FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Podpora rozpoznávání intervalových grafů pro knihovnu Boost
Student:	Juraj Bielik
Vedoucí:	RNDr. Ondřej Suchý, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

- Seznamte se pravidly vývoje součástí knihovny Boost.
- Seznamte se s algoritmem pro rozpoznavání intervalových grafů [Michel Habib, Ross McConnell, Christophe Paul, Laurent Viennot: Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing, Theoretical Computer Science Volume 234, Issues 1–2, 6 March 2000, Pages 59-84]
- Seznamte se se současnými možnostmi knihoven Boost Graph Library a Interval Container Library.
- Naimplementujte výše uvedený algoritmus pro rozpoznávání intervalových grafů jako součást knihovny Boost .
- Výstupem algoritmu by měl být odpovídající seznam intervalů.
- Obdobně implementujte algoritmus pro převod seznamu intervalů na intervalový graf.
- Analyzujte možnost využití BFS z knihovny Boost a jeho modifikaci na LexBFS pomocí hooků.
- Pokuste se udělat algoritmus co nejobecnější, s využitím šablon, jak je v knihovnách Boost obvyklé.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 19. února 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalárska práca

Podpora rozpoznávání intervalových grafů pro knihovnu Boost

Juraj Bielik

Katedra softwarového inženýrství
Vedúci práce: RNDr. Ondřej Suchý, Ph.D.

13. mája 2021

Poďakovanie

V prvom rade by som sa chcel poďakovať vedúcemu tejto bakalárskej práce, pánovi RNDr. Ondřejovy Suchému, Ph.D., za to, že bol ústretový a trpezlivý pri riešení problémov, ktoré sa počas tvorby tejto práce vyskytli. Zároveň by som chcel poďakovať svojej rodine za to, že boli chápaví a podporovali ma počas celého procesu písania tejto práce. V neposlednom rade ďakujem svojim priateľom, vďaka ktorým som bol schopný prísť aj na iné myšlienky.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval(a) samostatne a že som uviedol(uviedla) všetky informačné zdroje v súlade s Metodickým pokynom o etickej príprave vysokoškolských záverečných prác.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona, v znení neskorších predpisov, a skutočnosť, že České vysoké učení technické v Praze má právo na uzavrenie licenčnej zmluvy o použití tejto práce ako školského diela podľa § 60 odst. 1 autorského zákona.

V Prahe 13. mája 2021 . . . . . .

Odkaz na túto prácu

Bielik, Juraj. *Podpora rozpoznávání intervalových grafů pro knihovnu Boost.* Bakalárska práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Náplňou tejto bakalárskej práce je rozšírenie externej knižnice Boost Graph Library, pracujúcej s grafovou reprezentáciou dát pre programovací jazyk C++. Je pridaná funkcionalita rozpoznávania intervalových grafov a s tým súvisiacich funkcií. Čitateľ je oboznámený s knižnicami Boost Graph Library, Interval Container Library a algoritmami Zjemenie partície, Lex-BFS, Test chordálnosti grafu, Strom klík, Rozpoznávanie intervalového grafu a Konštrukcia intervalového grafu. Následne je rozoberaná implementácia daných algoritmov a použitých štruktúr, s dôrazom na genericitu rozhraní funkcií poskytnutých používateľovi. Časť práce sa venuje testovaniu vypracovaného riešenia. Na záver sú diskutované dosiahnuté časové zložitosti a možné vylepšenia implementácie. Výstupom práce je použiteľné rozšírenie funkcionality Boost Graph Library, dodržiavajúce jej princípy.

Kľúčové slová intervalový graf, rozpoznanie intervalového grafu, Lex-BFS, test chordálnosti grafu, strom klík, konštrukcia intervalového grafu, knižnice Boost, Boost Graph Library

Abstract

The purpose of this thesis is to extend the Boost Graph Library, which specializes in working with graph representation of data in programming language C++. Functionality for interval graph recognition and related functions is added. The reader will get familiar with libraries Boost Graph Library, Interval Container Library and algorithms Partition refinement, Lex-BFS, Chordality test, Clique Tree, Interval graph recognition and Construction of interval graph. Subsequently, the reader will be acquainted with implementation analysis of these algorithms and the structures, with heavy emphasis on generic functions' interfaces provided to the user. A part of this work is devoted to testing of the created solution. Time complexities and possible enhancements of the implementation are discussed at the end. The result of this thesis is an usable extended functionality for Boost Graph Library which adheres to its standards.

Keywords interval graph, interval graph recognition, Lex-BFS, chordality test, clique tree, interval graph construction, Boost libraries, Boost Graph Library

Obsah

Zoznam obrázkov

Zoznam výpisov kódu

Úvod

Intervalové grafy sú využiteľné v mnohých sférach odborného zamerania a to nielen vo výskume, ale napríklad aj pri problematike plánovania operácií. Knižnica *Boost Graph Library* (*BGL*) pre programovací jazyk C++ slúži primárne na prácu s grafmi a poskytuje pre jej používateľov mnoho generických algoritmov. Do dnešného dňa však nepodporuje rozpoznávanie intervalových grafov, ktorým sa bude táto práca zaoberať.

Výstupom tejto práce bude riešenie rozpoznávania intervalových grafov s lineárnou časovou zložitosťou a využitím generického programovania za pomoci šablón v jazyku C++ ako je v knižnici *BGL* štandardom [1]. Zároveň bude vytvorená funkcia slúžiaca na prevod množiny intervalov na odpovedajúci intervalový graf. Zprístupnených pre používateľa bude päť osobitných funkcií súvisiacich s tématikou rozpoznávania intervalových grafov, ktoré majú využitie aj v samostatnom kontexte, ako napríklad funkcia testujúca chordálnosť grafu. Vďaka tomu bude mať používateľ tejto knižnice možnosť používať ešte širšie spektrum operácií nad grafmi uchovávajúcimi jeho dáta.

Kapitola 1

Cieľ práce

Táto práca sa bude zaoberať rozšírením *BGL* o funkcionalitu rozpoznávania intervalových grafov s časovou zložitosťou O($n + m$), ktorej výsledkom bude množina možných intervalov jednotlivých vrcholov, v prípade že je zadaný graf skutočne intervalovým grafom. Zároveň bude potrebný algoritmus lexografického prehľadávania do šírky (*Lex-BFS*), algoritmus testujúci chordálnosť grafu a algoritmus na tvorbu stromu klík, ktoré po sekvenčom použití v uvedenom poradí vyprodukujú vhodný vstup pre algoritmus na overenie intervalového grafu. Spomínané algoritmy doposiaľ nie sú podporované knižnicou *BGL*, pričom sú zásadné pri riešení rozpoznávania intervalových grafov rozoberanom v tejto práci a navyše majú aj osobitné využitia pri práci s grafmi. Takisto budú sprístupnené pre používateľov tejto knižnice.

Ako jeden z cieľov je taktiež vytvorenie funkcie na prevod zoznamu intervalov na intervalový graf. Táto funkcia bude taktiež sprístupnená používateľovi cez generické rozhranie ako rozšírenie *BGL*.

Dôležitým faktorom pri rozširovaní *BGL* je možnosť opätovného použitia kódu, čo je dosiahnuteľné použitím vhodných šablón jazyka C++ [1].

Pre overenie funčnosti jednotlivých riešení budú použité *Unit testy* ktoré ponúka knižnica *Boost Test Library*.

Užívatelia budú mať dostupnú dokumentáciu opisujúcu prácu s vytvorenými funkciami.

1.1 Štruktúra

Štuktúra tejto práce sa delí na päť celkov.

Kapitola 1 sa venuje cieľom tejto práce.

Kapitola 2 sa bude zaoberať použitými Boost knižnicami. Väčšia pozornosť bude venovaná v prvej časti knižnici Boost Graph Library, ktorú táto práca rozširuje. V druhej časti sa bude nachádzať základný popis knižnice Interval Container Library.

Kapitola 3 bude analyzovať päť vybraných algoritmov, na ktorých bude stáť riešenie tejto práce. Pri každom bude uvedený pseudokód a následne popis myšlienok a princípov na ktorých sú založené.

Kapitola 4 bude predstavovať najväčšiu časť tejto práce. Na jej začiatku bude stanovenie funkčných a nefunkčných požiadaviek na nasledujúcu implementáciu. Následne v nej budú rozoberané použité vlastné štruktúry, vybrané podporné funkcie a spracovanie implementácie jednotlivých algoritmov z kapitoly 3.

Kapitola 5 sa v prvej časti bude venovať knižnici Boost Test Library, ktorá bude použitá na testovanie implementácie. V druhej časti bude opísaný spôsob testovania finálneho riešenia. Na záver bude vyhodnotená miera splnenia požiadaviek z kapitoly 4, popis dokumentácie a návod na inštaláciu.

Kapitola 2

Analýza použitých Boost knižníc

Táto kapitola sa bude zaoberať analýzou Boost knižníc Boost Graph Library a Interval Container Library.

2.1 Boost Graph Library

BGL je knižnica pre programovací jazyk C++ určená na prácu s grafovou reprezentáciou dát a algoritmami s nimi operujúcimi. Je súčasťou vysoko hodnoteného komplexu knižníc *Boost*, zameraných na široké spektrum použití v programovacom jazyku C++ [2].

Značnou charakteristikou tejto knižnice je používanie C++ šablón, vďaka ktorým je umožnené znovupoužitie existujúceho kódu podľa potrieb používateľa. Samotná knižnica poskytuje rôzne typy grafov (a ich vnútorných kontajnerov) a algoritmov nezávislých na sebe.

Štandardom tejto knižnice je aj využitie čisto hlavičkových súborov. To má za následok, že knižnicu nie je potrebné kompilovať. Výnimkou tohto pravidla sú iba syntaktické analyzátory GraphViz a GraphML [1].

2.1.1 Genericita

Genericita v *BGL* je dosiahnutá tromi spôsobmi:

- **Interoperabilita dátovej štruktúry a algoritmov**

  Každý algoritmus ktorý táto knižnica obsahuje je dizajnovaný neutrálnym spôsobom vzhľadom na vstupnú štruktúru grafu. To má za následok že jedna šablónová funkcia dokáže pracovať s viacerými kontajnermi, čo ju robi značne nezávislú.

  Výsledkom tohto prístupu je zníženie veľkosti kódu knižnice z O($n*m$) na O($n+m$), kde n počet možných reprezentácií dát v grafových štruktúrach a m je počet algoritmov zahrnutých v knižnici. I keď to na prvý pohľad

môže vyzerať zanedbateľne, nie je tomu tak keďže tento rozdiel naberá na markantnosti s rastúcim počtom algoritmov a dátových štruktúr. Na ilustráciu tohto princípu je možné použiť jednoduchý príklad, v ktorom používame 3 dátové štruktúry a 25 algoritmov. Finálny výsledok nám ukáže, že výsledný počet potrebných implementácií fukncií je iba 28 namiesto 75 funkcií.

Na prechod dátovej štruktúry sú použité separátne iterátory pre vrcholy, hrany a susedné vrcholy.

Ak používateľ nechce/nemôže použiť vstavané štruktúry *BGL*, je mu poskutnutá možnosť pracovať s funkciami tejto knižnice cez externý adaptér určený pre tento prípad.

- **Rozšírenie za pomoci návrhového vzoru *Visitor***

  Algoritmy a kontajnery sú rozšíriteľné cez návrhový vzor *Visitor*. Dobrým príkladom je *BFS* (funkcia `breadth_first_search()`), kde používateľ môže cez jeden z parametrov upraviť výstup v rôznych kritických bodoch prehľadávania grafu podľa jeho potrieb [3]. Možnosti rozšírenia existujúceho algoritmu *BFS* za účelom tejto práce budú rozobrané v sekcii 3.2.

- **Parametrizácia vrcholov a hrán grafu**

  Aby mohol používateľ plnohodnotne využívať grafovú štruktúru, je potrebná možnosť pridania hodnôt na jednotlivé vrcholy a hrany. V niektorých prípadoch je takto dokonca potrebné priradiť viacero hodnôt nesúcich rozličné informácie. *BGL* to umožňuje cez priradenie parametrizovaných vlastností elementom grafu ku ktorým následne priradí identifikačné tagy. Používateľ potom môže pristupovať ku konkrétnym vlastnostiam na vrchole a hrane cez mapu vlastností, pričom každý súbor rovnakých vlastností má vlastnú mapu.

