Katedra Informatiky
Fakulta Matematiky, Fyziky a Informatiky
Univerzita Komenského, Bratislava

Implementácia algoritmov pre projekt Boost

(bakalárska práca)

Marek Ludha

Vedúci: Mgr. Peter Košinár

Bratislava, 2007

Čestne prehlasujem, že som túto diplomovú prácu vypracoval samostatne s použitím citovaných zdrojov.

. . . . . .

Poďakovanie

Jediný rozhovor s múdrym človekom je prínosnejší ako desať rokov štúdia.
čínske príslovie

Chcem sa poďakovať môjmu školiteľovi Mgr. Petrovi Košinárovi za rady pri tvorbe tejto práce, Mgr. Michalovi Foríškovi za poskytnutie templatov pre sadzbu v programe TEXa Martinovi Rejdovi za pomoc pri zvládaní rôznych zvláštností jazyka C++.

Abstract

The goal of this thesis is to implement basic graph algorithms with regard to their I/O complexity. We decided to implement them in such a way that they are easily integrated with Boost library for the sake of better code reusability. In the first part we describe the background situation and possible utilization. In the second chapter we define I/O complexity and we study external algorithms DFS, BFS and mergesort. We also take a short glance at programming paradigms used throughout the Boost library. The third chapter contains design of code structure and implementation decisions. User's documentation can be found in the last chapter.

Keywords: external algorithms, graph algorithms, I/O complexity, generic programming

Abstrakt

Slabým miestom aplikácii spracovávajúcich veľké množstvo dát zvykne byť nemožnosť uložiť všetky dáta do operačnej pamäte a následné veľké množstvo I/O operácii, ktoré je spojené s prístupmi k týmto dátam. Cieľom tejto práce je implementovať základné grafové algoritmy s ohľadom na ich I/O zložitosť. Pre lepšiu znovupoužiteľnosť kódu sme ich implementovali tak, aby boli ľahko začleniteľné do knižnice Boost. V prvej časti sa zaoberáme popisom situácie na pozadí tohoto problému a možnosťou ich využitia. V druhej kapitole definujeme pojem I/O zložitosti a študujeme externé algoritmy DFS, BFS a mergesort. V krátkosti tu nahliadneme do programovacích paradigiem použitých pri návrhu knižnice Boost. Tretia kapitola obsahuje návrh štruktúry kódu a prijaté implementačné rozhodnutia. Používateľskú dokumentáciu možno nájsť v poslednej kapitole.

Kľúčové slová: externé algoritmy, grafové algoritmy, I/O zložitosť, generické programovanie

Obsah

Kapitola 1

Úvod

640kB ought to be enough for anybody.
B. Gates [1]

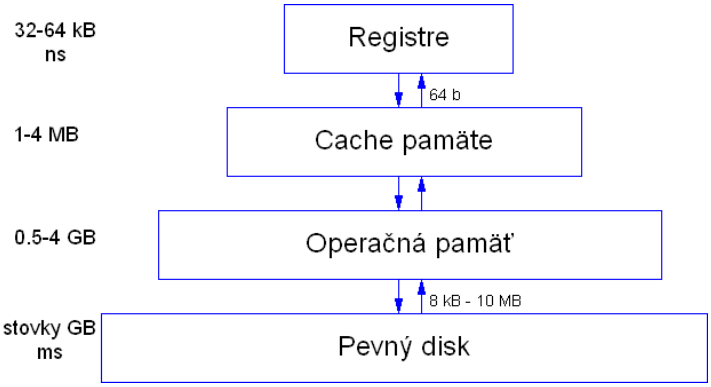
1.1 Motivácia

Táto kapitola je spracovaná podľa [Vit01].

Z ekonomických dôvodov používajú dnešné bežne dostupné počítačové systémy hierarchiu pamätí. Každá úroveň je charakteristická výrobnou cenou prepočítanou na jednotku úložného priestoru, výkonom (prenosovou rýchlosťou a dobou odozvy) a úložnou kapacitou. Na najnižšej úrovni sa nachádzajú registre a cache pamäte procesora. Dosahujú najvyšší výkon a sú najdrahšie na výrobu. Ako operačná pamäť sa zvyčajne používa pamäť typu DRAM. Na najvyššej úrovni sa používajú pevné disky, ktoré si uchovávajú svoj stav aj po odpojení napájania. Obrázok 1.1 znázorňuje hierarchiu pamätí ako aj charakteristiky jednotlivých úrovní.

Bežne používané programovacie jazyky používajú na adresovanie tejto hierarchie pojem virtuálnej pamäte. Tento model predpokladá, že pamäť je organizovaná lineárne a je možné adresovať ju pomocou jednej celočíselnej adresy. Táto funkcionalita je zabezpečená z časti procesorom a z časti operačným systémom. Procesor sa stará o uchovávanie posledne používaných dát v cache pamäti a prednačítavanie dát, o ktorých existuje predpoklad, že budú v krátkom čase potrebné (napríklad inštrukcie nasledujúce po aktuálne spracovávanej inštrukcii). Operačný systém sa stará o presuny medzi operačnou

[1] v roku 1981, citát z [Arg96]



Obr. 1.1: Hieararchia pamätí. Vľavo sú úložné kapacity a prístupové doby, na rozhraniach úrovní sú typicky používané veľkosti blokov.

pamäťou a stránkovacím súborom (pagefile) a obsluhuje výpadky stránky (situácia, keď proces požaduje prístup k stránke, ktorá bola presunutá do stránkovacieho súboru). Tento prístup k adresácii vďaka svojej elegancii zvyšuje produktivitu programovania. Navyše za predpokladu, že spracovávané dáta sú rádovo rovnako veľké ako operačná pamäť, je možné pomocou neho vytvoriť efektívne fungujúce programy. Z týchto dôvodov zaznamenal v softwarovom priemysle rozsiahly úspech.

Zo spôsobu fungovania virtuálneho adresného priestoru ale vyplýva, že nie všetky pamäťové prístupy sú rovnocenné čo sa týka doby odozvy. Veľké adresné priestory typicky presahujú aj do najvyššej úrovne pamäťovej hierarchie, pričom rýchlosť prístupu do tejto úrovne je o niekoľko rádov menšia ako prístup do najnižších úrovní. Napríklad načítanie registra procesora trvá rádovo nanosekundy (10^{-9} sekundy), prístup do operačnej pamäte trvá rádovo desiatky nanosekúnd, ale opozdenie pri načítavaní z pevného disku je niekoľko milisekúnd (10^{-3} sekundy). To znamená asi milionkrát pomalšie.

K zväčšovaniu tohoto rozdielu ešte prispieva súčasný trend spomalenia rastu prenosových rýchlostí pevných diskov. Kým procesory zaznamenávajú nárast výkonu 40%-60% ročne a kapacity pevných diskov rastú o 60%-80% ročne, nárast výkonu pevných diskov je len okolo 7%-10% [RW94]. Z týchto dôvodov sú prenosy medzi operačnou pamäťou a pevným diskom bottleneckom aplikácii, ktoré spracovávajú veľké objemy dát.

