny výsledků nad těmito dotazy. V hlavním souboru sift1M se nachází binární data obsahující 1 000 000 rozdílných bodů v prostoru o dimenzi 128 a floatové hodnotě jendotlivých atributů v rozmezí od 0 do 217, přičemž v atributech jsou pouze celočíselné hodnoty.

Při vkládání prvků do hnsw bylo EfC nastaveno na 200, a M i MMax na hodnotu 16.

Z tabulky (3.2) i grafu (3.6) lze vidět že při nízkém Ef = 20 je čas operace také velice nízký, a to 0,413ms, přesnost je ale bohužel rovněž až moc nízká 0,8386 a proto je Ef = 20 nevyhovující. Naopak příliš vysoká hodnota Ef = 300, má sice velice vysoukou přesnost 0,9980, bohužel čas operace je v tomto případě zbytečně příliš vysoký 2,8ms na to jaký nárůst přesnosti získáme. Pro představu oproti Ef = 20, se u Ef = 300 čas operace zvedl 6,8x a přesnost stoupla o 15,94%. Proto je i Ef = 300 nevyhovující. Z měření lze vidět že ideální hodnota Ef pokud nám závisí na přesnosti nad 99,5% je okolo 200, získáváme vysoukou přesnost 0,9957 za přijatelný čas vykonání operace 1,9ms. Při porovnání s Ef = 20, dojde k 4,8x zpomalení ale přesnost se zvedne o 15,7%. Nicméně hodnota Ef = 200 má stále relativně vysoký čas vzhledem k násrůstu přesnoti, takže pokud bychom upřednostnili hlavně vysokou propustnost a spokojili se s přesností 98% tak bychom mohli volit i podstatně nižší Ef. Při Ef = 100 máme přesnost 0,9831 a čas 1,2ms, opět při porovnání s Ef = 20 nám dochází k zpomalení, tentorkát pouze 2,8x a přesnost nám stoupá o 14,5%. Z měření tedy můžeme vyvodit že pro naši kolekci a vybrané parametry vytváření je pro KNN vhodné Ef v rozmezí od 100-200 dle toho zda preferujeme propustnost nebo přesnost.

Přesnost	Ef	Prům. čas [us]	Min. čas [us]
0,83862	20	413,289	412
0,8953	30	517,506	515
0,92742	40	618,912	616
0,94677	50	705,307	704
0,95976	60	796,93	796
0,96877	70	887,826	884
0,97486	80	978,329	978
0,97943	90	1068,17	1066
0,98313	100	1158,61	1155
0,98599	110	1245,67	1244
0,98806	120	1332,38	1328
0,98971	130	1413,94	1411
0,99114	140	1500,58	1496
0,99243	150	1585,35	1582
0,99327	160	1670,44	1668
0,99414	170	1750,63	1747
0,99475	180	1832,2	1827
0,99524	190	1915,4	1912
0,99571	200	1997,4	1993
0,99618	210	2072,92	2068
0,99633	220	2152,36	2150
0,99672	230	2233,03	2225
0,99693	240	2327,24	2308
0,99703	250	2382,41	2374
0,99738	260	2540,25	2529
0,99764	270	2609,32	2605
0,99788	280	2671,21	2669
0,99794	290	2752,72	2748
0,99806	300	2832,61	2829

Tabulka 3.1: Nárůst přesnosti a času operace u HNSW_KNN s rostoucím Ef



Obrázek 3.6: Graf závislosti času na přesnosti

Kapitola 4

Filtr

Pod pojmem filtr si můžeme představit booleovskou podmínku omezující hodnoty jednotlivých atributů (dimenzí) prvků v n-dimenzionálním prostoru. Konkrétně tedy filtr říká jaké hodnotě se mají jednotlivé atributy rovnat, případně v jakém intervalu by se měly atributy pohybovat. Toto platí pouze pro atributy jež filtr omezuje, atributy které filtr nezmiňuje vůbec nekontrolujeme a akceptujeme je bez závislosti na jejich hodnotě.

U KNN slouží filtr pro omezení vyhledání výsledných K prvků. Při vyhledávání výsledku se prochází i prvky které filtru nevyhovují, ale do vráceného výsledku jsou vloženy pouze prvky jejichž atributy vyhovují omezení filtru.