2.1.2 Štruktúry

Reprezentácia dát v grafovej štruktúre je využívaná v rôznych profesionálnych a akademických odvetviach. Z tejto širokej škály použitia vyplýva aj potreba personalizovanej štruktúry grafu, v ktorej používateľ reprezentuje dáta podľa svojich potrieb. *BGL* poskytuje vlastné reprezentácie grafu s vysokou mierou parametrizácie. Koknrétne ide o dve triedy grafov a jeden zoznam hrán [1]:

- `adjacency_list` – vysoko univerzálna dátová štruktúra s rozsiahlou mierou parametrizácie, ako napríklad orientácia grafu, či úprava vlastností vrcholov a hrán,

- `adjacency_matrix` – hrany sú uložené vo forme matice o veľkosti $|V|^2$, kde $|V|$ značí počet vrcholov grafu,

- `edge_list` – zameranie na prácu s hranami grafu.

Trieda `adjacency_list` je veľmi univerzálna a má široký záber využití, preto je jej v nasledujúcej časti venované viac pozornosti.

Predstavuje vysoko parametrizovateľnú štruktúru cez výber šablónových parametrov:

```
adjacency_list<OutEdgeList, VertexList, Directed,
      VertexProperties, EdgeProperties,
      GraphProperties, EdgeList>
```

kde parametre `OutEdgeList` a `VertexList` umožnujú výber kontajnerov, v ktorých budú hrany vrcholov a samotné vrcholy uložené, čo sa môže odraziť na následnej časovej zložitosti pri rôznych operáciach nad grafom.

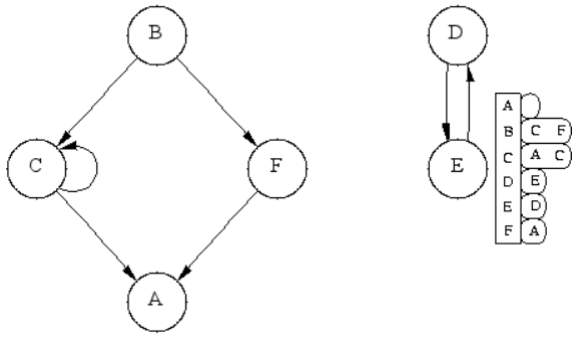
Parameter `Directed` určuje orientáciu grafu. Jednou z možností pri orientovanom grafe je aj spätná referencia na výstupný vrchol hrany.

Parametre `VertexProperties` a `EdgeProperties` slúžia na vnútorné priradenie vlastností vrcholom a hranám.

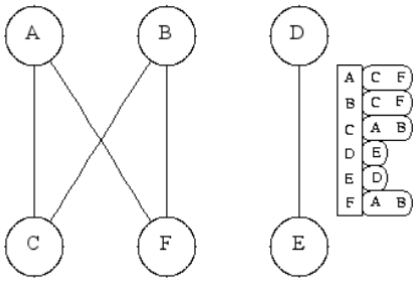
Parameter `GraphProperties` špecifikuje úložný priestor pre objekt grafu.

Parameter `EdgeList` je selektorom kontajnera pre zoznam hrán grafu. Oproti `OutEdgeList` sa táto reprezentácia neviaže ku jednotlivým vrcholom, ale skôr sa ide výčet všetkých hrán ktoré graf obsahuje.

`adjacency_list` je navrhnutý ako dvojdimenzionálna štruktúra kde prvá dimenzia je zoznam vrcholov grafu ktoré majú ďalšiu dimenziu reprezentujúcu zoznam ich hrán (Obr. 2.1 a 2.2) [4].



Obr. 2.1: Reprezentácia orientovaného grafu typu adjacency_list.
(Prevzané zo stránky `https://www.boost.org/doc/libs/1_75_0/libs/graph/doc/adjacency_list.html`, [4])



Obr. 2.2: Reprezentácia neorientovaného grafu typu adjacency_list.
(Prevzané zo stránky `https://www.boost.org/doc/libs/1_75_0/libs/graph/doc/adjacency_list.html`, [4])

2.1.3 Property map

To, čo spája matematickú abstrakciu, ktorou grafy sú a reálne problémy ktoré nimi riešime, sú práve vlastnosti vrcholov a hrán. Tie môže reprezentovať napríklad vzdialenosti, váha, zafarbenie, atď. *BGL* toto umožnuje použitím máp vlastností (*Property Maps*). Ich rozhranie ponúka generické metódy na prácu s vlastnosťami vrcholov a hrán grafu, pričom ku každej mape vlastností sa pristupuje osobitným objektom.

Delíme ich na dva druhy a to na vnútorné a na vonkajšie. Vnútorné mapy vlastnosti sú súčasťou grafu a ich existencia je podmienená životom grafu. Vonkajšie mapy vlastností nie sú súčasťou grafu, ale sú uložené mimo neho. Ich existencia nie je podmienená grafom, čo je využiteľné v prípade, že ich účel je iba krátkodobý, napríklad trvanie algoritmu `breadth_first_search()`. Musia byť algoritmu predané ako argument [5].

2.1.4 Grafové koncepty

BGL nepoužíva iba jeden, ale viacero grafových konceptov, ktorými je možné skúmať a manipulovať s grafom. Účel takto zvoleného prístupu sa skladá z dvoch častí. Prvou sú rozličné požiadavky algoritmov na funkcionalitu štruktúry, s ktorou pracujú. Spravidla ide iba o malú podmnožinu možných funkcionalít. Druhou príčinou je rozdielna schopnosť štruktúr efektívne vykonávať operácie určené na prácu s konkrétnym algoritmom.

Neorientované grafy ponúkajú rovnaké rozhranie ako orientované grafy. I keď nehovoríme o smere hrany, aj pre graf tohto typu môžeme používať funkcie ako `out_edges()` alebo `in_edges()`, no výslednou množinou budú všetky susedné hrany vrcholu, pričom v orientovanom grafe by sa výsledky líšili a ich prienik by tvoril prázdnu množinu. Rozdiel taktiež nastáva v porovnávaní

hrán, konkrétne či sa hrany rovnajú. Pri práci s orientovaným grafom sa nikdy hrany (u, v) a (v, u) nerovnajú, zatiaľ čo v neorientovanom grafe áno [6].

Riešenie rozoberané v tejto práci pracuje s nasledujúcimi grafovými konceptmi:

- **AdjacencyGraph**

  Tento koncept funckiou `adjacent_vertices()` ponúka prístup ku susedným vrcholom grafu v konštantom čase. Nezaoberá sa však orientáciou hrán, ale iba ich susednosťou, čo je postačujúce pre riešenie tejto práce. [7].

- **VertexListGraph**

  Concept *VertexListGraph* poskytuje možnosť iterovať cez zoznam vrcholov grafu `g`. Funkcia `vertices(g)` v konštantnom čase vráti pár iterátorov ukazujúcich na začiatok a koniec zoznamu vrcholov. Návratovou hodnotou funkcie `num_vertices(g)` je počet vrcholov grafu [8].

- **MutableGraph**

  Tento koncept poskytuje možnosť pridávania a odstraňovania vrchlov a hrán aj po jeho vytvorení.

  Časová zložitosť pridania nového vrcholu výrazom `add_edge()` je amortizovane konštantná. Vkladanie novej hrany má časovú zložitosť amortizovane konštantnú alebo v prípade kontroly na prevenciu pridania paralelných hrán môže byť $O(log(E/V))$, kde E je počet hrán a V je počet vrcholov grafu. Táto prevencia sa však vykonáva iba pre grafové typy, ktoré paralelné vrcholy zakazujú [9].

Do dnešného dňa v tejto knižnici neexistuje podpora na rozpoznávanie intervalových grafov [1]. Riešenie spracované v tejto práci teda nezefektívňuje už vytvorenú funkcionalitu, ale pridáva úplne novú k už existujúcemu celku, ktorým je *BGL*, pričom taktiež dáva dôraz na opätovné využitie kódu.

2.2 Interval Container Library

Interval Container Library (*ICL*) je rovnako ako Boost Graph Library súčasťou komplexu knižníc Boost pre programovací jazyk C++. Jej používateľovi poskytuje viacero reprezentácií intervalov (`interval`) a štruktúr na ich uchovanie a prácu s nimi (`interval_set` a `interval_map`).

Rozhranie pre kontajnery `interval_set` a `interval_map` sa skladá z dvoch dôležitých aspektov.

- Oba vystavuju rovnaké funkcie ako set alebo mapa prvkov.

- Keďže sú prvky zoskupené v intervaloch alebo segmentoch je možné cez ne iterovať.

Zároveň ponúkajú rozšírenú funkcionalitu na prácu s intervalmi, ako napríklad preťažené operatóry `+=` a `-=` pre `interval_set`, ktorých výsledkom je zjednotenie alebo rozdiel intervalov uložených v kontajnery `interval_set` [10].

2.2.1 Intervalové štruktúry

ICL poskytuje viacero možností na reprezentáciu intervalov, pričom sú všetky typovo parametrizovateľné. Ide napríklad o `discrete_interval`, `continuous_interval`, a staticky viazané typy intervalov s fixnými ohraničeniami intervalov.

`discrete_interval` je určený na prácu s diskrétnymi dátovými typmi, ako napríklad `int` alebo vhodná reprezentáciu času. Tieto dátove typy majú pevnú minimálnu krokovateľnú jednotku.

`continuous_interval` je určený na prácu so spojitými dátovými typmi, ako napríklad `double` alebo `string`.

Staticky viazané intervalové typy s fixným ohraničením sú `right_open_interval`, `left_open_interval`, `open_interval` a `closed_interval`. Ich ohraničenie je pevne dané a teda nie je predané ako argument pri ich vytvorení.

Pre riešenie rozoberané v tejto práci nie je nutná hlbšia analýza tejto knižnice z dôvodu jej okrajového použitia pri implementácii.

Kapitola 3

Použité algoritmy

Táto kapitola sa zaoberá vybranými funkciami slúžiacimi na rozpoznávanie intervalových grafov (algoritmy 1, 2, 3, 4 a 5) a naivným riešením konštrukcie intervalového grafu zo vstupného zoznamu intervalov (algoritmus 6).

Intervalový graf je taký graf, ktorý umožňuje každému vrcholu priradiť interval tak, že vrcholy sú spojené hranou práve vtedy, keď sa ich intervaly pretínajú.

Z toho vyplýva, že intervalový graf je druhom chordálneho grafu, pri ktorom je možné vytvoriť strom maximálnych klík, ktoré je následne možné usporiadať tak, aby sa každý vrchol vstupného grafu nachádzal v tomto zoradení klík v sekvencii. Toto zoradenie sa nazýva reťaz klík.

Riešenie rozpoznania intervalových grafov je produktom série algoritmov (rozoberaných v následných podsekciách) v špecifickom poradí. Jednotlivé algoritmy budú prístupné samostatne a nie iba ako jedna zreťazená funkcia. Dôvodom voľby daného prístupu je ich možná využiteľnosť aj za osobitným účelom (napríklad rozpoznávanie chordálneho grafu) a teda tento prístup zaručuje možnosť využitia dielčích funkcií podľa potreby namiesto predom vnúteného zreťazenia.

Je dôležité zmieniť, že algoritmy zjemnenie partície (algoritmus 1), Lex-BFS (algoritmus 2), test chordálnosti grafu (algoritmus 3), tvorba stromu klík (algoritmus 4) a rozpoznanie intervalového grafu (algoritmus 5) sú vybrané z článku *Habib a kol.* [11].

V nasledujúcich sekciách je rozoberaná analýza jednotlivých algoritmov. Ich poradie je uvedené v potrebnom zreťazení na priechod celým procesom rozpoznania intervalového grafu od počiatočného grafu. Algoritmus konštrukcie intevalového grafu (algoritmus 6) má opačnú funkciu, pričom neslúži na rozpoznanávanie intervalových grafov, ale dokáže ich vytvoriť na základe zoznamu intervalov.