Mnohé programy vykazujú istú lokálnosť pri prístupe k pamäti: chvíľu pracujú s istou súvislou množinou dát a potom prenesú pozornosť na ďalšiu

množinu. Operačné systémy využívajú túto predvídateľnosť prístupov k pamäti tým, že pre každý bežiaci proces udržujú v operačnej pamäti pracovnú množinu stránok (working set). Tá zhruba zodpovedá nedávno používaným stránkam. Pri rozhodovaní o tom, ktorú stránku zaradiť do pracovnej množiny, sa používajú heuristiky navrhnuté so zámerom minimalizovať výskyt výpadkov stránky.

Tieto heuristiky sú navrhované všeobecne, teda tak, aby sa dokázali vysporiadať s každou situáciou. Preto nemožno očakávať, že naplno využijú lokálnosť každého algoritmu. Niektoré algoritmy sú od návrhu nelokálne, preto aj heuristika, ktorá by poznala presnú postupnosť požiadaviek na stránky by musela spraviť veľké množstvo I/O operácii. V takýchto prípadoch nie je možné dosiahnuť zlepšenie bez zásahu do samotného algoritmu. Z týchto dôvodov je nutné navrhovať algoritmy spracovávajúce veľké množstvo dát so zreteľom na počet I/O operácii a takisto je nutné explicitne riadiť prenosy medzi pevným diskom a operačnou pamäťou.

1.2 Využitie

Algoritmy spracovávajúce veľké množstvá dát sa dajú nájsť napríklad v aplikáciach z oblasti databázových systémov, geografických informačných systémov, návrhu, simulácie a verifikácie VLSI čipov, logického programovania s obmedzeniami, počítačovej grafiky a virtuálnej reality, výpočtovej biológie, fyziky a geofyziky a v meteorológii. Dobrý príklad takéhoto systému je projekt NASA nazývaný EOS, ktorý je zameraný na sledovanie zemského povrchu, oceánov a biosféry. Očakáva sa, že bude spracovávať až petabyty dát.

Kapitola 2

Prehľad problematiky

> The difference in speed between modern CPU and disk technologies is analogous to the difference in speed in sharpening a pencil by using a sharpener on one's desk or by taking an airplane to the other side of the world and using a sharpener on someone else's desk.
>
> **D. Cormer** [1]

2.1 Diskový model

Externé algoritmy popisované v tejto práci explicitne narábajú s presunmi a rozmiestnením dát. Preto je nutné mať primerane presný a jednoduchý diskový model, pomocou ktorého bude možné určovať efektívnosť jednotlivých algoritmov. Táto kapitola je spracovaná podľa [CGG+95] a [RW94].

Pevný disk pozostáva z niekoľkých rotujúcich záznamových platní. Na záznam sa typicky používajú obidva povrchy platní. Na každom záznamovom povrchu týchto platní sa nachádza samostatná hlava, ktorá uskutočňuje čítanie aj zapisovanie. Dáta sú uchovávané v sektoroch, ktoré sú rozmiestnené do sústredných kružníc nazývaných stopy. Stopy, ktoré možno čítať pri určitej pozícii hláv bez ich presunu, sa nazývajú cylinder. Pri požiadavke na čítanie alebo zapisovanie konkrétnej adresy sa musí hlava presunúť nad príslušný cylinder a počkať, kým požadovaná adresa prejde popri hlave v dôsledku rotácie platní. Mechanizmus ovládajúci pohyb hláv umožňuje iba presun všetkých hláv naraz. Čas potrebný na presun hlavy medzi dvomi náhodnými cylindrami je väčšinou medzi 3 až 10 milisekundami. Takisto priemerný čas

[1] citát z [Arg96]

potrebný na presun požadovaného sektoru pod hlavu (polovica času potrebného na úplné otočenie platne) sa pohybuje rádovo na tejto úrovni. Aby bol tento spôsob fungovania efektívnejší, pri požiadavke na čítanie sa prenášajú veľké skupiny súvislých dát, ktoré budeme nazývať bloky.

Hlavná vlastnosť modelovaných diskov, ktorú bude použitý model zahŕňať, je dlhý prístupový čas v porovnaní s prístupovými časmi do operačnej pamäte. Používa nasledujúce parametre:

- N - počet objektov v inštancii problému
- M - počet objektov ktoré sa zmestia do operačnej pamäte
- B - počet objektov ktoré sa zmestia do jedného bloku

pričom $M < N$, $1 \leq B \leq \frac{M}{2}$ a M je celočíselný násobok B.

Keďže prístupový čas k dátam na pevnom disku je rádovo väčší ako čas strávený procesorom pri výpočtoch, budeme posudzovať efektívnosť algoritmov iba na základe asymptotického odhadu množstva vykonaných I/O operácii. Ukazuje sa, že algoritmy optimálne vzhľadom na I/O zložitosť sú často optimálne aj vzhľadom na časovú zložitosť. Za I/O operáciu budeme považovať čítanie alebo zapisovanie B objektov nasledujúcich za sebou. Zložitosť použitých algoritmov budeme vyjadrovať pomocou zložitostí dvoch základných operácií, scanning a sorting. Použijeme pre ne nasledujúce skratky:

$$scan(x) = \frac{x}{B}$$

zodpovedá počtu operácii, ktoré treba spraviť na prečítanie x po sebe zapísaných objektov

$$sort(x) = \frac{x}{B} \log_{\frac{M}{B}} \frac{x}{B}$$

zodpovedá počtu operácii, ktoré treba spraviť na utriedenie x po sebe zapísaných objektov.

2.2 Popis implementovaných algoritmov

2.2.1 DFS

Táto kapitola je spracovaná podľa [CGG+95]. Obsahuje popis externého algoritmu DFS, ktorý je použiteľný pre orientované aj neorientované grafy.

Poznámky k implementácii tohoto algoritmu sa dajú nájsť v kapitole 3.3.5. Používateľská dokumentácia je v kapitole 4.18.

Algoritmus DFS sa dá použiť na riešenie mnohých problémov na orientovaných grafoch, napr. hľadanie silno súvislých komponent, zisťovanie acyklickosti grafu a topologické triedenie.

Popisovaný algoritmus je jemnou modifikáciou klasického algoritmu DFS. Spočíva v tom, že keď počet navštívených vrcholov presiahne $\Theta(M)$, algoritmus modifikuje grafové dáta uložené na disku. Graf s V vrcholmi a E hranami budeme reprezentovať pomocou dvoch polí. Pole *edge* veľkosti E obsahuje koncové vrcholy hrán utriedené podľa počiatočných vrcholov. Pole $index[i]$ veľkosti V určuje index do poľa *edge*, od ktorého začínajú hrany vychádzajúce z vrchola i. Špeciálne hodnota $index[V]$ určuje koniec zoznamu hrán pre vrchol $V - 1$. Z toho vyplýva, že následníci vrcholu i sú vrcholy $\{edge[j] \mid index[i] \leq j < index[i + 1]\}$.

Pri prechode budeme udržiavať zásobník vrcholov, ktorý (pri čítaní od dna po vrch) zodpovedá ceste od počiatočného vrchola po aktuálny. Tento zásobník môže obsahovať až N vrcholov, preto je nutné udržiavať ho na disku.