Selektivita filtru [6] je číslo v rozsahu od 0 do 1 a udává procentuální počet prvků, které filtr přijme. Čím více je filtr vybíravý tím bližší hodnota k 0 mu bude přiřazena a tím více je filtr selektivnější. Naopak filtry přijímající většinu prvků budou mít přiřazeny číslo blíže k 1 a jejich selektivita bude tedy klesat, s tím že filtry přijímající úplně všechny prvky budou mít hodnotu rovno 1.

4.1 Implementace filtru a využití v HNSW

Filtr se implementoval jako vector objektů třídy VecDim. Tato třída reprezentuje jeden atribut a obsahuje ID dimenze, 0 - (vecDim - 1), vectory hodnot kterým má atribut (daná dimenze) nabývat nebo intervaly (tuple hodnoty od do) ve kterých se má atribut nacházet. Filtr je tedy nakonec vector který pro všechny atributy které chceme omezovat obsahuje hodnoty a intervaly kterým daná atribut musí vyhovovat, tedy minimálně jedné z těchto hodnot. Atributy které nijak neomezujeme ve vectoru atributů vůbec nejsou, porovnáváme jen ty atributy které nás ve filtru zajímají, ty ostatní přeskakujeme. Pro ověření zda daný prvek vyhovuje filteru jsem vytvořil pomoctou třídu VecDimHelper která pro daný prvek (vector hodnot) vrátí zda vyhovuje filteru nebo ne. Tato pomocná třída také generuje filtry dle určitých kritérií nebo umožnůje parsovat filtry z textové podoby a naopak.

V HNSW se filtr využívá u metody KNNFilter. Této metodě oproti její verzi bez filtru musíme tedy krom queryNodu, K a Ef i samotný filtr. Následně funguje stejně jen s tím rozdílem že používá vlastní SearchLayerFilter pro získání F nejbližších prvků. Tato metoda funguje obdobně jako její verze bez filtru SearchLayerKNN. Rozdíl v implementaci s filtrem je v tom že nevracíme W nejbližších nalezených prvků ale F nalezených nejbližších prvků které zároveň vyhovují filtru. Na začátku tedy do F uložíme entryPoint (jediný prvek v W) pokud splňuje podmínku filtru v opačném případě začíná průchod jako v popsaném Search v sekci HNSW. Rozdíl je v tom že jedna z ukončovacích podmínek průchodu je původně vzdálenost nejbližšího prvku z C je větší než vzdálenost nejvzdálenějšího prvku z W, v případě SearchLayerFilter je tato ukončovací podmínka rozšířena o to že zároveň musí platit že velikost F je rovna zadanému K. Tím pádem je zaručeno že při průchodu se prvky prochází dokud nenalezneme K hledaných prvků vyhovujících filtru nebo již nemáme co procházet a C je prázdné. Následně je ještě algoritmus rozšířen o část ve které do F vkládáme nově nalezené prvky e (e jsou sousedi nejbližších prvků c z C) pokud tento prvek filtr přijímá a zároveň je vzdálenost e menší než vzdálenost nejvzdálenějšího prvku z F nebo je velikost F menší než K.

4.2 Měření HNSW implementace s filtrem

Při měření se provádělo 1000 testů pro každé K a různé selektivity, z výsledků těchto testů se následně získával průměr. U rozšířené HNSW_KNN implementace s filtrem (F) byla hodnota Ef nastavena pevně na 200. U implementace bez filtru (B) získáváme z metody KNN Kb prvků, a v těchto prvcích následně kontrolujeme zda se v nic hnachází alespoň K prvků vyhovujících filtru a ty následně vracíme jako výsledek. Implementace bez filtru měla Ef nastavené také na 200, ale pokud v získaném Kb nebylo alespoň K prvků vyhovujících filtru, tak se zvedalo Kb, a pokud bylo $Kb > Ef$ tak se hodnota Ef nastavovala na hodnotu Kb (Ef nemůže být nikdy menší než K protože z nalezeného Ef se vrací K nejbližščích prvků). Pro získání filtrů s určitou selektivitou se provádělo náhodné generování filtrů, spočítání kolika prvkům z celkových dat filtry vyhovují a pokud selektivity spadaly do požadovaných hranic tak se textové reprezentace filtrů ukládaly do souborů a následně použili při testování. Veškeré výsledky testů jsou zaznamenány v tabulce (4.2).