3.1 Zjemnenie partície

Vstup je zoradená partícia $L = (X_1, ..., X_n)$ setu E a podmnožina $S \subseteq E$.
Výstup je zoradená partícia $L' = (X'_1, ..., X'_m)$.
Postup:

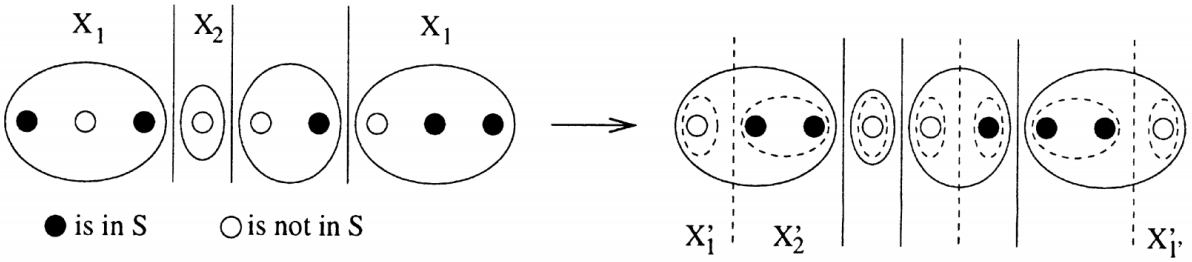
Algoritmus 1: Zjemnenie partície

```
foreach triedu X_a do
    nech Y = S ∩ X_a;
    if Y nie je prázdna a Y ≠ X_a then
        odstráň Y z X_a;
        vlož Y vedľa X_b v L;
    end
end
```

Partícia je zoradená kolekcia disjunktných podmnožín množiny E. Tieto podmnožiny sa nazývajú triedy a ich zjednotenie tvorí kompletnú množinu E. Partícia sa zjemňuje štiepením partičných tried podľa $S \subseteq E$, kde sa každá trieda X_a nahradí za $X_a \cap S$, $X_a \setminus S$.

Po prevedení zjemnenia pre každú triedu X_a platí že:

- $X'_a \subseteq S$, alebo,
- $X'_a \cap S = \emptyset$.



Obr. 3.1: Pred a po zjemnení partície podľa množiny S. (Prevzané z článku *Habib a kol.* [11].)

Zjemnenie partície je všeobecný algoritmus, ktorý zjemní nejakú partíciu množiny prvkov E podľa jej podmnožiny S. Ide teda o všeobecný prístup radenia prvkov ktorý jednotlivé algoritmy implementujú podľa ich špecifických potrieb.

Algoritmy Lex-BFS (algoritmus 2) a rozpoznávanie intervalového grafu (algoritmus 5) sú založené na zjemnení partície [11].

3.2 *Lex-BFS*

Vstup je graf $G = (V, E)$.
Výstup je *Lex-BFS* zoradenie vrcholov $\pi^{-1}(1), ..., \pi^{-1}(n)$.
Postup:

Algoritmus 2: Lex-BFS

```
Nech L je zoradený zoznam vrcholov V delených na partičné triedy X;
i ← n;
while L nie je prázdny do
    nech X_a je posledná trieda;
    pivot ← posledný vrchol v X_a;
    odstráň pivot z triedy a vymaž ju ak je prázdna;
    π(pivot) ← i ;
    i ← i − 1 ;
    foreach triedu X_b do
        nech Y predstavuje susedov pivota v X_b;
        if Y nie je prázdna a Y ≠ X_b then
            odstráň Y z X_b;
            vlož Y priamo za X_b v L;
        end
    end
end
return π
```

Štandardné vyhľadávanie do šírky nešpecifikuje presné poradie pridávania ešte nenavštívených susedných vrcholov aktuálneho vrcholu do fronty z ktorej následne vyberá ďalší vrchol na spracovanie. Lexografické prehľadávanie do šírky (*Lex-BFS*) na druhú stranu rozširuje výber ďalšieho vrcholu na spracovanie o pridanie obmedzení zužujúcich tento výber, čím dosahuje žiadúce vlastnosti.

Výsledné poradie, v ktorom boli vrcholy spracované sa nazýva *Lex-BFS zoradenie*. Je možné dosiahnuť časovú zložitosť $O(n + m)$ za použitia vhodných dátových štruktúr pracujúcich s vrcholmi grafov, pričom sa použije každý vrchol ako pivot maximálne raz.

Dôležitý koncept, ktorý bude použitý taktiež v nasledujúcich podsekciách sú praví susedia vrcholu x v zoradení. Táto množina bude označovaná ako $RN(x)$ a formálne môže byť zapísaná nasledovne:

$$\{y : y \in N(x) \cap \pi(y) > \pi(x)\}$$

kde $N(x)$ je množina susedných vrcholov x a π je vzostupne očíslované zoradenie vrcholov.

Taktiež je potrebné zadefinovať rodiča vrcholu x ($parent(x)$), ktorý reprezentuje najľavejší vrchol $RN(x)$ [11].

Aj keď *BGL* už poskytuje štandarné prehľadávanie do šírky [3] a jeho správanie je do značnej miery upraviteľné cez návrhový vzor *Visitor* [12], nie je vhodné ho použitím tohto návrhového vzoru upraviť natoľko, aby kopíroval spôsob prehľadávania Lex-BFS. Oba algoritmy používajú rozličný prístup ku reprezentácii dát a na to nadväzujúceho výberu ďalšieho vrcholu na spracovanie. Klasické BFS pridáva susedné vrcholy do fronty, z ktorej následne vyberie ďalší prvok na spracovanie [3]. Metóda riešenia Lex-BFS rozoberaná v tejto práci však nepracuje s frontou, no ako pivota si vyberá posledný nenavštívený vrchol z listu všetkých vrcholov. Zároveň pracuje s partičnými triedami, ktoré reprezentujú disjunktné množiny ešte nenavštívených vrchlov. Tieto triedy ďalej štiepy na základe susedných vrcholov aktuálneho pivota [11]. Po analýze algoritmických a štrukturálnych rozdielov týchto dvoch prístupov bolo uznané za praktickejšie naimplementovať samostnatnú funkciu pre Lex-BFS, ako upravovať už existújúcu funkciu BFS, ktorá je súčasťou *BGL*.

3.3 Test chordálnosti grafu

Vstup je graf $G = (V, E)$ a Lex-BFS zoradenie vrcholov π.
Výstup je TRUE ak je π perfektné eliminačné zoradenie, inak FALSE.
Postup:

Algoritmus 3: Chordality test

```
foreach vrchol x ∈ V do
    nech RN(x) sú susedia x napravo v π;
    nech parent(x) je najľavejší vrchol v RN(x);
end
foreach vrchol x ∈ V do
    if (RN(x) \ parent(x)) ⊈ RN(parent(x)) then
        return FALSE;
    end
end
return TRUE;
```

Test chordálnosti grafu neupravuje zoradenie vyprodukované *Lex-BFS* algoritmom, je však podstatným medzikrokom pred vstupom do algoritmu na tvorbu stromu klík (algoritmus 4), ktorý je bližišie rozoberaný v nasledujúcej podsekcii.

Použitie tohto algoritmu je založené na nasledujúcej myšlienke:

> Nech π je perfektné eliminačné zoradenie vrcholov grafu, tak potom $\{x\} \cup RN(x)$ tvoria kliku. Ak tomu pre nejaké x tak nie je, nejde o perfektné eliminačné zoradenie [11].

Na základe tohto je možné určiť, či zoradenie, ktoré *Lex-BFS* vyprodukovalo je skutočne perfektné eliminačné zoradenie, pretože platí, že:

Graf G je chordálny $\iff$ Lex-BFS zoradenie vrcholov je perfektným eliminačným zoradením. [11]

3.4 Tvorba stromu klík

Vstup je chordálny graf $G = (V, E)$ a Lex-BFS zoradenie vrcholov π.
Výstup je strom klík AT grafu G.
Postup:

Algoritmus 4: Strom klík

```
Nech T je strom vrcholov určený parent(x);
nech r je koreňom T;
C(r) ← {r};
foreach pre každý vrchol x stromu T okrem r v preorder poradí do
    if RN(x) ≠ C(parent(x)) then
        vytvor novú kliku C(x);
        C(x) ← {x} ∪ RN(x);
        parent(C(x)) ← C(parent(x))
    else
        C(parent(x)) ← C(parent(x)) ∪ {x};
        C(x) ← C(parent(x));
    end
end
```

Ak je graf G chordálny, je možné preň vytvoriť strom maximálnych klík. To znamená usporiadať maximálne kliky grafu do stromovej štruktúry tak, že kliky obsahujúce daný vrchol indukujú súvislý podgraf tohto stromu.

Platí, že pre každý vrchol x je $RN(x)$ podmožinou jeho predkov v strome T a zároveň, že množina $\{x\} \cup \mathrm{RN}(x)$ tvorí nejakú kliku grafu G. Keďže je strom T prechádzaný v preorder poradí od jeho koreňa, tak pri spracovaní každého vrcholu x už bolo spracované $\mathrm{RN}(x)$ a teda, ak je klika jeho rodiča rovná $\mathrm{RN}(x)$, je rozšírená o vrchol x. Ak tomu tak nie je, je vytvorená nová klika $\{x\} \cup \mathrm{RN}(x)$. Teda pri prechádzaní stromu T platí, že po každom spracovanom vrchole x je dosiahnutá množina maximálnych klík pre podgraf grafu G indukovaný už spracovanými vrcholmi.

3.5 Rozpoznávanie intervalového grafu

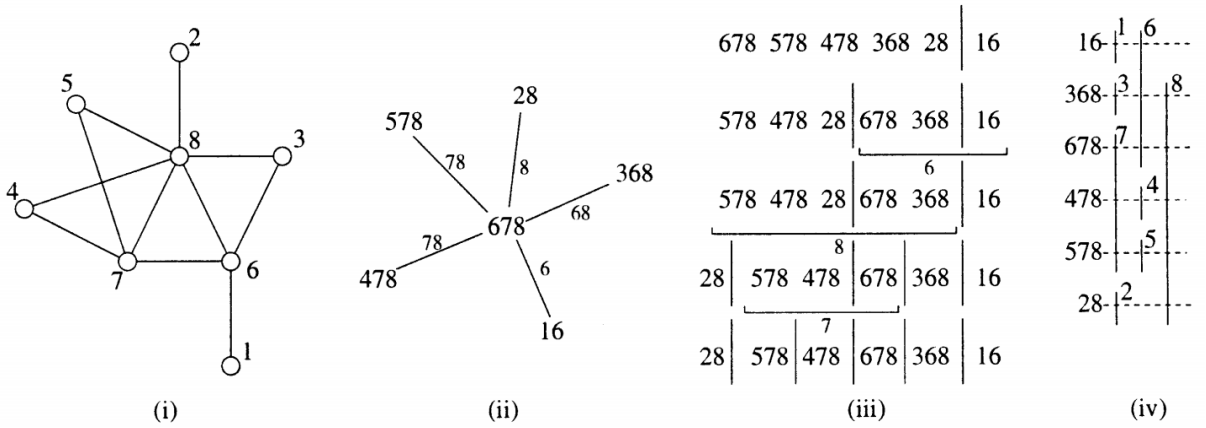
Vstup je graf $G = (V, E)$.
Výstup ak je vstupný graf G intervalový: zoznam možných intervalov pre vrcholy V.
Postup:

Algoritmus 5: Rozpoznávanie intervalového grafu

Over či je G chordálny graf za pomoci Algoritmu 3;
nech T je strom klík, ktorý je výsledkom Algoritmu 4;
nech X je množina maximálnych klík $X = \{C_1, ..., C_l\}$;
nech L je usporiadaný zoznam tried X;
nech q je zásobník *pivotov*;
$q \leftarrow \emptyset$;
while *existuje trieda* X *s veľkosťou* > 1 *v* L **do**
  if $q = \emptyset$ **then**
    nech C_l je posledná nájdená klika v X;
    nahraď X za $X \setminus \{C_l\}, \{C_l\}$ v L;
    $S \leftarrow \{C_l\}$;
  else
    vyber ešte nespracovaný vrchol x z q;
    nech S je množina všetkých maximálnych klík obsahujúcich vrchol x;
    nech X_a je prvá a X_b je posledná trieda obsahujúca kliku z S;
    nahraď X_a za $X_a \setminus S, X_a \cap S$ a X_b za $X_b \cap S, X_b \setminus S$ v L;
  end
  foreach *zostávajúcu hranu* (C_i, C_j), *kde* $C_i \in S$ *a* $C_j \notin S$ **do**
    $q \leftarrow q \cup (C_i \cap C_j)$;
    odstráň (C_i, C_j) z T;
  end
end
foreach *vrchol* x **do**
  vytvor pre x intervaly podľa súvislého výskytu x v L;
  if x *nemá presne jeden interval* **then**
    return *Graf* G *nie je intervalový*
  end
end
return *Graf* G *je intervalový*

Tento algoritmus pracuje so vstupným grafom s predpokladom, že ide o intervalový graf. Na konci je zoradnie klík vstupného grafu otestované, či sa každý vrchol $x \in V$ nachádza v po sebe idúcich klikách bez prerušenia. Ak tento test zlyhá, nejde o intervalový graf. V opačom prípade sú vyprodukované možné intervaly pre jednotlivé vrcholy x tak, aby sa jednalo o intervalový graf.