Na začiatku bude počiatočný vrchol aktuálnym. Keď sa niektorý vrchol stane aktuálnym, uložíme ho na zásobník, zaradíme ho do zoznamu navštívených vrcholov a budeme prechádzať jeho zoznamom nasledovníkov, pričom postupne sa každý z nich stane aktuálnym vrcholom. Ak je niektorý nasledovník v zozname navštívených vrcholov, nenavštevujeme ho znova. Keď vyčerpáme zoznam nasledovníkov pre aktuálny vrchol, odstránime ho zo zásobníka a za aktuály vrchol považujeme vrchol na zásobníku pod ním. Algoritmus skončí odstránením posledného vrcholu zo zásobníka.

Jediný problém nastane, keď zoznam navštívených vrcholov zaplní voľnú pamäť. Vtedy prejdeme zoznamom hrán uloženým na disku a zahodíme tie hrany, ktoré končia vo vrchole, ktorý už bol navštívený. Tým zabezpečíme, že tieto vrcholy už viac nenavštívime. Preto môžeme zoznam navštívených vrcholov vyprázdniť.

Takýto algoritmus má zložitosť I/O operácii $O((1 + \frac{V}{M})scan(E) + V)$ [CGG+95].

2.2.2 Mergesort

Táto kapitola je spracovaná podľa [Arg96]. Obsahuje popis externého algoritmu mergesort. Poznámky k implementácii tohoto algoritmu sa dajú nájsť

v kapitole 3.3.3. Používateľská dokumentácia k implementácii je v kapitole 4.10.

Externý mergesort vychádza z klasického mergesortu. V prvej fáze načítava vstup po častiach veľkosti M, každú takúto časť utriedi a zapíše naspäť na disk. Tým vznikne $\lceil \frac{N}{M} \rceil$ utriedených podpostupností. V tejto fáze treba $O(scan(N))$ I/O operácii.

Druhá fáza prebieha po úrovniach. Na každej úrovni sa postupne zlúči M po sebe idúcich podpostupností do jednej, pričom na poslednej úrovni vznikne zlúčením jediná utriedená postupnosť. Podstatná skutočnosť v tejto fáze je, že s použitím pamäte veľkosti M možno zlúčiť $\frac{M}{B}$ postupností do jednej s použitím lineárneho počtu operácii v závislosti od počtu zlučovaných prvkov. Dosiahneme to tak, že z každej postupnosti načítame prvý blok a postupne budeme zapisovať skupiny B najmenších doteraz nezapísaných prvkov. Keď zapíšeme všetky prvky z niektorého bloku, načítame nasledujúci blok zo zodpovedajúcej postupnosti.

Na začiatku prvej úrovne je $\lceil \frac{N}{M} \rceil$ podpostupností. Na každej ďalšej úrovni klesne tento počet M/B-násobne, teda na začiatku i-tej úrovne ich bude rádovo $\frac{N/M}{(M/B)^{i-1}}$. Posledná úroveň je tá, na ktorej počet postupností klesne na 1, teda úrovní bude $O(\log_{\frac{M}{B}} \frac{N}{M})$. Zlučovanie prvkov na každej úrovni prebehne v lineárnom čase od ich počtu, teda tento algoritmus potrebuje $O(N \log_{\frac{M}{B}} \frac{N}{M})$ operácii.

2.2.3 BFS

Táto kapitola je spracovaná podľa [MR99]. Obsahuje popis externého algoritmu BFS pre neorientované grafy. Poznámky k implementácii tohoto algoritmu sa dajú nájsť v kapitole 3.3.4. Používateľská dokumentácia je v kapitole 4.15.

Algoritmus BFS je jedným zo základných grafových algoritomv a dá sa využiť pri riešení niektorých problémov, napríklad hľadanie najkratšej cesty v neohodnotenom grafe alebo zisťovanie bipartitnosti.

Použijeme rovnakú reprezentáciu grafu ako v kapitole 2.2.1. Množinu vrcholov vo vzdialenosti d od počiatočného vrcholu budeme značiť $front(d)$. Ďalej nech $nbr(x)$ je množina vrcholov susedných s x a nech $nbr(X)$ je multimnožina ktorá vznikne zjednotením $nbr(x)$ pre všetky $x \in X$.

Klasický BFS algoritmus pracuje nasledovne: Na začiatku je $front(0)$ jednoprvková množina obsahujúca iba počiatočný vrchol. Množina $front(t)$

je potom množina vrcholov, ktoré susedia s niektorým vrcholom z $front(t-1)$, ale ešte neboli označené ako navštívené. Tento prístup si vyžaduje pamätať si pre každý vrchol či už bol navštívený a adresovať dáta v tejto dátovej štruktúre v nepredvídateľnom poradí.

My budeme zostrojovať $front(t)$ nasledujúcou postupnosťou krokov:

1. zostrojíme $nbr(front(t-1))$
2. odstránime duplicitné prvky z postupnosti z kroku 1.
3. z postupnosti z kroku 2. odstránime tie prvky, ktoré sa nachádzajú v $front(t-1) \cup front(t-2)$

[MM02] uvádza nasledujúci argument o korektnosti takéhoto postupu: Predpokladajme, že množiny $front(0), \ldots, front(t-1)$ už boli vytvorené. Uvažujme o vrchole v, ktorý je sused vrchola $u \in front(t-1)$. Vzdialenosť z počiatočného vrchola do v je aspoň $t-2$. Keby táto vzdialenosť bola menšia, potom by sa do vrchola u dalo dostať na menej ako $t-1$ krokov (cestou cez v). Poznamenajme, že toto tvrdenie platí iba pre neorientované grafy (v orientovanom by sa z počiatočného vrcholu nemuselo dať dostať do u cez v). Toto je súčasne dôvod, prečo tento algoritmus nefunguje pre orientované grafy. Zároveň do u sa dá dostať na $t-1$ krokov a v je sused v, teda do v sa dá dostať najviac na t krokov. Z toho vyvplýva, že vrchol v musí byť v jednej z množín $front(t-2)$, $front(t-1)$, $front(t)$, teda každý sused vrchola z $front(t-1)$ je v jednej z týchto množín. Preto je korektné počítať $front(t)$ ako $nbr(front(t-1)) \setminus (front(t-1) \cup front(t-2))$.

Takýto algoritmus má zložitosť počtu I/O operácii $O(N + sort(N+M))$.

2.3 Paradigmy v knižnici Boost

Algoritmy implementované ako súčasť tejto práce boli navrhované tak, aby sa dali jednoducho integrovať do knižnice Boost. V tejto kapitole zhrnieme princíp generického programovania z pohľadu knižnice Boost, ktorý bol použitý aj pri ich implementácii. Vychádzame z dokumentácie, ktorú možno nájsť na stránke http://boost.org/libs/graph/doc/index.html.