V tabulce (4.2) lze vidět že čím nižší je selektivita filtru tím kratší dobu operace KNN zabírá u implementace s filtrem i bez něj. Dokonce při $K << Ef$ dochází k tomu že filtry se selektivitou 50%, 75% a 90% mají praticky stejné časy u jednotlivých implementací. Je tomu tak protože ve vyhledaném Ef prvků se nalezne požadované K které vyhovuje filtru a tím pádem není potřeba provádět průchody navíc pro dohledání potřebného počtu prvků do K. Konkrétně pro $K = 10, Ef = 200$ trvá KNN přibližně 2,8ms pro všechny selektivity u implementace s filtrem (F). Implementace bez filtru (B) je obecně rychlejší při nízkém K se selektivitě od 50% do 90% za předpokladu že známe požadované potřebné Kb ve kterém najdeme K hledaných prvků, což by v praxi úplně neplatilo protože nevíme kolik přesně prvků musíme vrátit a muselo by tedy dojít k přibližným odhadům případně k opakovanému vyhledávání. Implementace B má čas operace přibližně 2,1ms

pro nízké selektivity, 2,7ms pro selektivitu 25% což je srovnatelné s časem 2,8ms u implementace F. Každopádně u vysokých selektivit vidímě že implementace B se vyplatí, má totiž 2,3x lepší čas než implementace B, konkrétně 3ms oproti 7ms. Stejně tak při stoupajícím K a konstantním Ef můžeme vidět že implementace F má lepší časy KNN operací než B i pro nižší selektivity.

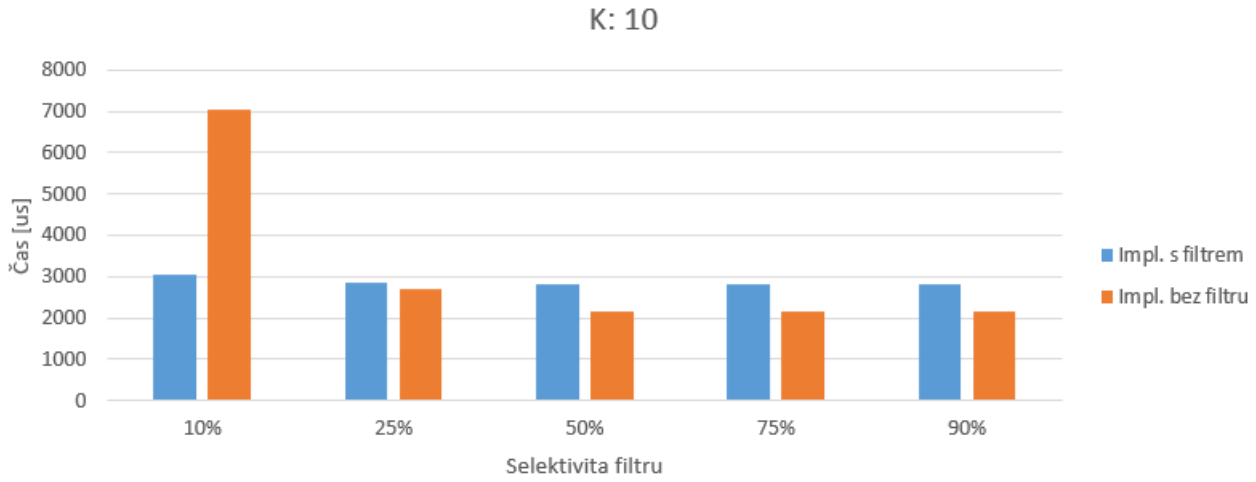
Z grafů porovnání průměrného času KNN operace implementace s a bez filtru pro různá K (4.2) můžeme vidět již zmíněné chování. Tedy to že když známe Kb pro impl. B tak pro K = 10 (4.1) je impl. F rychlejší pouze pro vysokou selektivitu 10%. S rostoucím K = 50 (4.2) je impl. F rychlejší až do selektivity 25%. Při K = 100 (4.3) je impl. B rychlejší až do selektivity 50% a nakonec u K = 200 (4.4) je impl. F rychlejší až do 75% s tím že u selektivity 90% mají impl. F i B relativně srovnatelné časy 3,05ms (F) a 2,81ms (B). Každopádně pro vysoké selektivity okolo 10% je impl. F vždy rychlejší než impl. B i když známe potřebné Kb pro impl. B.

Dále lze vidět že s rostoucím K a konstantním Ef se stává že získaný počet KNN v implementaci F je o něco málo menší než bylo požadované K a je proto potřeba Ef zvyšovat při rostoucím K (4.3). Implementace s filtrem vrací obdobné výsledky jako ta bez filtru, pokud se získané výsledky liší tak kvůli tomu že implementace bez filtru vyhledává až příliš prvků (může mít mnohem vyšší Ef) a tím pádem zahodí ty které jsou daleko od queryNodu a implementace s filtrem si tyto prvky ponechala a ukončila se jakmile našla požadované K v Ef prvcích.