Na začiatku je klika, ktorá obsahuje posledný nájdený vrchol algoritmom



Obr. 3.2: Priechod algoritmom 5. (i) Vstupný intervalový graf, pričom vrcholy sú očíslované na základe zoradenia vyprodukovaného Lex-BFS algoritmom. (ii) Strom maximálnych klík vytvorený z (i). (iii) Partition refinement množiny klík. (iv) Intervalová reprezentácia získanej reťaze klík. (Prevzané z článku *Habib a kol.* [11].)

Lex-BFS, presunutá na koniec do novej partičnej triedy.

Každý vrchol vstupného grafu je použitý ako pivot maximálne raz, no môže byť použitý až keď ho obsahujú kliky vo viacerých partičných triedach. Kliky, ktoré vrchol obsahujú tvoria indukovaný podgraf stromu klík z algoritmu 4. Vrchol sa stáva vhodným pivotom, až keď nejaká hrana tohto podgrafu pretína dve partičné triedy. Keďže pre každý vrchol je takýto podgraf stromu klík spojitý, tak množina vrcholov, ktorých takéto podgrafy obsahujú hranu (C_i, C_j) je $C_i \cap C_j$. Na to nadväzauje, že keď sa prvýkrát C_i a C_j nachádzajú v rozličných triedach, môžeme vrcholy $C_i \cap C_j$ označiť za vhodné pivoty.

Partičné triedy, ktoré obsahujú pivot sa rozdelia tak, že X_a sa nahradí za $X_a \setminus S, X_a \cap S$ a X_b sa nahradí za $X_b \cap S, X_b \setminus S$, kde S sú kliky obsahujúce pivota, X_a a X_b sú prvá a posledná partičná trieda obsahujúca nejakú kliku s pivotom. Výnimkou je klika obsahujúca prvý pivot, ktorá je odčlenená do samostatnej triedy na koniec zoznamu už na začiatku algoritmu. Delené sú iba triedy X_a a X_b, pretože všetky zvyšné triedy obsahujúce pivot sa už v následnosti nachádzajú v zoradení medzi nimi. Cieľom tohto postupu je postupné zoradenie klík tak, aby sa každý vrchol nachádzal v po sebe nasledujúcich klikách [11].

Obr. 3.2 zachytáva celý proces vytvárania reťaze klík zo vstupného grafu.

3.6 Konštrukcia intervalového grafu

Vstup je zoznam intervalov L.
Výstup je intervalový graf $G = (V, E)$.
Postup:

Algoritmus 6: Konštrukcia intervalového grafu

```
foreach interval x ∈ L do
    vytvor nový vrchol pre x v G;
end
foreach interval x ∈ L do
    foreach interval y ∈ L, kde y ≠ x do
        if x ∩ y ≠ ∅ a hrana (x, y) ∉ E then
            vytvor novú hranu (x, y) v G;
        end
    end
end
```

Ide o naivné riešenie konštrukcie intervalového grafu zo vstupné zoznamu intervalov. Pre každý interval zo vstupného zoznamu je vytvorený odpovedajúci vrchol vo výslednom grafe G. Pokračuje sa naivným prístupom, kde sa pre každý vrchol patriaci nejakému intervalu zo vstupného zoznamu skontrolujú prieniky so všetkými ostatnými intervalmi a v prípade prieniku jednotlivých intervalov sa vytvoria hrany v grafe G medzi vrcholmi odpovedajúcimi intervalom.

Korektnosť algoritmu je triviálne dokázateľná z definície intervalového grafu.

Kapitola 4

Implementácia

V tejto kapitole budeme rozoberať našu implementáciu jednotlivých algoritmov z kapitoly 3 ako súčasť *BGL* a zároveň štruktúry použité v týchto funkciách. Zameriame sa na rozhranie poskytnuté používateľovi, použité štruktúry a prevedenie algoritmov.

4.1 Požiadavky

Ešte pre začatím analýzy implementácie je dôležité uviesť požiadavky ktoré sa budeme snažiť naplniť. Tie sa odvíjajú od algoritmov súvisiacich s rozpoznávaním intervalových grafov a štandardov *BGL* ktorú budeme rozširovať.

Algoritmy rozoberané v kapitole 3, okrem algoritmu 4.10, majú časovú zložitosť $O(n + m)$ [11]. Pre naše riešenie teda bude podstatné optimálne implementovať tieto algoritmy za účelom priblíženia sa tejto časovej zložitosti.

Keďže rozširujeme už existujúcu knižnicu, je podstatné dodržiavať jej štandardy. V našom príde ide primárne o genericitu rozhraní poskytnutých používateľovi. Zámerom je, aby sme používateľa neobmedzovali na nami vybrané štruktúry, ale aby sme boli schopní pracovať so vstupmi pre ktoré sa rozhodne používateľ. Toto pravidlo sa nedá aplikovať stopercentne a teda našim cieľom bude užívateľa obmedziť iba na nutné minimum.

Požiadavky delíme na dve katégorie a to funkčné a nefunkčné.

4.1.1 Funkčné požiadavky

F1 Implementácia algoritmu Lex-BFS.

F2 Implementácia algoritmu test chordálnosti grafu.

F3 Implementácia algoritmu tvorba stromu klík.

F4 Implementácia algoritmu rozpoznávanie intervalového grafu.

F5 Implementácia algoritmu konštrukcia intervalového grafu.

F5 Poskytnutie generického rozhrania používateľovi.

F6 Časové zložitosti funkcií zodpovedajúce časovým zložitostiam implementovaných algoritmov.

4.1.2 Nefunkčné požiadavky

N1 Vytvorenie dokumentácie pre používateľa

N2 Dodržanie konvencií štylizácie kódu knižnice *BGL*

4.2 Obecné vlastnosti implementovaných funkcií

V tejto sekcií budeme hovoriť o spoločných črtách implementovaných funkcií.

V každej funkcii, v ktorej pracujeme s grafom ako vstupom vyžadujeme, aby bol tento graf neorientovaný.

Všetky používateľovi dostupné funkcie a štruktúry sú súčasťou menného priestoru `boost`.

Na začiatku každej funkcie sprístupnenej používateľovi kontrolujeme pomocou `BOOST_CONCEPT_ASSERT` [13] a `BOOST_STATIC_ASSERT` [14], či sú užívateľom predané argumenty správneho typu (podľa jednotlivých požiadaviek rozhraní funkcií). Tento proces sa deje ešte pri kompilácii.

Dosahované časové zložitosti implementácie funkcií `clique_tree()` (sekcia 4.8) a `construct_interval_graph()` (sekcia 4.10) môžu byť ovplyvnené na základe používateľom zvolenej grafovej štruktúry (podsekcia 2.1.4), na čo nemáme dosah. Zároveň v každej funkcii okrem `chordality_test()` pracujeme so štruktúrami, ktorých operácie sú závislé na ich hašovacej funkcii a teda nemôžeme garantovať uvádzané časové zložitosti. Tie sú uvedené za predpokladu konštantých operácií s grafmi a konštantným vyhľadávaním zahašovaných prkov v kontajneroch.

4.3 Použité štruktúry

Aby sme boli schopní dosiahnuť časovú zložitosť $O(n+m)$, na ktorú kladieme dôraz pri všetkých algoritmoch okrem algoritmu konštrukcie intervalového grafu (algoritmus 6), ktorý je implementovaný naivne, bolo súčasťou riešenia implementácie aj vytvorenie vlastných dátových štruktúr, ktoré budeme rozoberať v tejto sekcii.

Využitie jednotlivých členov týchto štruktúr je popísané v nasledujúcich sekciách, kde sú použité. Ide o nasledujúce tri štruktúry.

4.3.1 Element

V kóde: `vertex_element<VertexDescriptor>`

Táto štruktúra obaľuje jednotlivé vrcholy grafu pre efektívnejšiu prácu s grafom. Šablónový parameter `VertexDescriptor` reprezentuje dátový typ vrcholu grafu. Objekty štruktúry Element vždy alokujeme dynamicky kvôli možnosti odkazovania sa na ne cez ukazovatele uložené v iných objektov.

Obsahuje nasledovné členy:

- `vertex_descriptor` – vrchol grafu, ktorý táto štruktúra obaľuje, je predaný ako argument do konštruktora,
- `list_position_it` – iterátor ukazujúci na aktuálnu pozíciu v zozname všetkých Elementov,
- `containing_partition_it` – iterátor ukazujúci na aktuálnu pozíciu `partition_class`, ktorej Element patrí v zozname partičných tried,
- `containing_clique` – ukazovateľ na kliku, do ktorej bol pridaný ako do prvej,
- `neighbours` – zoznam ukazovateľov na susedné vrcholy v grafe, obalené v štruktúre Element,
- `RN` – zoznam ukazovateľov na susedné vrcholy napravo od neho v *Lex-BFS* zoradení, obalené v štruktúre Element,
- `parent` – ukazovateľ na vrchol najviac naľavo v jeho `RN`, obalený v štruktúre Element, rodič v stromovej štruktúre Elementov,
- `tree_children` – zoznam ukazovateľov na jeho potomkov v stromovej štruktúre Elementov.

4.3.2 Clique

V kóde: `clique<VertexDescriptor>`

Štruktúra `clique<VertexDescriptor>` (Clique) je určená na reprezentáciu klík. Objekty štruktúry Clique vždy alokujeme dynamicky kvôli možnosti odkazovania sa na ne cez ukazovatele uložené v iných objektoch.

Obsahuje nasledovné členy:

- `list_position_it` – iterátor ukazujúci na aktuálnu pozíciu v zozname všetkých Clique,
- `containing_partition_it` – iterátor ukazujúci na aktuálnu pozíciu `partition_class`, ktorej Clique patrí v zozname partičných tried,

- `parent` – ukazovateľ na rodiča v stromovej štruktúre klík,
- `children` – zoznam ukazovateľov na jej potomkov v stromovej štruktúre klík,
- `neighbours` – zoznam ukazovateľov na všetky susedné kliky (rodič a potomkovia) v stromovej štruktúre klík,
- `unvisited_neighbours` – nezoradená množina ukazovateľov na všetky susedné kliky (rodič a potomkovia) v stromovej štruktúre klík,
- `clique_elements` – zoznam ukazovateľov na Elementy, ktoré patria tejto klike,
- `clique_vertices` – zoznam všetkých vrcholov grafu, ktoré patria tejto klike,
- `list_position_index` – dátový typ `unsigned int` určujúci číslo v poradí vytvorenia klík,
- `visited` – nepoužívaná `bool` hodnota, vždy `false`, jej existencia slúži iba na implementačnú kompatibilitu s niektorými funkciami.

4.3.3 Partition

V kóde: `partition_class<Component>`

Predstavuje univerzálnu štruktúru pre reprezentáciu partičnych tried. Môže pracovať so štruktúrami Element (podsekcia 4.3.1) a Clique (podsekcia 4.3.2), ktoré pomenúvavame spoločným názvom „komponent“.

Obsahuje nasledovné členy:

- `begin` – iterátor ukazujúci na začiatočnú pozíciu tejto partičnej triedy v zozname komponetov,
- `back` – iterátor ukazujúci na koncovú pozíciu tejto partičnej triedy v zozname komponetov,
- `back` – `bool` hodnota hovoriaca o tom, či trieda bola vytvorená rozdelením zľava (`false`) alebo sprava (`true`), využívané pri algoritme 5,
- `has_pivot` – `bool` hodnota, ktorá informuje o výskyte aktuálneho pivota v nejakej klike ktorú obsahuje,
- `time` – dátový typ `unsigned int`, ktorý hovorí o pomyselnom čase vzniku partície.

V kóde nepoužívame výraz „class“ (trieda), ale „partition“ (partícia), i keď tým myslíme partičné triedy a nie rozličné partície. Dôvodom je vyhnutie sa konfliktom s kľúčovým slovom `class` jazyka C++ pri pomenúvavaní inštancií a definovaní typov tejto štruktúry.

4.4 Podporné funkcie

Podporné funkcie, ktoré budeme rozoberať v tejto sekcii nie sú určené na použitie používateľom, preto sa nebudeme zaoberať ich rozhraním. Z tohto vyplýva, že pre ne nie sú implementované ani *statické asserty*. Sú skryté do menného priestoru `internal`.

Nutne pri nich neuvádzame časové zložitosti, pretože tie sú zahrnuté vo výsledných časových zložitostiach vonkajších funkcií, ktoré ich používajú. Nie sú rozoberané všetky implementované podporné funkcie, no ide skôr o výber tých, ktoré implementujú dielčie časti algoritmov.