Generické programovanie v jazyku C++ používa črtu jazyka nazývanú šablony. Pomocou nich je možné naprogramovať triedu alebo funkciu, ktoré pracujú s objektami, ktorých typ nie je pri implementácii známy, teda je možné použiť ich s viacerými rôznymi typmi, ktoré implementujú rovnaké

rozhranie. Samotný jazyk C++ ale neposkytuje žiadny mechanizmus na kontrolu, či použitý typ spĺňa požiadavky kladené na príslušné rozhranie. Bežná prax je vyjadriť tieto požiadavky v názve parametra šablony alebo v dokumentácii. S tým zvyknú byť spojené nasledujúce problémy:

- Dokumentácia je príliš všeobecná, neúplná alebo žiadna.
- V prípade nesplnenia požiadaviek je výsledná chybová hláška kompilátora ťažko pochopiteľná. Väčšinou totiž neukazuje na miesto volania šablóny, ale týka sa implementácie volanej funkcie.
- Požiadavky nie sú formulované priamo v kóde, čo znižuje jeho čitateľnosť a môže spôsobiť, že sa dostane do nesúladu s dokumentáciou.

V knižnici Boost sa za týmto účelom používajú špeciálne triedy na kontrolu typov. Spolu s preddefinovanými rozhraniami (nazývanými koncepty) tvoria Boost Concept Check Library. Pomocou tejto knižnice je možné do kódu vkladať výrazy, ktoré pri kompilácii kontrolujú, či všetky použité typy spĺňajú príslušný koncept. Tieto výrazy zodpovedajú štandardu C++ a nepredstavujú žiadnu dodatočnú záťaž pri behu programu. Všetka kontrola sa vykonáva pri kompilácii. Ak použitý typ nespĺňa príslušný koncept, výsledná chybová hláška bude ukazovať na miesto kontroly konceptu, teda bude z nej možné vydedukovať, že použitý typ nespĺňa niektorú požiadavku.

Generické programovanie spočíva v použití nasledujúcich princípov:

1. *Oddelenie algoritmov od dátových štruktúr*

   Algoritmy sú implementované tak, aby sa nespoliehali na spôsob, akým je konkrétna dátová štruktúra implementovaná. Namiesto toho s ňou komunikujú iba pomocou definovaného rozhrania. Príkladom takéhoto rozhrania môže byť iterátor, pomocou ktorého možno pracovať so všetkými prvkami štruktúry v určitom poradí. V praxy to znamená, že každý algoritmus a dátová štruktúra je implementovaná iba raz. Nie je teda nutné implementovať zvlášť prehľadávanie do hĺbky grafu reprezentovaného pomocou poľa následníkov a zvlášť prehľadávanie do hĺbky grafu reprezentovaného pomocou ukazovateľov na vrcholy. To umožní redukovať množstvo kódu z $O(M * N)$ na $O(M + N)$, kde M je počet algoritmov a N počet dátových štruktúr. Ďalšia výhoda tohoto prístupu spočíva v jednoduchom použití dát, ktoré boli vytvorené pre

inú knižnicu alebo pre spracovanie v inom programovacom jazyku. Takéto dáta nie je nutné konvertovať, stačí naprogramovať triedu, ktorá ich bude čítať a sprístupňovať pomocou definovaného rozhrania.

2. *Rozširovanie algoritmov pomocou funkčných objektov*

   Algoritmy implementované v knižnici Boost sú rozšíriteľné. To znamená, že funkcii implementujúcej príslušný algoritmus sa pri volaní predá objekt, ktorý spĺňa koncept príslušného visitora. Každý algoritmus má definované udalosti, ktore pri jeho behu budú nastávať (napr. objavenie nového vrchola pri prehľadávaní grafu). Pri vzniku takejto udalosti zavolá algoritmus príslušnú funkciu objektu, ktorý bol predaný ako visitor. To umožňuje volať používateľom definovaný kód pri vzniku udalostí, čo robí algoritmy veľmi flexibilnými.

3. *Parametrizácia typu prvku*

   Dátové štruktúry je možné spraviť nezávislé od typu prvkov ktoré ukladajú pomococu parametrizácie tohoto typu. Pri reprezentácii grafu často treba s vrcholmi alebo hranami asociovať určité hodnoty (vlastnosti). Pri použití parametrizácie nemusí ich typ byť dopredu známy.

Kapitola 3

Implementácia

Beware of bugs in the above code; I have only proved it correct, not tried it.
D. Knuth [1]

Kód popisovaný v tejto kapitole bol kompilovaný pomocou prekladača GCC 3.4.4 v prostredí Cygwin. Použité boli iba štandardné črty jazyka C++, preto by ho malo byť možné preložiť ľubovoľným prekladačom, ktorý tento štandard dodržuje. Boost je knižnica, ktorá pozostáva iba z hlavičkových súborov (header-only library), teda ju nie je možné predkompilovať. Takisto tento kód sa kompiluje až spoločne s programom, ktorý ho využíva.