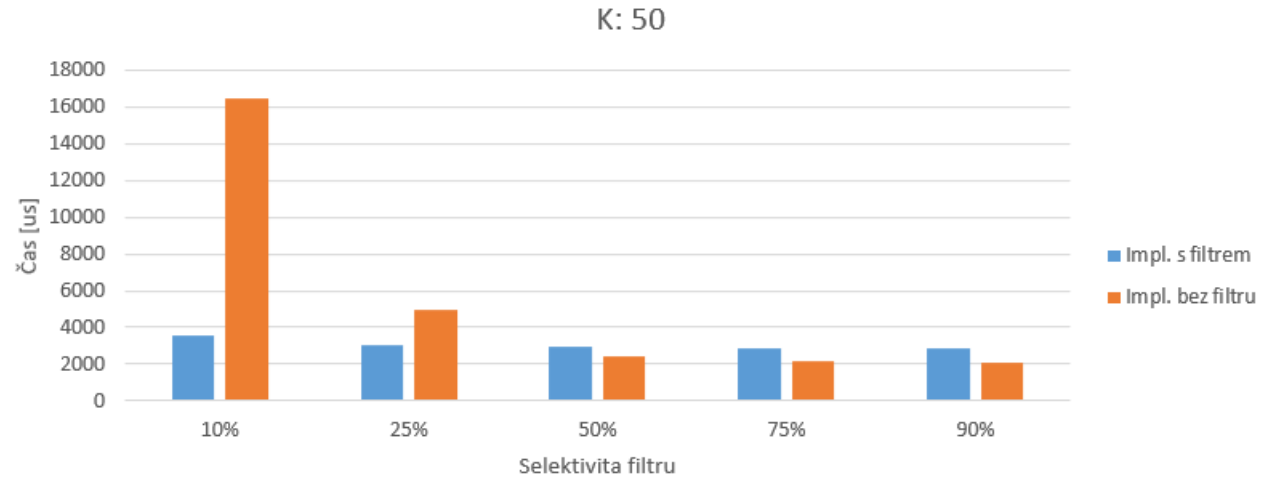
Sel[%]	tF[us]	tB[us]	Shoda FB	Získ. KNN u F	Prům. potř. Kb u B
K: 10					
10	3040,97	7056,26	0,91	10,00	715,28
25	2867,57	2703,60	0,99	10,00	124,00
50	2817,17	2168,04	1,00	10,00	25,96
75	2805,95	2163,28	1,00	10,00	15,83
90	2811,61	2163,50	1,00	10,00	13,34
K: 50					
10	3534,66	16451,46	0,73	49,67	1800,22
25	3076,86	4966,35	0,93	49,96	426,49
50	2921,99	2435,99	0,99	50,00	105,44
75	2903,95	2129,43	1,00	50,00	71,57
90	2891,03	2113,17	1,00	50,00	58,76
K: 100					
10	3961,65	25212,24	0,58	98,77	2747,15
25	3268,71	8161,75	0,85	99,88	753,35
50	2999,55	2952,85	0,97	100,00	221,45
75	2946,42	2226,74	1,00	100,00	160,95
90	2930,98	2117,40	1,00	100,00	142,65
K: 150					
10	4292,07	32129,19	0,50	147,38	3547,55
25	3411,34	11481,21	0,76	149,78	1060,20
50	3089,03	3804,70	0,95	150,00	319,30
75	3036,32	2642,55	0,98	150,00	226,25
90	2992,38	2173,99	1,00	150,00	197,20
K: 200					
10	4578,50	37901,86	0,45	195,29	4234,40
25	3587,91	14764,61	0,68	199,68	1367,60
50	3175,45	5156,43	0,90	200,00	417,50
75	3096,15	3594,97	0,95	200,00	290,35
90	3051,25	2810,96	0,99	200,00	251,65

*F = implem. KNN s filtrem, B = impl. KNN bez filtru, sel = selektivita filtru, tF = prům. čas F, tB = průměrný čas B, Shoda FB = shoda získaných prvků z F a B, Získ. KNN u F = prům. počet získaných prvků ze zvoleného K u F, Prům. potř. Kb u B = průměrný počet zadaného Kb v B pro získání K prvků