4.4.1 Rozdelenie partíčnej triedy

```
template<class Component, class Partition>
void split_partition_to_back(Component *component,
                             std::list<Partition> &partitions,
                             unsigned int current_time)
```

Výpis kódu 4.1: Rozhranie funkcie `split_partition_to_back()`

Táto funkcia presunie komponent z jeho pôvodnej triedy do novovytvorenej hneď za ňou. Presun funguje tak, že vymeníme požadovaný komponent s posledným prvkom jeho partičnej triedy bez toho, aby sme narušili postupnosť iterátorov zoznamu komponentov, čo by mohlo mať za následok narušenie ohraničení jednotlivých partičných tried. Nepoužívame teda metódy `insert()` a `erase()`, ale iba vymieňame hodnoty na daných pozíciách. Aby sme boli schopní túto výmenu urobiť v konštantom čase, využijeme člena `list_position_it`, vďaka ktorému nie je potrebné prehľadávať celý zoznam komponentov, pretože ukazuje na presnú pozíciu presúvaného komponentu. Taktiež nepotrebujeme hľadať, v ktorej triede v zozname `partitions` sa komponent nachádza, pretože každý si drží prístup ku svojej partícii v členovi `containing_partition_it`. Vďaka tomu sme schopní previesť všetky potrebné operácie v konštantom čase.

Dôležitou súčasťou tejto funkcie je taktiež vytvorenie novej triedy, do ktorej presunieme komponent. Novú triedu vytvárame iba v prípade, že trieda napravo od tej pôvodnej, z ktorej komponent odoberáme, nemá rovnakú časovú stopu ako parameter `current_time` a zároveň má člen `from_right` nastavený na hodnotu `false` (teda bola taktiež vytvorená zľava doprava). Ak tomu tak je, vieme, že susedná partičná trieda vznikla už pri predošlom volaní tejto

funkcie pri presune prvého komponentu v rámci tej istej iterácie. V takom prípade iba rozšírime túto novú triedu a pôvodnú zmenšíme doľava. Ak sa však časové stopy nerovnajú, z čoho vyplýva, že susedná partícia bola vytvorená pri nejakej predošlej iterácii s iným pivotom alebo bola vytvorená sprava, tak vytvoríme novú triedu, ktorú vložíme hneď napravo od pôvodnej triedy v zozname `partitions`. Takto vytvorená partícia bude obsahovať iba jediný komponent. Taktiež musíme zmenšiť pôvodnú partíciu o jeden komponent doľava. Keďže triedu vytvárame smerom zľava doprava, nastavíme hodnotu člena `from_right` na `false`.

Posledným krokom je nastavenie iterátora novej triedy (či už novo vytvorenej alebo so zhodnou časovou stopou) členovi `containing_partition_it` spracovávanej komponenty.

Funkcia `split_partition_to_front()` pracuje obdobne, no presúva komponenty a vytvára novú triedu smerom doľava. Členovi `from_right` novej triedy nastavuje hodnotu `true` a pri kontrole, či je potrebné vytvoriť novú triedu alebo má pridať komponent do už existujúcej triedy naľavo, požaduje hodnotu `true` od člena `from_right` na rozdiel od pridávania doprava, kde pre presun komponentu do existujúcej vedľajšej triedy požadujeme `false`.

4.4.2 Zoradenie *RN*

```
template<class VertexDescriptor>
void order_elements_RN(
    const typename std::list<vertex_element<VertexDescriptor> *> elements)
```

Výpis kódu 4.2: Rozhranie funkcie `order_elements_RN()`

Aby sme mohli jednotlivé zoznamy členov `RN` efektívne porovnávať, je potrebné ich predtým zoradiť. O toto zoradenie sa stará podporná funkcia `order_elements_RN`, o ktorej v tejto podsekcii budeme bližšie hovoriť. Jej časová zložitosť je lineárna.

Predtým než začneme radenie *RN* vytvoríme dve samostatné vnútorné štruktúry.

Prvou je `std::vector<Element *> elements_vector`, ktorá bude slúžiť na prístup k elementom v konštantom čase. To využijeme na konci funkcie pri priraďovaní zoradených *RN* jednotlivých vrcholov odpovedajúcim elementom.

Druhou je `std::vector<std::vector<unsigned int>> pairs_vector` [15], ktorá bude reprezentovať dvojice čísiel, ktoré budeme radiť. Zanorený vektor slúži iba na reprezentáciu týchto dvojíc, a teda pri inicializácii ho vytvoríme s fixnou veľkosťou dva. Dôvodom tohto prístupu namiesto použitia `std::pair` [16], je priamočiarejšia implementácia algoritmu CountingSort [17], kde bude stačiť pri jej zavolaní poslať iba určujúci index tejto dvojice ako argument hovoriaci o tom, podľa ktorých hodnôt z dvojíc radíme. Index 0

bude označovať čísla pozícií vrcholov Lex-BFS zoradenia a index 1 bude označovať čísla pozícií ich vrcholov *RN*. Počet dvojíc sa teda bude rovnať súčtu všetkých veľkostí *RN*.

Následne prechádzame všetky elementy v zozname `elements` počas čoho plníme vektory `elements_vector` elementmi v rovnakom poradí, ako sa vyskytujú v `elements` a `pairs_vector` dvojicami uvádzanými v predošlom odseku.

Ďalej algoritmom CountingSort zoradíme celý vektor. Dvojice radíme iba podľa indexu 1, pretože hodnoty v indexe 0 budú slúžiť iba na identifikáciu elementu, ktorému budeme priraďovať do *RN* hodnotu z indexu 1, a preto ich nie je potreba radiť.

Všetkým elementom nastavíme ich členy `RN` na prázdne zoznamy, aby sme im mohli priradiť nové, zoradené *RN*.

Posledným krokom je iterovanie vektorom `pairs_vector`, počas čoho napĺňame odznovu členy `RN` jednotlivých elementov (viď výpis kódu 4.3).

```
for (auto pair : pairs_vector) {
    elements_vector[pair[0]]->RN.push_back(elements_vector[pair[1]]);

    if (elements_vector[pair[0]]->parent == nullptr)
        elements_vector[pair[0]]->parent = elements_vector[pair[1]];
}
```

Výpis kódu 4.3: For cyklus napĺňajúci výsledné `RN`

Výpis kódu 4.3 funguje správne, pretože `elements_vector` zoradením odpovedá zoznamu `elements`. Zároveň, ak element doposiaľ nemá nastavený ukazovateľ na rodiča (člen `parent`), nastavíme mu ho na aktuálny pridávaný element do jeho člena `RN`. Prípad, kedy je rodič ešte nenastavený, vzniká iba pri pridávaní prvého elementu, a keďže elementy pridávame v už zoradenom poradí, vždy to bude prvý element člena `RN`, čo je definíciou jeho rodiča.

4.4.3 Vytvorenie stromovej štruktúry

```
template<class Graph, class IT, class VertexDescriptor>
void clique_tree_aux(const Graph &g,
                IT o_begin,
                IT o_end,
                std::list<clique<VertexDescriptor> *> &cliques,
                std::list<vertex_element<VertexDescriptor> *> &elements)
```

Výpis kódu 4.4: Rozhranie funkcie `clique_tree_aux()`

V tejto podsekcii budeme hovoriť o podpornej funkcii `clique_tree_aux()`. Používame ju na vytvorenie vnútornej stromovej štruktúry klík reprezentovanej členmi `parent` a `children` našej štruktúry Clique (podsekcia 4.3.2).

Jej parametrami sú vstupný graf `g`, iterátory ukazujúce na začiatok a koniec kontajnera vrcholov v Lex-BFS zoradení `o_begin` a `o_end`, zoznam `cliques` obsahujúci kliky typu Clique a zoznam elementov `elements`. Pri opise tejto funkcie budeme používať pojem „klika“ ako objekt Clique.

Je dôležité uviesť, že budeme používať dve odlišné stromové štruktúry. Jednu pre reprezentáciu stromu elementov a druhú pre strom klík.

Na začiatku je if podmienka, ktorá kontroluje, či graf `g` nie je prázdny. V prípade, že je prázdny, nastavíme `cliques` na prázdny zoznam a ukončíme funkciu. V opačnom prípade pokračujeme.

Skonštruujeme elementy zoznamu `elements` v poradí podľa Lex-BFS zoradenia, pridáme im susedov a *RN*, ktoré následne zoradíme (podsekcia 4.4.2). Z elementov funkciou `construct_element_tree()` zostavíme strom, kde rodičom elementu bude element uložený v členovi `parent`.

Teraz sa dostávame ku samotnej konštrukcii stromu klík. Koreňom celého stromu bude posledný element v zozname `elements`. Dynamickou alokáciou preň vytvoríme novú kliku, ktorú mu nastavíme do člena `containing_clique`. Táto klika bude na začiatku obsahovať iba jeden element.

Vo for cykle prechádzajúceho potomkov koreňa stromu elementov v *preorder* poradí voláme rekurzívne funkciu `construct_clique_tree()`, ktorej ako argumenty predáme `cliques`, potomka ako koreň podgrafu stromu a index, ktorý budeme pri vytvorení každej novej kliky inkrementovať o 1. Úlohou tejto funkcie je rekurzívne skonštruovať výsledný strom klík.

Táto funkcia sa skladá z if else podmienky a for cyklu, v ktorom sa budeme rekurzívne zanorovať.

V podmienke sa pýtame na to, či sa *RN* aktuálneho koreňa nerovná vrcholom v klike jeho rodiča. Na porovnávanie používame funkciu `std::equal` zo štandardnej knižnice [18].

Ak je podmienka splnená, vytvoríme novú kliku pre aktuálny koreň, do ktorej pridáme všetky elementy jeho člena `RN` a zodpovedajúce vrcholy. Zároveň tejto novej klike nastavíme člena `parent` na kliku rodiča koreňa a tej pridáme novú kliku na koniec člena `children`. Novú kliku pridáme na koniec zoznamu `cliques` a nastavíme jej člena `list_position_index` na už inkrementovaný index. Tieto dve kliky navzájom pridáme do ich členov `neighbours`.

Ak podmienka splnená nie je, pridáme element do kliky jeho rodiča a index neinkrementujeme.

Nakoniec voláme vo for cykle túto funkciu rekurzívne na všetkých potomkov aktuálneho koreňa.

4.5 Zjemnenie partície

Algoritmus na zjemnenie partície používame v algoritmoch Lex-BFS (algoritmus 2) a rozpoznávanie intervalového grafu (algoritmus 5). Keďže ide o všeobecný prístup ku zoraďovaniu prvkov množiny, je potrebné upraviť ho tak,

aby vyhovoval jednotlivým algoritmom [11]. Z tohto dôvodu nie je implementovaný ako samostatný celok, ale ako súčasť algoritmov, v ktorých je použitý.

4.6 Lex-BFS

Vďaka lexografickému prehľadávaniu do šírky (Lex-BFS) zoradíme vrcholy grafu do perfektného eliminačného zoradenia (ak ide o chordálny graf), ktoré budeme využívať v nasledujúcich algoritmoch. Je teda prvým článkom v reťazi funkcií, ktorej výsledkom bude rozhodnutie, či je vstupný graf intervalový.

Táto funkcia je implementáciou algoritmu 2.

4.6.1 Rozhranie

```
template<class Graph, class Inserter>
void lex_bfs(const Graph &g, Inserter ins)
```

Výpis kódu 4.5: Rozhranie funkcie `lex_bfs()`

Používateľ musí pri volaní funkcie `lex_bfs()` predať dva argumenty. Typ týchto argumentov funkcia prijíma ako šablónové parametre `Graph` pre vstupný graf a `Inserter` pre `insert_iterator`.

Prvým parametrom je graf `g`, ktorý je vstupným grafom. Pre tento graf bude vykonané *Lex-BFS* zoradnie vrcholov. Zároveň požadujeme, aby `g` spĺňal koncepty *VertexListGraphConcept* a *AdjacencyGraphConcept* (podsekcia 2.1.4). Dôvodom je potreba iterovať cez vrcholy grafu a možnosť nájsť susedné vrcholy pre každý vrchol.

Druhým parametrom je *insert_iterator*, čo je iterátor, ktorý slúži na vkladanie hodnôt do kontajnera, pre ktorý bol vytvorený, ktoré v našom prípade budú predstavovať vyprodukované *Lex-BFS* zoradenie. Jeho definujúca kategória musí byť *output_iterator_tag* [19]. V neposlednom rade kontajner, do ktorého tento iterátor vkladá, musí uchovávať hodnoty rovnakého typu ako je dátový typ vrcholov vstupného grafu.

4.6.2 Prevedenie

Používané štruktúry sú Element (podsekcia 4.3.1) a Partition (podsekcia 4.3.3).