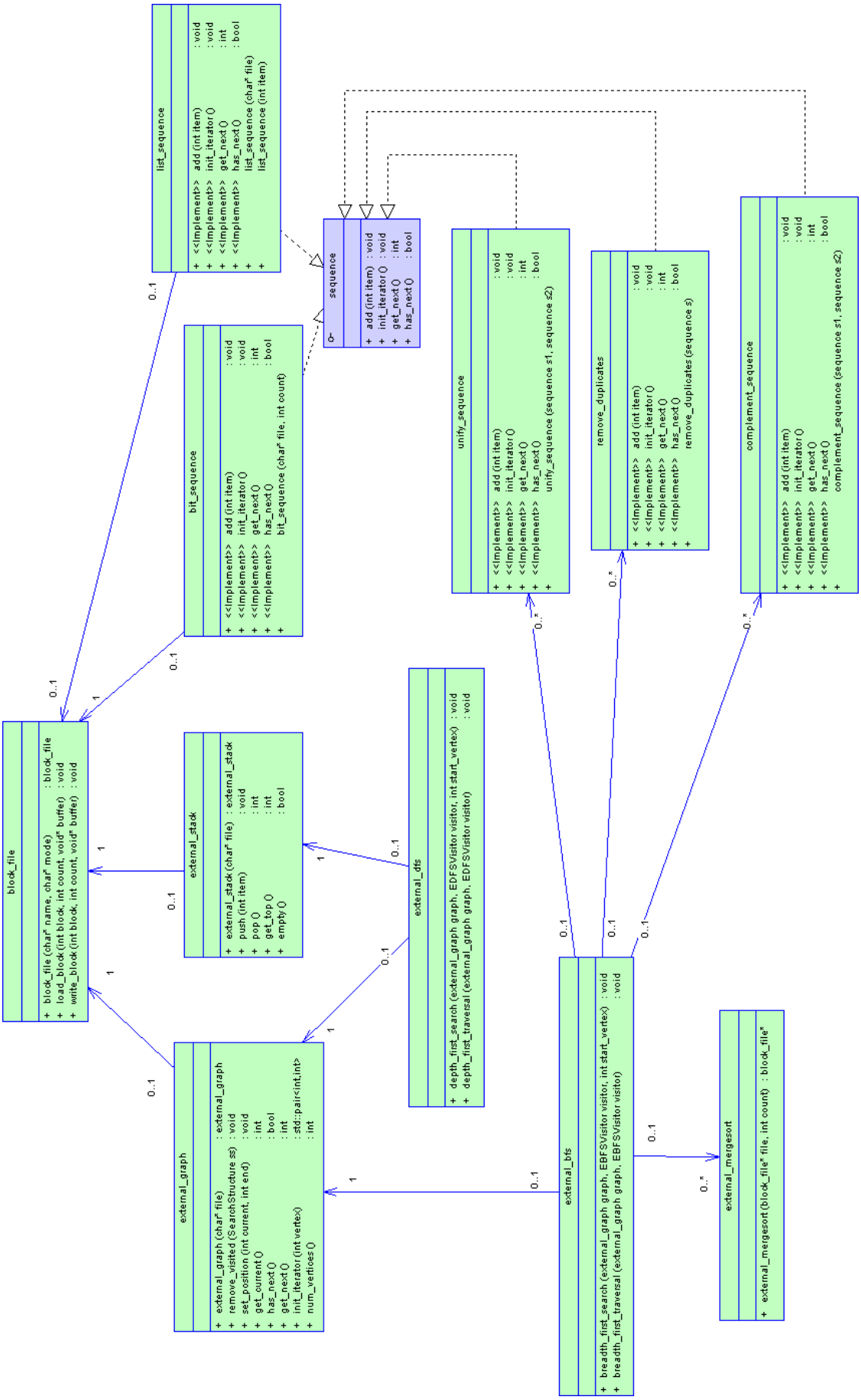
3.1 Návrh

Model štruktúry implementovaného kódu je znázornený na obrázku 3.1. Použitý je architektonický štýl layered (vrstvový). Na najnižšej vrstve je implementovaný prístup k diskovému systému pomocou triedy block_file. Úlohou tejto triedy je poskytovať abstrakciu nad dostupným hardwarom a zabezpečiť ľahkú modifikovateľnosť kódu (napríklad pri rozširovaní o podporu viacerých diskov).

Na vyššej vrstve sú triedy implementujúce dátové štruktúry, konkrétne ide o tieto:

- external_graph (orientovaný graf reprezentovaný zoznamom následníkov pre každý vrchol)

[1] citát z [Arg96]



Obr. 3.1: Class diagram 2

- external_stack (zásobník ukladajúci čísla)
- list_sequence (postupnosť čísel reprezentovaná zoznamom čísel)
- bit_sequence (postupnosť čísel reprezentovaná bitovým poľom)

Nad touto vrstvou sú implementované jednoduché operácie na manipuláciu s postupnosťami:

- unify_sequence (zjednotenie dvoch utriedených postupností do utriedenej postupnosti)
- remove_duplicates (odstránenie duplikátov z utriedenej postupnosti)
- complement_sequence (výsledkom je utriedená postupnosť, ktorá obsahuje prvky prvej postupnosti, ktoré nie sú v druhej postupnosti)
- external_mergesort (utriedi postupnosť list_sequence uloženú na disku externou verziou algoritmu mergesort)

Na najvyššej vrstve sú implementované samotné algoritmy na prechod grafom, konkrétne prehľadávanie do šírky (BFS) a prehľadávanie do hĺbky (DFS). Operácie external_depth_first_search a external_breadth_first_search uskotočnia prechod komponentou grafu, ktorá je daná počiatočným vrcholom. Operácie external_depth_first_traversal a external_breadth_first_traversal prejdú postupne každou komponentou súvislosti grafu. Výstup z tejto vrstvy je realizovaný volaním funkcií visitora pri dosiahnutí príslušných udalostí.

3.2 Použité formáty

Ako vstupný parameter algoritmov realizujúcich prechod grafom je možné použiť ľubovoľnú triedu, ktorá spĺňa koncept EGraphConcept. V rámci tejto práce sme implementovali takúto triedu s názvom external_graph, ktorá požadované dáta načítava zo súboru, ktorý dostane ako parameter v konštruktore. Tento súbor pozostáva z niekoľkých po sebe idúcich blokov, ktorých veľkosť je definovaná konštantou BLOCK_SIZE v súbore block_file.hpp. Používaná veľkosť typu pre celé čísla je 4B. Prvý blok obsahuje jediné číslo, počet vrcholov v grafe. Zvyšok je rezervovaný pre budúce použitie. Za týmto blokom nasleduje zoznam offsetov v rámci súboru, na ktorých končí zoznam nasledovníkov pre jednotlivé vrcholy. Tento zoznam obsahuje jednu položku

pre každý vrchol grafu a je zarovnaný na hranicu bloku. Za ním nasledujú zoznamy nasledovníkov pre jednotlivé vrcholy. Táto časť je presná kópia poľa *edge* z kapitoly 2.2.1. Každý zoznam nasledovníkov musí byť utriedený.

Takýto formát grafu zodpovedá funkčnosťou, časovou a I/O zložitosťou jednotlivýych operácii formátu, ktorý je popísaný v kapitole 2.2.1. Súbory v takomto formáte možno generovať programom generate. Ten číta textový popis grafu zo štandardného vstupu a výsledok zapisuje na štandardný výstup. Vstup pozostáva z niekoľkých čísel oddelených bielymi znakmi. Prvé číslo V predstavuje počet vrcholov v grafe. Za ním nasleduje V čísel, počty susedov pre vrcholy 0 až $V - 1$ v tomto poradí. Ďalej nasleduje $V - 1$ postupností zodpovedajúcich dĺžok, ktoré predstavujú zoznamy nasledovníkov pre príslušný vrchol.

3.3 Implementácia

Táto kapitola obsahuje problémy, na ktoré sme narazili pri implementácii algoritmov popísaných v kapitole 2.2. Dokumentácia k verejným metódam implementovaných tried sa nachádza v časti 4.

3.3.1 external_graph.hpp

Pri implementácii triedy external_graph vznikol problém so štandardným prístupom k čítaniu zoznamu susedov pomocou iterátora. Pri implementácii algoritmu DFS by bolo nutné mať vytvorený jeden iterátor pre každý vrchol na zásobníku. Týchto vrcholov môže byť až V, pričom súčasne potrebujeme pracovať iba s jedným iterátorom. Tento problém sme vyriešili tým, že každý objekt external_graph súčasne poskytuje aj rozhranie podobné rozhraniu iterátora a funkcie na zistenie a nastavenie stavu iterátora. Tento stav iterátora budeme ukladať na zásobník, čím napodobníme prácu rekurzívnej implementácie DFS s iterátormi uloženými v jednotlivých rámcoch na zásobníku. Elegantnejšie riešenie, ktorého implementácia predstavuje možnosť rozšírenia práce, spočíva v definícii všeobecného rozhrania na serializáciu objektov a jeho použití pri ukladaní serializovaných iterátorov na zásobník. Takisto by bolo nutné upraviť implementáciu zásobníka tak, aby na neho bolo možné ukladať ľubovoľný serializovateľný objekt.