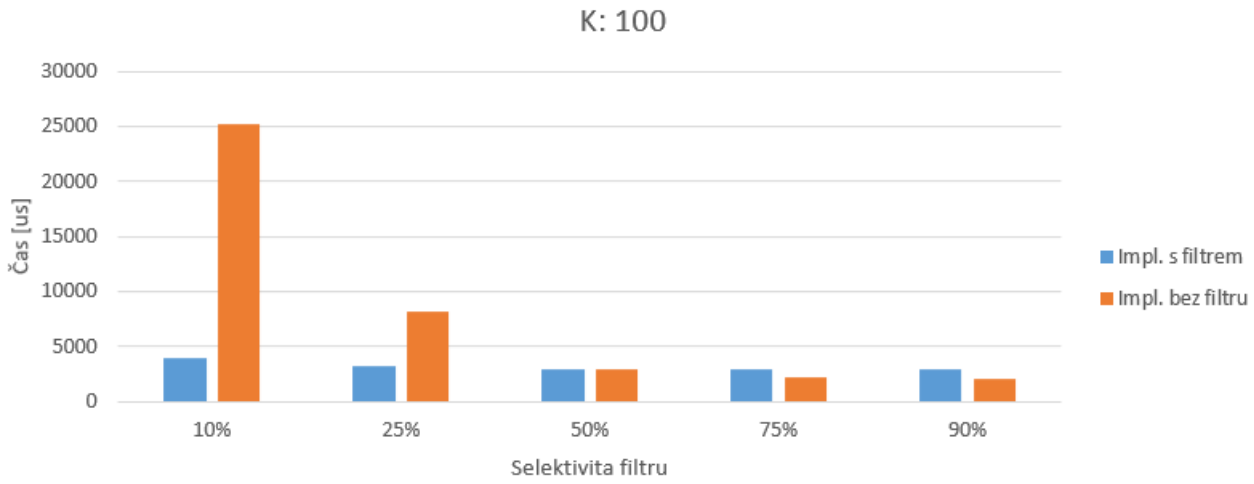
Tabulka 4.1: Porovnání vlastností u implementase KNN s filtrem a bez filtru



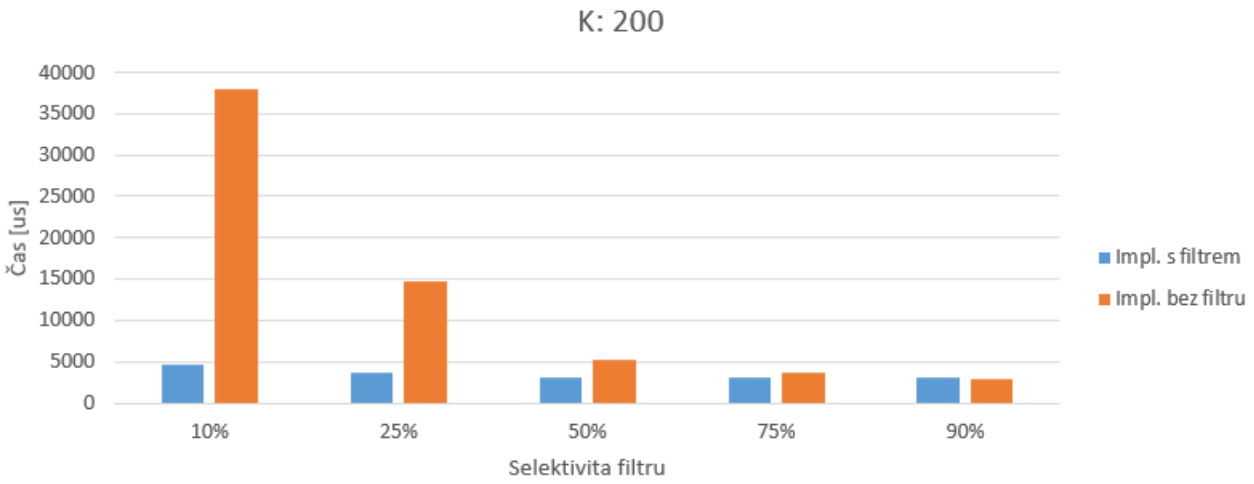
Obrázek 4.1: Porovnání průměrného času operace u implementací F a B, K: 10



Obrázek 4.2: Porovnání průměrného času operace u implementací F a B, K: 50



Obrázek 4.3: Porovnání průměrného času operace u implementací F a B, K: 100



Obrázek 4.4: Porovnání průměrného času operace u implementací F a B, K: 200

4.3 Problém filtru s vysokou selektivitou u HNSW

Při použití implementace HNSW s využitím filtru se může stát že nám metoda KNNFilter nevrátí K hledaných prvků, ale výsledný počet hledaných prvků bude o něco málo nižší (4.2). Pokud tak nastane tak nám ve výsledku chybí nízky počet prvků, nejčastěji 1 nebo 2 a pravděpodobnost že k tomuto jevu dojde je relativně nízká.