Na začiatku skontrolujeme, či nejde o prázdny graf a ak áno, ukončíme funkciu. V opačnom prípade obalíme všetky vrcholy do dynamicky alokovanej štruktúry Element a následne im priradíme susedov na základe hrán grafu. Ukladáme ich do zoznamu `elements` typu `std::list<Element *>` kvôli konštantnej časovej zložitosti pridávania a odoberania jednotlivých prvkov.

Vytvoríme zoznam `partitions` typu `std::list<Partition *>`, v ktorom budú uložené všetky triedy, pričom ich zoradenie je podstatné. Do tohto listu

vložíme počiatočnú partičnú triedu reprezentovanú štruktúrou Partition pokrývajúcu všetky elementy a nastavíme jej iterátor v zozname `partitions` všetkým elementom do člena `containing_partition_it`.

Vytvoríme zoznam vrcholov grafu `ordering`, do ktorého budeme postupne ukladať zoradenie prvkov. Nastavíme časovú stopu `current_time` na hodnotu 1 a vojdeme do cyklu `while(!elements.empty())`.

Ako pivot zvolíme posledný element v poslednej triede z `partitions` (ktorý predstavuje posledný element v `elements`), nastavíme preň `visited` na `true`, aby sme s ním už pri opätovnom navštívení nepracovali a pridáme ho do zoznamu `ordering` na začiatok. Ak by partičná trieda, do ktorej pivot patrí zostala po jeho spracovaní prázdna, tak ju vymažeme z `partitions`, inak koniec tejto triedy posunieme o jeden prvok dopredu. Pivota odstránime z `elements`.

Následne vo for cykle pre každého suseda pivota, okrem tých, ktorí už boli označení za navštívených (`visited == true`), zavoláme pomocnú funkciu `split_partition_to_back()` (podsekcia 4.4.1), kde ako argument parametru `component` pošleme aktuálne spracovávaného suseda pivota vo for cykle. Ďalšími argumentami bude zoznam tried (`partitions`) a časová stopa aktuálnej iterácie while cyklu. Výsledkom bude presun suseda do novej partície napravo.

Po skončení predošlého for cyklu zvýšime časovú stopu o 1 a pokračujeme vo while cykle od vrchu.

Po spracovaní všetkých elementov skončí while cyklus, po ktorom už iba za pomoci vstupného vkladového iterátora skopírujeme `ordering` do používateľovej štruktúry, uvoľníme dynamicky alokovanú pamäť a ukončíme funkciu.

Časová zložitosť je $O(n + m)$, kde n je počet vrcholov vstupného grafu `g` a m je počet hrán.

4.7 Test chordálnosti grafu

Test chordálnosti grafu priamo neprodukuje výstup, ktorý by sme použili pri rozpoznávaní intervalových grafov. Slúži primárne na kontrolu, či je vstupný graf chordálny na základe vyprodukovaného Lex-BFS zoradenia, ktoré musí byť perfektným eliminačným zoradením v prípade chordálneho grafu. Ak tomu tak nie je, tak z definície vyplýva, že vstupný graf nie je intervalový.

Táto funkcia je implementáciou algoritmu 3.

4.7.1 Rozhranie

```
template<class Graph, class Iter>
bool chordality_test(const Graph &g, Iter o_begin, Iter o_end)
```

Výpis kódu 4.6: Rozhranie funkcie `chordality_test()`

Používateľ musí pri volaní funkcie `chordality test()` predať tri argumenty. Typ týchto argumentov funkcia prijíma ako šablónové parametre `Graph` pre vstupný graf a `Iter` pre iterátory štruktúry, v ktorej má používateľ uložené zoradenie vrcholov.

Prvým parametrom je graf `g`, ktorý je vstupným grafom. Požadujeme aby graf zahŕňal koncepty *VertexListGraphConcept* a *AdjacencyGraphConcept* (podsekcia 2.1.4).

Druhý a tretí parameter predstavujú iterátory ukazujúce na začiatok a koniec štruktúry zoradenia vrcholov grafu `g`. Požadujeme, aby boli kategórie *forward_iterator*, čo je kategória iterátorov, ktoré umožnujú prechod štruktúrou smerom dopredu [20]. Taktiež kontrolujeme, či štruktúra, ktorú prechádzame, obsahuje prvky typu zhodného s reprezentáciou vrcholov vstupného grafu.

4.7.2 Prevedenie

Používame štruktúru Element (podsekcia 4.3.1).

Ako prvé obalíme vrcholy grafu do štruktúry Element a následne im pridáme susedné elementy. Ukladať si ich budeme v zozname `elements`.

Ďalej každému elementu naplníme člen `RN`. Keďže budeme neskôr potrebovať tieto členy elementov porovnávať s ohľadom na dobrú časovú zložitosť, je potreba tieto zoznamy najprv zoradiť. Na ich vzostupné zoradenie[1] použijeme podpornú funkciu `order_elements_RN()` (podsekcia 4.4.2).

Následne vo for cykle kontrolujeme, či sú *RN* vrcholov podmnožinami *RN* ich rodičov. Akonáhle máme členy `RN` elementov zoradené, stávajú sa tieto jednotlivé kontroly triviálnymi z hľadiska riešenia v lineárnom čase.

Ak *RN* nejakého vrcholu nie je podmnožinou *RN* jeho rodiča, vstupný graf nie je chordálny, a teda ukončíme funkciu s návratovou hodnotou `false`.

Ak kontrolu splnia všetky vrcholy, vstupný graf je chordálny a vrátime hodnotu `true`.

Časová zložitosť je O(n+m), kde n je počet vrcholov vstupného grafu `g` a m je počet hrán.

[1] Všetky elementy si držia v členovi *ordering_pos* číslo, ktoré reprezentuje pozície ich vrcholov vo vstupnom zoradení, podľa ktorých ich budeme radiť

4.8 Strom klík

Funkcia `clique_tree()` nadväzuje na výstup funkcie `lex_bfs()`. Jej výstupom je strom klík skonštruovaný na základe Lex-BFS zoradenia vrcholov vstupného grafu `g`.

Táto funkcia je implementáciou algoritmu 4.

4.8.1 Rozhranie

```
template<class Graph, class Iter, class CliqueTree, class CliqueMap>
void clique_tree(const Graph &g,
                 Iter o_begin,
                 Iter o_end,
                 CliqueTree &out,
                 CliqueMap clique_map)
```

Výpis kódu 4.7: Rozhranie funkcie `clique_tree()`

Rozhranie tejto funkcie prijíma päť parametrov. Ich typy sú šablónovými parametrami.

Prvým je vstupný graf `g`, z ktorého vyprodukujeme strom klík. Je preň podmienkou, aby spĺňal koncepty *VertexListGraphConcept* a *AdjacencyGraphConcept* (podsekcia 2.1.4).

Druhým a tretím parametrom sú iterátory `o_begin` a `o_end` ukazujúce na začiatok a koniec používateľovej štruktúry, v ktorej má uložené zoradenie vrcholov. Slúžia na prechod zoradenia vrcholov, z ktorého budeme vytvárať *RN* jednotlivých vrcholov. Požadujeme, aby to boli iterátory typu *forward_iterator* [20].

Štvrtým parametrom je výstupný graf `out`. Aby sme mu mohli pridávať nové vrcholy a hrany, je potrebné, aby typ jeho grafu spĺňal koncept *MutableGraphConcept* (podsekcia 2.1.4). Keďže nie je potrebné prechádzať jeho vrcholy a susedné vrcholy, koncepty *VertexListGraphConcept* a *AdjacencyGraphConcept* nie sú potrebné.

Posledným argumentom je mapa vlastností `clique_map` slúžiaca na mapovanie klík na jednotlivé vrcholy grafu `out`. Požadujeme, aby kľúčom boli vrcholy výstupného grafu. Ako typ hodnoty tejto mapy očakávame kontajner, ktorý bude mať implemtovanú funkciu `insert(iterator, value)`[2] určenú na pridávanie nového prvku. Zároveň musí do nej byť možné pridávať hodnoty.

Pre používateľa sme vytvorili aj novú štruktúru na označenie vlastnosti s názvom `clique_t` pre prípad, že by mu na popísanie klík nevyhovovalo žiadne z označení vlastností, ktoré *BGL* momentálne ponúka (viď. výpis kódu 4.8).

[2]Parameter *iterator* predstavuje iterátor ukazujúci na nejaké miesto v danom kontajnery do ktorého ideme pridať hodnotu (*value*)

```
struct clique_t {
    typedef boost::vertex_property_tag kind;
};
```

Výpis kódu 4.8: Štruktúra na označenie vlastností `clique_t`

4.8.2 Prevedenie

Používané štruktúry sú Element (podsekcia 4.3.1) a Clique (podsekcia 4.3.2).

Funkcia `clique_tree()` je zložená z dvoch podporných funkcií. Prvou z nich je `clique_tree_aux()`, ktorá slúži na vytvorenie stromu klík v rámci vnútorných štruktúr funkcie a následne funkcia `construct_output()`, ktorá z týchto vnútorných štruktúr skonštruuje výsledny graf `out`.

Podpornej funkcii `clique_tree_aux()` sa venujeme v podsekcii 4.4.3, a teda ju tu nebudeme znovu rozoberať.

Budeme sa však zaoberať implementáciou funkcie `clique_tree_aux()`, pretože i keď ide o samostatnú funkciu, jej jediné použitie je počas funkcie `clique_tree()`, pre ktorú bola vytvorená.

Na začiatku funkcie `clique_tree()` skontrolujeme, či je graf `g` súvislý. Takisto skontrolujeme či je graf chordálny pomocou funkcie `chordality_test()` (sekcia 4.7). Ak tomu tak nie je, vypíšeme na chybový výstup správu informujúcu užívateľa o nesplnení jednej z týchto podmienok a následne funkciu ukončíme.

Inak pokračujeme vytvorením prázdnych zoznamov `elements` pre elementy (typ Element) a `cliques` pre kliky (typ Clique). Tieto zoznamy predáme funkcii `clique_tree_aux()` spolu s parametrami `g`, `o_begin` a `o_end`. Výstupom bude vytvorená stromová štruktúra klík reprezentovaná vnútornými členmi jednotlivých klík zo zoznamu `cliques`.

Následne podpornou funkciou `construct_output()` zostavíme z tejto vnútornej reprezentácie výstupný graf. Na jej začiatku vytvoríme nový vektor `std::vector<CTVertexDescriptor> node_vector`, kde šablónový parameter `CTVertexDescriptor` predstavuje typ vrcholov výstupného grafu. V grafe `out` následne pre každú kliku zo zoznamu `cliques` vytvoríme nový vrchol. Zároveň si tieto vrcholy budeme odkladať do `node_vector`.

Vo for cykle, ktorým iterujeme zoznam klík odzadu, pridávame do grafu hrany medzi aktuálnou dvoma vrcholmi odpovedajúcimi aktuálnej klike a jej rodičom. Tento cyklus zastavíme pri prvej klike, ktorá rodiča nemá, pretože je koreňom stromu klík. Pri hľadaní vrcholov klík v grafe `out` si pomáhame členom klík `list_position_index` pridaným pri vytváraní tejto štruktúry vo funkcii `clique_tree_aux()`, ktorý odpovedá indexu vo vektore `node_vector`.

V poslednom kroku tejto funkcie, vo vnútri for cyklu iterujúceho cez zoznamu klík napĺňame mapu vlastností. V ňom z každej kliky skopírujeme vrcholy grafu `g`, ktoré táto reprezentuje a pridáme ich do mapy vlastností, kde

ako kľúč použijeme vytvorený vrchol pre túto kliku. Tu sa `construct_output()` skončí a vrátime sa do funkcie `clique_tree()`.

Na jej konci už len uvoľníme dynamicky alokovanú pamäť použitých štruktúr.

Časová zložitosť funkcie `clique_tree()` v prípade, že pridávanie hrán má konštantnú časovú zložitosť, je O(n+m), kde n je počet vrcholov vstupného grafu `g` a m je počet hrán.

4.9 Rozpoznávanie intervalového grafu

Rozpoznávanie intervalových grafov je hlavná téma, ktorou sa zaoberá táto práca a funkcia `interval_graph_recognition()` je tá, ktorá rozhoduje, či ide o intervalový graf. Navyše v prípade, že je graf intervalový, priradí jednotlivým vrcholom tohto grafu možné intervaly. Je teda posledným článkom postupnosti funkcií na rozpoznávanie intervalových grafov.

Táto funkcia je implementáciou algoritmu 5.

4.9.1 Rozhranie

```
template<class Graph, typename IntervalMap>
bool interval_graph_recognition(const Graph &g, IntervalMap interval_map)
```

Výpis kódu 4.9: Rozhranie funkcie `interval_graph_recognition()`

Rozhranie prijíma dva parametre.