3.3.2 external_sequence.hpp

Triedy implementované v tomto súbore riešia problém operácii s postupnosťami (zjednotenie, odstránenie duplikátov, rozdiel pri zachovaní utriedenia) metódou pipes-and-filters. Zdroje dát sú v tomto prípade triedy list_sequence a bit_sequence. Do nich je možné pridávať členy postupnosti pomocou funkcie add(int). Filtre sú triedy unify_sequence, complement_sequence a remove_duplicates, ktoré čítajú čísla z postupností predaných v konštruktore a prevádzajú s nimi príslušné operácie. Komunikácia medzi objektami prebieha pomocou rozhrania, ktoré je definované konceptom ESequenceConcept. Z tohoto prístupu vyplýva, že filtrujúce postupnosti si udržujú iba pozíciu v zdrojových postupnostiach, nevytvárajú si lokálnu kópiu ich dát. Preto je nutné zabezpečiť, aby pri každom použití filtrujúcej postupnosti existovali všetky zdrojové postupnosti, ktoré používa. Filtrujúce postupnosti takisto v deštruktore nevolajú deštruktor zdrojových postupností, ich pamäť je teda nutné uvoľniť zvlášť.

3.3.3 external_mergesort.hpp

Triedenie v rámci operačnej pamäte v prvej fáze bolo implementované pomocou štandardnej funkcie qsort(). Jednotlivé úrovne druhej fáze sú implementované pomocou for-cykla. Na začiatku každej iterácie sa prepočítajú parametre aktuálnej úrovne, ako poradové číslo, počet a maximálna dĺžka podpostupností. Vnorený for-cyklus iteruje cez všetky M-tice podpostupností, ktoré sa na danej úrovni budú zlučovať. Následný while-cyklus z nich vyberá najmenší prvok a zapíše ho do výstupnej postupnosti. Keďže najmenšie prvky vstupných postupností budú po vstupnom súbore rozptýlené, výstupná postupnosť sa zapisuje do nového súboru, ktorý bude po dokončení úrovne vymenený s vstupným. Prístup k M-tici práve zlučovaných postupností je realizovaný pomocou triedy mergesort_buffer. Táto obsahuje buffer na uloženie jedného bloku každej postupnosti a metódy na zistenie aktuálneho prvku postupnosti s daným indexom, zistenie či daná postupnosť má ďalší prvok a posun na ďalší prvok. Na pozadí zabezpečuje načítavanie ďalších blokov postupností a ošetruje výnimočné situácie (posledná postupnosť na každej úrovni zvykne byť kratšia ako ostatné, postupností môže byť menej ako maximálny počet). Pri prechode na ďalšiu úroveň alebo M-ticu postupností sa táto trieda inicializuje volaním funkcie set_level() alebo set_first_run().

3.3.4 external_bfs.hpp

Funkcia external_breadth_first_search() implementuje prehľadávanie zadanej komponenty súvislosti do šírky podľa kapitoly 2.2.3. Postupnosti $front$ budeme reprezentovať pomocou objektov triedy list_sequence. Objekt front_1 zodpovedá postupnosti $front(t)$, front_2 zodpovedá $front(t-1)$ a front_3 zodpovedá $front(t-2)$. Postupnosti $front(t)$ konštruujeme v cykle while postupne pre rastúce t dovtedy, kým niektorá z vytvorených postupností nebude prázdna. Postupnosť $nbr(front(t-1))$ vytvoríme spojením zoznamov nasledovníkov pre všetky vrcholy, ktoré sa nachádzajú v $front(t-1)$. Postupnosť nbr následne utriedime algoritmom mergesort. Potom vytvoríme filtrovaciu postupnosť, ktorá odstráni duplikáty z nbr a ponechá iba tie prvky, ktoré nie sú ani v front_2 ani v front_3 (použitím remove_duplicates, complement_sequence a unify_sequence). Túto postupnosť prenesieme po jednotlivých prvkoch do front_1 (po zmazaní front_3 by filtrujúca postupnosť prestala fungovať, preto ju treba uložiť do súbora). Tento krok nepredstavuje žiadnu prácu navyše, pretože filtrujúce postupnosti si neuchovávajú vlastnú kópiu dát, takže dôjde iba ku kopírovaniu dát z troch súborov do jedného a priebežnej filtrácii.

Funkcia external_breadth_first_traversal() implementuje prechod celým grafom do šírky. Používa jednu postupnosť bit_sequence, v ktorej eviduje ešte nenavštívené vrcholy. Na každý postupne volá external_breadth_first_search s vlastným visitorom. Tento zaznamenáva udalosti navštívenia vrchola do postupnosti a volá visitora, ktorý bol funkcii predaný ako parameter.

3.3.5 external_dfs.hpp

Funkcia external_depth_first_search() implementuje prehľadávanie zadanej komponenty súvislosti do hĺbky podľa 2.2.1. So štruktúrou na ukladanie navštívených vrcholov budeme často robiť operácie vloženia prvku a zistenie prítomnosti. Preto ju reprezentujeme pomocou triedy std::set⟨int⟩, ktorá implementuje tieto operácie v čase $O(\log N)$. Na začiatku každej iterácie hlavného while-cyklu je na vrchu zásobníka stav iterátora pre aktuálny vrchol. Ak iterátor, ktorý zodpovedá tomuto stavu, ukazuje na nejaký vrchol, ktorý sa nenachádza medzi navštívenými vrcholmi, potom tento vrchol navštívime a vložíme na vrchol zásobníka. Tým sa stane aktuálnym vrcholom v nasledujúcej iterácii. Ak naopak aktuálny vrchol nemal žiadneho ďalšieho suseda, potom ho odstránime zo zásobníka.

Funkcia external_depth_first_traversal() implementuje prechod celým grafom do hĺbky podobným spôsobom, ako funkcia external_breadth_first_traversal() v časti 3.3.4.

Kapitola 4

Dokumentácia

4.1 Trieda block_file

Implementuje prístup k diskovým súborom spolu s cachovaním.

Definované konštanty

BLOCK_SIZE
Veľkosť jedného bloku v bytoch.

MEMORY_BLOCKS
Počet blokov, ktoré je možné alokovať v pamäti.

MEMORY_ITEMS
Koľko čísel sa zmestí do týchto blokov. Skratka pre `(`*MEMORY_BLOCKS* `*` *BLOCK_SIZE* `/` **sizeof(int)**`)` .

Členovia

char `*`*buffer*
Vnútorný buffer.

block_file(**char***` `*name*, **char***` `*mode*, **int** *del=2*)
Otvorí súbor s názvom name s právami mode. Hodnota del kontroluje zmazanie súboru pri volaní deštruktora. 0 znamená nezmazať, 1 znamená zmazať, štandardná hodnota 2 znamená zmazať súbor, ak bol vytvorený pri vytvorení

objektu.

void *load_block*(**int** *block*)
Načíta požadovaný blok do vnútorného buffera. Hodnota block je offset od začiatku súbora, načíta sa blok ktorému táto pozícia patrí. Tento spôsob adresovania sa používa pri všetkých ostatných funkciách. Zmení aktuálnu pozíciu vo vnútri súbora na načítaný blok.

void *load_block*(**int** *block*, **int** *count*, **void** **b*)
Načíta count blokov do buffera b, začínajúc od bloku block.

void *write_block*()
Zapíše jeden blok z vnútorného buffera na aktuálnu pozíciu v súbore (uloženú pri poslednom volaní **void** *load_block*(**int**)) .

void *write_block*(**int** *block*, **int** *count*, **void** **b*)
Zapíše count blokov z buffera b, začínajúc od bloku block.