Tento jev nastává ve chvíli kdy hodnota K je stejně vysoká nebo jen o něco málo nižší než Ef a zároveň je selektivita filtru vyšší, do 25%.

K tomuto problému nastává protože HNSW algoritmus pro procházení grafu a výběr nalezených prvků použitý v metodě SearchKNNFilter lze zakončit dvěma způsoby. První způsob jak zakončit algoritmus průchodu který nás aktuáně nezajímá je ten kdy vzdálenost nejbližšího prvku z C je vyšší než vzdálenost nejvzdálenějšího prvku z W a zároveň je počet prvků v F (původně W) roven zadanému K.

Druhý způsob jak ukončit průchod, ten který způsobí problém, je ten kdy je C prázdný. Stane se tedy že projdeme prvky z C a všechny jejich nenavštívené sousedy kteří jsou blíže než nejvzdálenější prvek z W. Během tohoto průchodu jednoduše nenarazíme na dostatečný počet prvků které by vyhovovaly filtru a F obsahuje tedy méně nejbližších prvků než je počet co hledáme.

Tento problém lze jednoduše vyřešit tím že zvýšíme hodnotu Ef aby byla více rozdílna a vyšší než hodnota K. S roustoucím Ef při zachování K a filtru poroste i čas vykonání operace, ale bude se zvyšovat pravděpodobnost že získáme přesně K prvků ve výsledku a poroste i přesnost operace.

Kapitola 5

Závěr

K závěru bych řekl že část práce určená pochopení a implementaci HNSW byla mnohem pracnější než následné rozšíření o filtry. Je tomu tak protože rozšíření nevyžaduje mnoho nových implementací stačí jen již naimplementované algoritmy rozšířit o pár podmínek.

Dále bych chtěl zmínil že implementace takovýchto problémů má velice složité debugování a to hned z několika důvodů. Data jsou obrovská a stává se že implementace je obdobná s referencí do doby než se vloží prvek který je v datech daleko. Zabírá to hlavně spoustu času z důvodu dlouhého vytváření gragů a následného hledání kde přesně konkrétně došlo k rozdílu. Ve filnální implementaci se využívá i náhodné vybrání vrstev a pokud dva prky mají stejnou vzdálenost tak může dojít k rozdílným výsledkům.

V práci jsem až moc častou využíval std::vector, použití jednoduchých polí ať už staticky nebo dynamicky alokovaných by přineslo v určitých místech lepší propustnosti při správném využití nebo by graf zabíral méně paměti.

Práce mi určitě přinesla lepší pochopení C++, manipulaci s daty a důležitost jednotlivých rozhodnutí ohledně využití datových struktur nebo algoritmů pro zlepšení výsledné propustnosi a potřebné paměti. Projekt mě bavil a mimo složitý debug jsem si ji užil a jsem rád že si tento projekt zvolil. KNN je zajímavý problém a HNSW je zajímavý, jednoduchý a zároveň efektivní způsob jak jej řešit.

Literatura

1. MALKOV, Yu A; YASHUNIN, Dmitry A. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*. 2018, roč. 42, č. 4, s. 824–836.
2. *git-hnswlib: hnswlib* [online]. 2022. [cit. 2022-03-06]. Dostupné z: `https://github.com/nmslib/hnswlib`.
3. *git-hnsw: hnswlib* [online]. 2022. [cit. 2022-04-10]. Dostupné z: `https://github.com/RadimBaca/HNSW`.
4. *ann-benchmarks* [online]. 2022. [cit. 2022-03-06]. Dostupné z: `http://ann-benchmarks.com/index.html`.
5. *Nearest neighbor search* [online]. 2022. [cit. 2022-03-06]. Dostupné z: `https://en.wikipedia.org/wiki/Nearest_neighbor_search`.
6. *Optimalizace v INFORMIXU* [online]. 2022. [cit. 2022-04-04]. Dostupné z: `http://www.ms.mff.cuni.cz/~jkoc5219/Optimalizace_v_INFORMIXU.html`.