Prvým parametrom je vstupný graf `g`. Pre tento graf posudzujeme, či ide o intervalový graf. Požadujeme, aby zahŕňal koncepty *VertexListGraphConcept* a *AdjacencyGraphConcept* (podsekcia 2.1.4). Dôvodom je, že budeme potrebovať prechádzať jeho vrcholy a zároveň budeme pracovať so susednými vrcholmi.

Druhým parametrom je `interval_map`. Ten predstavuje mapu vlastností, do ktorej pridáme navrhované intervaly pre jednotlivé vrcholy. Požadujeme, aby bola mapou vlastností pre vrcholy a aby bolo možné do nej zapisovať. Typ jej hodnoty musí podporovať konštrukciu párom dátových typov `unsigned int` ako argumenty konštruktora. To použijeme pri vytváraní intervalov pre vrcholy.

4.9.2 Prevedenie

Používame tu štruktúry Element (podsekcia 4.3.1), Clique (podsekcia 4.3.2) a Partion (podsekcia 4.3.3).

Do premmennej `N` si uložíme počet vrcholov grafu. Overíme, či sa tento počet nerovná nule a ak áno, ukončíme funkciu s návratovou hodnotou `false`. Pomocou funkcie `is_graph_connected()` overíme, či je graf spojitý a ak nie

je, taktiež vrátime `false` a ukončíme funkciu. Následne si vytvoríme zoznam vrcholov `std::list<VertexDescriptor> o`, ktorý pošleme do funkcie `lex_bfs()` (sekcia 3.2), ktorá ho naplní zoradenými vrcholmi grafu. Potom funkciou `chordality_test()` (sekcia 4.7) skontrolujeme, či je graf `g` chordálny. Ak nie je, ukončíme funkciu s návratovou hodnotou `false`.

Ako ďalšie si vytvoríme dva zoznamy, `std::list<Element *> elements` a `std::list<Clique *> cliques` a zavolaním funkcie `clique_tree_aux()` ich naplníme. Dôležitým je pre nás hlavne strom klík v zozname `cliques`. Premennú pre časovú stopu nastavíme na 0 a vytvoríme zoznam partičných tried `std::list<Partition> partitions`. Do neho pridáme počiatočnú triedu pokrývajúcu všetky kliky. Na ukladanie pivotov budeme používať frontu typu `std::deque<>` [21] do ktorej budeme pridávať na koniec a odoberať spredu. Aby sme vedeli, či už bol vrchol spracovaný ako pivot, vytvoríme kontajner `std::unordered_set<unsigned int> processed_pivots`, vďaka ktorému budem môct priemerne v konštantnom čase vyhľadať konkrétny vrchol [22] a ak ho nájdeme, a teda už bol raz použitý, tak ho znovu nepridáme. Ako šablónový paramater tejto štruktúry používame `unsigned int`, pretože na identifikáciu vrcholov budeme používať prevažne ich poradové číslo v Lex-BFS zoradení. Dôvodom je, že nemáme garantované, že vrcholy grafu `g` budú reprezentované ako čísla, čo bude pre nás kľúčové ako je opisované v nasledujúcom odseku.

Ďalšími dôležitými štruktúrami sú `std::unordered_set<Clique *> S`, ktorá bude reprezentovať kliky obsahujúce aktuálny pivot a taktiež vektor `std::vector<std::vector<Clique *>> clique_map` z ktorej budeme plniť S. `clique_map` predstavuje dvojdimenzionálnu štruktúru, ktorej hodnoty prvej dimenzie sú vektory klík. Indexy prvej dimenzie mapujeme na elementy cez pozície ich vrcholov v Lex-BFS zoradení, vďaka čomu budeme schopní v konštantom čase nájsť všetky kliky obsahujúce element, a teda nebudeme musieť zakaždým prechádzať zoznam všetkých klík, čo by nepriaznivo ovplyvnilo výslednú časovú zložitosť. To isté by bolo dosiahnuť aj za použitia mapy, no našou snahou je eliminovať riziko nepriaznivého hašovania čo by negatívne ovplyvnilo časovú zložitosť tejto funkcie.

Kliku, ktorá obsahuje prvý vrchol v Lex-BFS zoradení presunieme na koniec zoznamu `cliques`. Dôvodom je, že táto klika vždy bude na okraji výslednej reťaze klík kvôli vlastnostiam Lex-BFS zoradenia [11].

Všetkým klikám skopírujeme člen `neighbours` do `unvisited_neighbours` za účelom následného rýchleho vyhľadávania a odstraňovania hrán.

Vchádzame do while cyklu, v ktorom budeme kliky radiť. Jeho vnútro delíme na dva hlavné prípady za pomoci if else podmienky vyhodnocujúcej, či je fronta klík prázdna.

Ak to je pravda, vstúpime dovnútra if bloku. Tento stav môže nastať iba v dvoch prípadoch a to hneď na začiatku pri prvej iterácii alebo keď sme už spracovali všetky pivoty a nie je možné pridať ďalšie bez toho, aby sa opakovali. V druhom prípade cyklus prerušíme. Ak ide o prvý prípad, tak poslednú kliku v zozname `cliques` oddelíme do novej samostatnej partície.

Tá bude obsahovať jedinú kliku. Pôvodnú triedu tejto kliky zmenšíme o jednu pozíciu doľava. Nezoradená množina `S` bude obsahovať iba túto kliku.

V prípade, že fronta nie je prázdna, a teda výraz v podmienke je `false`, vstúpime do else bloku. Prednú hodnotu fronty označíme za aktuálneho pivota a odstránime ju z fronty. Do `S` for cyklom vložíme všetky kliky z vektoru druhej dimenzie v `clique_map`, pričom indexom prvej dimenzie je samotný pivot. Pomocou funkcie `find_fl_occurence()` nájdeme prvú a poslednú triedu obsahujúcu pivot. Jej výstupom budú iterátory ukazujúce na tieto triedy v zozname `partitions`. Následne preiterujeme všetky kliky v `S` a pokiaľ bude trieda aktuálnej kliky prvou triedou obsahujúcou pivot, oddelíme túto kliku do novej triedy napravo volaním funkcie `split_partition_to_back()`. Ak trieda kliky nebude prvou ale poslednou, oddelíme ju naľavo použitím fukncie `split_partition_to_front()`. Obe tieto funkcie sú rozoberané v podsekcii 4.4.1. Ak trieda kliky nebude ani prvou ani poslednou, nič s ňou nerobíme a pokračujeme na ďalšiu kliku v `S`.

Po if else blokoch voláme funkciu `find_new_pivots()`, ktorá pridá do fronty pivotov nové hodnoty. Postupuje tak, že for cyklom prechádza susedov klík z `S` a pokiaľ aktuálny sused nepatrí do `S`[3], tak prienik vrcholov[4], ktoré obsahujú, pridáme na koniec fronty. Následnej inkrementujeme časovú stopu o 1.

Po skončení while cyklu bude zoznam `cliques` zoradený a ak je graf `g` intervalový, tak toto zoradenie bude tvoriť reťaz klík[5]. Aby sme kontrolu vykonali v lineárnom čase, budeme zoznam klík prechádzať iba raz a pre jednotlivé vrcholy si budeme značiť časovú stopu iterácie ako dvojice, kde prvou hodnotou bude prvý výskyt vrcholu v nejakej klike a druhý bude reprezentovať posledný výskyt. Ak nastane situácia, že narazíme na vrchol, ktorý už má tieto hodnoty nastavené a koncová časová stopa jeho dvojice nie je presne o 1 menšia, vieme, že kliky v ktorých sa nachádza, nie sú za sebou v postupnosti, a teda nejde o intervalový graf a po uvoľnení dynamicky alokovaných štruktúr ukončíme funkciu s návratovou hodnotou `false`. Ak táto kontrola nezlyhá ani pre jeden vrchol, tak tieto časové údaje upravíme[6] a použijeme ako intervaly. Ďalej ich vo for cykle priradíme zodpovedajúcim vrcholom do mapy vlastností.

Uvoľníme dynamicky alokované štruktúry `elements` a `cliques` a ukončíme algoritmus s návratovou hodnotou true.

[3] rýchla kontrola v std::unordered_set za predpokladu priaznivého hašovania [22]

[4] ich poradové čísla v Lex-BFS zoradení

[5] Zoradenie klík je také, že pre každý vrchol platí, že sa nachádza v neprerušenej postupnosti klík

[6] prvý vynásobíme čislom 10 a druhý vynásobíme číslom 10 a pripočítame 5, aby sme dostali intervaly vyhovujúce aj otvoreným intervalom

4.10 Konštrukcia intervalového grafu

Funkcia na konštrukciu intervalového grafu operuje opačným smerom ako zreťazenie algoritmov na rozpoznávanie intervalového grafu. Jej účelom je vytvoriť s používateľom zadanej štruktúry obsahujúcej intervaly z knižnice Interval container library (sekcia 2.2) zodpovedajúci intervalový graf.

Táto funkcia je implementáciou algoritmu 6.

4.10.1 Rozhranie

```
template<class Graph, class Iter, class IntervalMap>
void construct_interval_graph(Graph &g,
                              Iter begin,
                              Iter end,
                              IntervalMap interval_map)
```

Výpis kódu 4.10: Rozhranie funkcie `construct_interval_graph()`

Používateľ musí pri volaní funkcie `construct_interval_graph()` predať štyri argumenty. Typ týchto argumentov funkcia prijíma ako šablónové parametre `Graph` pre výstupný graf a `Iter` pre iterátory štruktúry obsahujúcej intervaly, na základe ktorých bude graf vytvorený.

Prvým parametrom je graf `g`, ktorý je výstupným grafom. To znamená, že do tohto grafu pridáme vrcholy a hrany tak, aby spĺňal definíciu intervalového grafu. Požadujeme, aby graf zahŕňal koncept *MutableGraphConcept* (podsekcia 2.1.4).

Druhý a tretí parameter predstavujú iterátory ukazujúce na začiatok a koniec štruktúry intervalov. Požadujeme, aby boli kategórie *forward_iterator* [20]. Taktiež kontrolujeme, či štruktúra, ktorú prechádzame, obsahuje intervaly z *ICL*, aby sme nad nimi mohli robiť operácie ponúkané danou knižnicou.

Posledný paramater je mapa vlastností vrcholov výstupného grafu. Pri nej staticky kontrolujeme, či je mapou vlastností, či sa typ jej kľúčov zhoduje s typom vrcholov výstupného grafu, či je typ jej hodnôt reprezentáciou intervalu z knižnice *ICL* a či je možné do nej zapisovať.

4.10.2 Prevedenie

Implementácia tohto algoritmu je pomerne intuitívna, keďže ide o naivné riešenie algoritmu. Na začiatku vytvoríme vektor `vertices_vector` pre ukladanie všetkých vrcholov, ktoré budeme do grafu pridávať. Následne vo for cykle pre každý interval vytvoríme vrchol v grafe g a tento vrchol taktiež uložíme do `vertices_vector`.

Keď už sme pridali všetky potrebné vrcholy do výstupného grafu, začneme pridávať hrany medzi nimi. To robíme pomocou dvoch for cyklov zanorených

v sebe. Vonkajší cyklus iteruje cez celý vektor `vertices_vector`, zatiaľ čo vnútorný iterauje iba od aktuálnej pozície vonkajšieho cyklu po koniec vektora, čo slúži aspoň na čiastočné zrýchlenie algoritmu.

Aby sme vedeli, ktoré vrcholy majú mať medzi sebou hranu, používame funkciu `intersects()` z *ICL*, ktorá vyhodnotí či sa dva intervaly pretínajú [23]. Ak sa aktuálne intervaly cyklov pretínajú, vytvoríme medzi nimi hranu. Zodpovedajúce vrcholy nájdeme vo vektore vrcholov použitím indexov ktoré odpovedajú aktuálnym iteráciam.

Časová zložitosť je $O(n^2)$, kde n je počet vstupných intervalov.

Kapitola 5

Testovanie

Aby sme boli schopní overiť či sú výsledky funkcií správne, je zapotreby otestovať ich. Na to použijeme *unit testy* z knižnice Boost Test Library (*BTL*).