4.2 Trieda external_graph

Implementuje externý graf pomocou zoznamu následníkov pre každý vrchol. Zároveň implementuje metódy na iterovanie cez zoznam následníkov. Spĺňa koncepty EGraphConcept a MultiRemoveEGraphConcept.

Členské funkcie

external_graph(**char*** *file*)
Vytvorí graf, ktorého dáta sú uložené v súbore s názvom file.

template <class *SearchStructure*> **void** *remove_visited*(*SearchStructure* **&***ss*)
Odstráň z grafu hrany, ktoré končia vo vrcholoch, ktoré obsahuje štruktúra ss. Volanie tejto funkcie modifikuje dáta uložené v asociovanom súbore. Po jej volaní ostávajú všetky pozície iterátorov v platnosti (ak bol aktuákny vrchol odstránený, je potrebné najprv zavolať funkciu *get_next*()).

std::*pair*<**int**, **int**> *get_position*()
Vráti aktuálnu a koncovú pozíciu vnútorného iterátora. Túto pozíciu možno

obnoviť pomocou volania *set_position*(**int , int**) .

void *set_position*(**int** *current*, **int** *end*)
Nastaví aktuálnu a koncovú pozíciu vnútorného iterátora na current a end.

int *get_current*()
Vráti vrchol, na ktorý ukazuje aktuálna pozícia iterátora.

bool *has_next*()
Vráti true, ak za aktuálnou pozíciou iterátora existuje ďalší vrchol.

int *get_next*()
Posunie aktuálnu pozíciu na nasledujúci vrchol a vráti ho.

std::*pair*<**int , int**> *init_iterator*(**int** *v*)
Nastaví vnútorný iterátor na začiatok zoznamu nasledovníkov vrcholu v.

Nečlenské funkcie

int *num_vertices*(*external_graph*& *g*)
Vráti počet vrcholov v grafe g.

4.3 Trieda external_stack

Implementuje externý LIFO zásobník na celé čísla s cachovaním.

Členské funkcie

external_stack(**char*** *file* = *„stack.000.tmp“*)
Vytvorí zásobník a priradí mu súbor, do ktorého sa budú dočasne ukladať dáta.

void *push*(**int** *item*)
Vloží na vchol zásobníka číslo item.

int *pop*()
Odstráni číslo z vrchola zásobníka a vráti ho. Táto funkcia nesmie byť volaná

na prázdny zásobník.

int *get_top*()
Vráti číslo z vrchola zásobníka bez jeho odstránenia. Táto funkcia nesmie byť volaná na prázdny zásobník.

bool *empty*()
Zistí, či je zásobník prázdny.

4.4 Koncept ESequenceConcept

Tento koncept spĺňajú všetky triedy, ktoré implementujú postupnosť. Význam jednotlivých operácii je rovnaký, preto ich popis uvádzame na tomto mieste.

Požadované funkcie

void *init_iterator*()
Inicializuje vnútorný iterátor postupnosti na začiatok.

int *get_next*()
Vráti nasledujúci člen postupnosti a posunie iterátor na ďalšiu pozíciu.

bool *has_next*()
Zistí, či postupnosť má ďalší člen.

4.5 Trieda list_sequence

Implementuje postupnosť čísel reprezentovanú pomocou zoznamu. Implementuje funkcie požadované konceptom ESequenceConcept s rovnakým významom.

Členské funkcie

list_sequence(**int** *v*)
Vytvorí postupnosť s jediným číslom v. Do takejto postupnosti nie je možné

pridávať ďalšie čísla.

list_sequence(**char** ∗*file*, **int** *del=2*)
Vytvorí postupnosť, ktorej dáta budú uložené do súbora s názvom file. Hodnota del kontroloje zmazanie súboru pri volaní deštruktora. 0 znamená nezmazať, 1 znamená zmazať, štandardná hodnota 2 znamená zmazať súbor, ak bol vytvorený pri vytvorení objektu.

void *flush*()
Zaistí zapísanie všetkých dát do súboru.

void *add*(**int** *v*)
Pridá na koniec postupnosti číslo v.

Členské funkcie s významom ako pri koncepte ESequenceConcept

void *init_iterator*()
int *get_next*()
bool *has_next*()

4.6 Trieda bit_sequence

Implementuje postupnosť čísel reprezentovanú bitovým poľom. Implementuje funkcie požadované konceptom ESequenceConcept s rovnakým významom.

Členské funkcie

bit_sequence(**char**∗ *file*, **int** *c*, **int** *del=2*)
Vytvorí postupnosť s maximálnym počtom členov c, ktorej dáta budú uložené do súbora s názvom file. Pri vytvorení obsahuje všetky čísla od 0 po $c-1$. Hodnota del kontroloje zmazanie súboru pri volaní deštruktora. 0 znamená nezmazať, 1 znamená zmazať, štandardná hodnota 2 znamená zmazať súbor, ak bol vytvorený pri vytvorení objektu.

void *remove*(**int** *v*)
Odstráni číslo v z postupnosti.

Členské funkcie s významom ako pri koncepte ESequenceConcept

void *init_iterator*()
int *get_next*()
bool *has_next*()

4.7 Trieda unify_sequence

template <class *Sequence*, **class** *Sequence2*>
class *unify_sequence*

Zjednotí dve utriedené postupnosti do utriedenej postupnosti. Nevytvára vlastnú kópiu dát, načítava ich zo zjednocovaných postupností podľa potreby.

Členské funkcie

unify_sequence(*Sequence* ∗*first*, *Sequence2* ∗*second*)
Vytvorí postupnosť, ktorá zjednocuje postupnosti first a second. Typy Sequence a Sequence2 musia spĺňať koncept ESequenceConcept.

Členské funkcie s významom ako pri koncepte ESequenceConcept

void *init_iterator*()
int *get_next*()
bool *has_next*()

4.8 Trieda complement_sequence

template <class *Sequence*, **class** *Sequence2*>
class *complement_sequence*

Vráti tie prvky prvej postupnosti, ktoré sa nenachádzajú v druhej postupnosti. Vstupné postupnosti musia byť utriedené, výsledná postupnosť bude takisto utriedená.

Členské funkcie

complement_sequence(*Sequence* ∗*first*, *Sequence2* ∗*second*)
Vytvorí postupnosť vytvárajúcu rozdiel posupnosti first oproti second. Vstupné postupnosti musia byť utriedené, výsledná postupnosť bude takisto utriedená.

4.9 Trieda complement_sequence

template <**class** *Sequence*>
class *remove_duplicates*

Vráti každý prvok vstupnej postupnosti práve raz. Vstupná postupnosť musí byť utriedená, výsledná postupnosť bude takisto utriedená.

Členské funkcie

remove_duplicates(*Sequence* ∗*ss*)
Vytvorí postupnosť, ktorá vráti prvky postupnosti ss bez duplikátov.

Členské funkcie s významom ako pri koncepte ESequenceConcept

void *init_iterator*()
int *get_next*()
bool *has_next*()

4.10 Funkcia external_mergesort

block_file∗ *external_mergesort*(*block_file* ∗*f*, **int** *count*)
Utriedi count čísel uložených po sebe v súbore f algoritmom mergesort. Vráti súbor obsahujúci výsledok. Tento súbor môže a nemusí byť rovnaký ako vstupný. V druhom prípade dôjde k volaniu deštruktora na vstupný súbor. Pomocou tejto funkcie je možné triediť dáta postupnosti list_sequence, predtým je nutné volať jej funkciu flush:
list_sequence s(„*tmp*“);

```
// naplň postupnosť s
s.flush();
s.f = external_mergesort(s.f, pocet);
```

4.11 Koncept EGraphConcept

Definuje rozhranie pre dátovú štruktúru, ktorá reprezentuje graf zoznamom nasledovníkov pre každý vrchol. Navrhnuté je tak, aby implementujúca trieda mohla ukladať dáta na pevnom disku. Význam požadovaných funkcií je rovnaký ako pri triede external_graph.

Požadované funkcie

void *EGraph*::*init_iterator*(**int** *vertex*)
bool *EGraph*::*has_next*()
int *EGraph*::*get_next*()
int *num_vertices*(*EGraph g*)

4.12 Koncept SearchStructureConcept

Definuje rozhranie pre štruktúru, ktorú je možné testovať na prítomnosť celého čísla.

Požadované funkcie

bool *SearchStructure*::*count*(**int** *n*)
Vráti true, ak sa v štruktúre nachádza číslo n. V opačnom prípade vráti false.

4.13 Koncept MultiRemoveEGraphConcept

Definuje rozhranie pre graf, z ktorého je možné odstrániť hrany končiace vo vrcholoch, ktoré sú uložené v štruktúre, ktorej typ spĺňa koncept SearchStructure. Význam požadovaných funkcií je rovnaký ako pri triede external_graph.

Požadované funkcie

void *MultiRemoveEGraph*::*remove_visited*(*SearchStructure ss*)

4.14 Koncept EBFSVisitor

Tento koncept definuje rozhranie pre visitora, ktorý môže byť predaný funkcii *external_breadth_first_search*()

.

Požadované funkcie

void *visit*(**int** *u*, **int** *v*, **int** *l*)
Volaná pri dosiahnutí vrcholu v so vzdialenosťou l, ktorý patrí do rovnakého komponentu súvislosti ako u. Číslo u je jednoznačný identifikátor komponentu.

4.15 Funkcia external_breadth_first_search

template <**class** *EGraph*, **class** *EBFSVisitor*>
void *external_breadth_first_search*(*EGraph* &*g*, *EBFSVisitor* &*visitor*, **int** *u*)
Implementuje prechod časti neorientovaného grafu g do šírky. Táto časť je definovaná komponentou súvislosti s vrcholom u. Navštívenie jednotlivých vrcholov je signalizované objektu visitor volaním funkcie *visit*() . Typ EGraph musí spĺňať koncept EGraphConcept a typ EBFSVisitor musí spĺňať koncept EBFSVisitorConcept.

4.16 Funkcia external_breadth_first_traversal

template <**class** *EGraph*, **class** *EBFSVisitor*>
void *external_breadth_first_traversal*(*EGraph* &*g*, *EBFSVisitor* &*visitor*)
Implementuje prechod neorientovaným grafom g do šírky. Prechod začne v komponente súvislosti, ktorá obsahuje vrchol s najmenším číslom. Pokračuje v komponente, ktorá obsahuje najmenší doteraz nenavštívený vrchol a končí v momente, keď všetky vrcholy boli navštívené. Navštívenie jednotlivých vrcholov je signalizované objektu visitor volaním funkcie *visit*() . Typ EGraph

musí spĺňať koncept EGraphConcept a typ EBFSVisitor musí spĺňať koncept EBFSVisitorConcept.

4.17 Koncept EDFSVisitor

Tento koncept definuje rozhranie pre visitora, ktorý môže byť predaný funkcii *external_depth_first_search*() .

Požadované funkcie

void *visit*(**int** *v*)
Volaná pri dosiahnutí vrcholu v.

4.18 Funkcia external_depth_first_search

template <**class** *EGraph*, **class** *EDFSVisitor*>
void *external_depth_first_search*(*EGraph* &*g*, *EDFSVisitor* &*visitor*, **int** *u*)
Implementuje prechod časti orientovaného grafu g do hĺbky. Táto časť je definovaná komponentou súvislosti s vrcholom u. Navštívenie jednotlivých vrcholov je signalizované objektu visitor volaním funkcie visit(). Typ EGraph musí spĺňať koncept EGraphConcept a MultiRemoveEGraphConcept a typ EDFSVisitor musí spĺňať koncept EDFSVisitorConcept.

4.19 Funkcia external_depth_first_traversal

template <**class** *EGraph*, **class** *EDFSVisitor*>
void *external_depth_first_traversal*(*EGraph* &*g*, *EDFSVisitor* &*visitor*)
Implementuje prechod orientovaným grafom g do hĺbky. Prechod začne v komponente súvislosti, ktorá obsahuje vrchol s najmenším číslom. Pokračuje v komponente, ktorá obsahuje najmenší doteraz nenavštívený vrchol a končí v momente, keď všetky vrcholy boli navštívené. Navštívenie jednotlivých vrcholov je signalizované objektu visitor volaním funkcie visit(). Typ EGraph musí spĺňať koncept EGraphConcept a MultiRemoveEGraphConcept a typ

EBFSVisitor musí spĺňať koncept EBFSVisitorConcept.

Záver

V rámci tejto práce sme naštudovali a implementovali externé algoritmy DFS, BFS a mergesort. Tieto algoritmy sme otestovali na funkčnosť. Aby malo význam zaoberať sa ich efektívnosťou, bolo by nutné previesť isté optimalizácie kódu, konkrétne zefektívniť cachovanie a skladať viacero požiadaviek na I/O operáciu do jednej. Ďalšie možné rozpracovanie tejto práce spočíva v dodefinovaní nových udalostí do rozhrania pre visitorov, pridaní podpory pre vlastnosti hrán a vrcholov do triedy external_graph a v centralizácii definície pre použité typy, napr. pre typ popisovača vrchola.

Literatúra

[Arg96] Lars Arge. *Efficient External-Memory Data Structures and Applications*. Basic Research in Computer Science, 1996.

[CGG+95] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *SODA '95: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 139–149, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[MM02] Kurt Mehlhorn and Ulrich Meyer. External-memory breadth-first search with sublinear i/o. In *ESA '02: Proceedings of the 10th Annual European Symposium on Algorithms*, pages 723–735, London, UK, 2002. Springer-Verlag.

[MR99] Kameshwar Munagala and Abhiram Ranade. I/o-complexity of graph algorithms. In *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 687–694. Society for Industrial and Applied Mathematics, 1999.

[RW94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, 1994.

[Vit01] Jeffrey Scott Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Comput. Surv.*, 33(2):209–271, 2001.