5.1 Boost Test Library

BTL ponúka komponenty zamerané na organizovanie testov do testovacích prípadov a testovacích sád, ktoré spolu vytvárajú hierarchickú stromovú štruktúru testov. Stromovú štruktúru môže používateľ vytvoriť manuálne alebo automaticky na základe typu zvolenej registrácie. Makrá obsahujúce `AUTO`, napríklad makro `BOOST_AUTO_TEST_CASE`, sú registrované automaticky zatiaľ čo tie, ktoré ho neobsahujú sú registrované manuálne, ako napríklad makro `BOOST_TEST_CASE`. *BTL* sa zároveň stará o vykonanie testov. Testovacie prípady predstavujú jednotlivé *unit testy*. Podľa testovacích funkcií sú delené na nulárne, unárne a šablónové. Nulárne sú založené na nulárnej testovacej funkcii pre jeden vstup. Unárne umožňujú používateľovi použiť testovaciu funkciu pre kolekciu vstupných hodnôt a teda riešia problém, kde je potrebné volať testovacie funkcie v cykle pre jednotlivé hodnoty, ktoré majú byť otestované. Šablónové testovacie prípady poskytujú možnosť vytvorenia série testov založených na zozname požadovaných typov, pričom následne pracujú obdobne k unárnym funkciám.

Testovacie sady reprezentujú podstrom v stromovej štruktúre všetkých testov. Inak povedané združujú testovacie prípady.

BTL ponúka rôzne varianty jej používania, konkrétne statickú knižnicu, dynamickú knižnicu, jednohlavičkovú variantu a externý spúšťač testov [24].

Aby používateľ nemusel vytvárať všetky potrebné štruktúry na vykonanie testovacej funkcie priamo v testovacom prostredí, táto knižnica pracuje s testovacími inventármi. Tie predstavujú triedy/štruktúry, ktoré sú priradené ku testovacím prípadom a sadám. Ich objekty sú vytvorené ešte pred vykonaním testovacieho prípadu a zničené až po jeho ukončení. Na deklaráciu ich

priradenia ku testovacím prípadom a sadám sa používa upravené makro, ako napríklad `BOOST_FIXTURE_TEST_CASE` [25].

Na validáciu výstupov sú vytvorené nástroje `BOOST_WARN`, `BOOST_CHECK` a `BOOST_REQUIRE`. Ako parameter prijímajú predikát s výstupnou booleovskou hodnotou [26].

5.2 Testovanie implementovaných funkcií

Na testovanie implementácie používame dynamickú variantu *BTL* a automatickú registráciu testovacích prípadov a sád. Na validáciu výstupov používame nástroj `BOOST_REQUIRE`. Je vytvorených šesť testovacích sád, pričom päť z nich obsahuje testy testujúce jednotlivé funkcie sprístupnené používateľovi a šiesta slúži na testovanie vybraných podporných funkcií.

Grafové a intervalové štruktúry, na ktorých testujeme implementáciu, používame ako testovacie inventáre v jednotlivých testovacích prípadoch, pričom jednu štruktúru používame aj vo viacerých testoch.

Pri testovaní funkcií sprístupnených používateľovi sa zameriavame nielen na správnosť výstupov, ale aj na genericitu poskytovaných rozhraní.

Na overenie správnosti výstupov sa pozeráme z dvoch hľadísk. Prvým je kompletné porovnanie výsledku s očakávaným výstupom vo väčšine testov. Napríklad pri funkcii rozpoznávania intervalových grafov (sekcia 4.9) nekontrolujeme iba či je návratová hodnota správna, ale ak očakávame návratovú hodnutu *true*, tak následne porovnávame i jej výstupnú mapu vlastností s očakávanými hodnotami. Druhým prístupom je testovanie nevyhovujúcich alebo okrajových možných vstupov. Medzi nevyhovujúce radíme napríklad nechordálny graf ako vstup do funkcie tvorby stromu klík (sekcia 4.8). Ako okrajovým prípadom myslíme napríklad kompletný alebo prázdny graf.

Medzi testy rozpoznávania intervalových grafov patrí taktiež test generujúci pseudonáhodné intervaly, z ktorých následne skonštruujeme grafovú reprezentáciu pomocou funkcie `construct_interval_graph()`. Výsledný graf je predaný funkcii `interval_graph_recognition()`, od ktorej očakávame návratovú hodnotu `true`. Testovanie sme robili pre maximálne desať vrcholov, aby bolo možné v prípade zlyhania odkrokovať jeho príčinu. Väčšie grafy by boli príliš komplikované na následné manuálne testovanie. Tento test sme vykonali pre viac ako 400 000 pseudonáhodných vstupov.

Aby sme otestovali genericitu rozhraní, ako vstupy používame rôzne dátové štruktúry, s ktorými by si mali funkcie byť schopné poradiť. Napríklad pri funkcii konštrukcie intervalového grafu používame diskrétne alebo spojité invervaly uložené vo vektore alebo zozname, pričom dátové typy ich hodnôt sú `unsigned int` alebo `double`.

Statické asserty v jednotlivých funkciách nemajú automatizované testy, ale pri ich pridávaní do funkcií boli testované manuálne.

Testovanie podporných funkcií je zamerané prevažne na kompletnú konštrukciu objektov štruktúry Element (podsekcia 4.3.1). Ďalej bola testovaná napríklad funkcia slúžiaca na zoradenie člena `RN` vo vnútri elementov, ktorá zahŕňa implementáciu algoritmu CountingSort.

5.2.1 Vyhodnotenie implementačných požiadaviek

V tejto podsekcií budeme hovoriť o naplnení požiadaviek zo sekcie 5.2.1 a dokumentácií funkcií ponúkaných používateľovi.

F1 Algoritmus Lex-BFS bol úspešne implementovaný.

F2 Algoritmus test chordálnosti grafu bol úspešne implementovaný.

F3 Algoritmus tvorba stromu klík bol úspešne implementovaný.

F4 Algoritmus test chordálnosti grafu bol úspešne implementovaný, doposiaľ však nie sú podporované nespojité grafy.

F5 Algoritmus konštrukcia intervalového grafu bol úspešne implementovaný, doposiaľ však nie sú podporované nespojité grafy.

F5 Všetky ponúkané funkcie prijímajú šablónové parametre a funkcie sú implementované tak, aby od používateľa vyžadovali iba nutné minimum .

F6 Funkcie dosahujú požadovaných časových zložitostí v prípade, že hašovanie dát uložených v kontajneroch nebude nepriaznivé. Toto riziko je však nízke pretože sú používané hašovacie funkcie kontajnerov zo štandarnej C++ knižnice, ktoré dosahujú v priemere konštantnú časovú zložitosť.

N1 Pre používateľa je vytvorená dokumentácia generovaná nástrojom Doxygen

N2 Štylizácia funkcií kopíruje štandardy *BGL*

5.2.2 Dokumentácia a inštalácia

Dokumentácia bola vytvorená iba pre funkcie poskytované používateľovi. Podporné funkcie nie sú dokumentované, pretože nie sú určené na priame použitie a ich rozhranie je schované v mennom prostredí `internal`.

Na jej tvorbu sme použili nástroj Doxygen. Zvolený bol preto že ide de facto o štandard tvorby dokumentácie pre programovací jazyk C++. Podporované sú však aj jazyky ako C, PHP, Java, Python atď. Ako náš formát dokumentácie sme zvolili HTML, no Doxygen dokáže vygenerovať dokumentáciu aj vo formátoch ako napríklad RTF alebo hyperlinkované PDF [27].

Obr. 5.1 je ukážkou dokumentácie funkcie `chordality_test()`. Obsahuje stručný popis náplne funkcie, vstupných a výstupných parametrov a požiadaviek na typ štruktúr predaných používateľom.

Functions

template<class Graph , class Iter >
bool **boost::chordality_test** (const Graph &g, Iter o_begin, Iter o_end)

Detailed Description

Determines whether the input graph is chordal based on the provided Lex-BFS ordering of vertices.

Template Parameters

Graph type of input graph structure, must provide Vertex List Graph Concept and Adjacency Graph Concept
Iter forward iterator of a container containing Graph vertex descriptors

Parameters

[in] **g** input graph
[in] **o_begin** iterator pointing to the beginning of vertex ordering container
[in] **o_end** iterator pointing to the end of vertex ordering container

Returns

`true` if **g** is a chordal graph, otherwise `false`

Obr. 5.1:

5.2.3 Inštalácia

Pri vývoji bol použitý operačný systém Ubuntu 18.04.2 LTS. Používaný kompilátor bol GNU g++. Implementácia pracuje s knižnicami Boost pre progamovací jazyk C++. Použitá verzia bola 1.65.1.0ubuntu1.

Nie je garantovaná kompatibilita s inými operačnými systémami a staršími verziami GNU g++ a Boost knižníc.

Na spustenie testov je možné použiť priložený makefile. Obsahuje nasledujúce príkazy:

all skompiluje kód a spustí výsledný spustiteľný súbor

compile iba skompiluje kód

run má takú istú funkcionalitu ako `all`

clean vymaže súbory *.o

Záver

Náplňou tejto bakalárskej práce bolo rozšíriť Boost Graph Library pre jazyk C++ o doposiaľ nepodporovanú funkcionalitu rozpoznávania intervalových grafov a k tomu nadväzujúcich funkcií, ktoré taktiež *BGL* doposiaľ nepodporuje, konkrétne Lex-BFS, test chordálnosti grafu, tvorbu stromu klík a prevod zoznamu intervalov späť na intervalový graf.

Pri implementácií bol kladený dôraz na genericitu a časovú zložitosť. Implementácia bola navrhnutá tak, aby od používateľa požadovala iba nutné minimum a inak mu nechávala voľnú ruku vo výbere štruktúr, s ktorými pracuje. Zároveň pri implementácii boli zvolené také štruktúry a postupy, aby sa časová zložitosť priblížila O(n+m). To sa ukázalo byť v kolízii s genericitou rozhrania, pretože nutné operácie s grafmi majú rozličné časové zložitosti odvíjajúce sa od používateľom zvolenej reprezentácie (sekcia 4.2). Podrobnejší rozbor miery splnenia požiadaviek sa nachádza v kapitole 5.2.1.

Výsledné riešenie tejto práce stále môže byť vylepšené:

- Používanie kontajnerov ktoré vyhľadávajú dáta na základe hašovacej funkcie so sebou nesie riziko nepriaznivého zahašovania prvkov, čo môže ovplyvniť výslednú časovú zložitosť. V rámci optimalizácie vytvorených funkcií by bolo vhodné zahrnúť stabilnejší prístup ku ukladaniu dát.

- Odstránenie závislosti funkcie na tvorbu stromu klík na implementáciu vkladacej funkcie *insert()*.

- Rozšírenie funkcií *strom klík* a *rozpoznávanie intervalového grafu* o možnosť spracovania nespojitých grafov.

Plánom pre ďalší vývoj tejto práce je po odstránení nedostatkov ponúknuť spracované rozšírenia vývojárom spravujúcim *BGL* ako oficiálnu súčasť súboru knižníc *Boost*, vďaka čomu by sa dosah tejto práce prudko zvýšil.

Celá práca vrátane dokumentácie je dostupná na školskom Gitlabe.
`https://gitlab.fit.cvut.cz/bielijur/interval_graph_recognition`

Literatúra

[1] Siek, J.; Lie-Quan, L.; Lumsdaine, A.: The Boost Graph Library (BGL). 2001, [online] [cit. 2021-04-21]. Dostupné z: https://www.boost.org/doc/libs/1_75_0/libs/graph/doc/index.html

[2] Rivera, R.; Dawes, B.; David, A.: 2007, [online] [cit. 2021-04-21]. Dostupné z: https://www.boost.org/

[3] Siek, J.: breadth_first_search. 2001, [online] [cit. 2021-04-21]. Dostupné z: https://www.boost.org/doc/libs/1_54_0/libs/graph/doc/breadth_first_search.html

[4] Siek, J.; Lie-Quan, L.; Lumsdaine, A.: BFS Visitor Concept. 2001, [online] [cit. 2021-04-22]. Dostupné z: https://www.boost.org/doc/libs/1_75_0/libs/graph/doc/adjacency_list.html

[5] Siek, J.: Property Maps. 2001, [online] [cit. 2021-04-22]. Dostupné z: https://www.boost.org/doc/libs/1_55_0/libs/graph/doc/using_property_maps.html

[6] Siek, J.: Graph Concepts. 2001, [online] [cit. 2021-04-22]. Dostupné z: https://www.boost.org/doc/libs/1_57_0/libs/graph/doc/graph_concepts.html

[7] Siek, J.: AdjacencyGraph. 2001, [online] [cit. 2021-04-22]. Dostupné z: https://www.boost.org/doc/libs/1_57_0/libs/graph/doc/AdjacencyGraph.html

[8] Siek, J.: VertexListGraph. 2001, [online] [cit. 2021-04-22]. Dostupné z: https://www.boost.org/doc/libs/1_57_0/libs/graph/doc/VertexListGraph.html

[9] Siek, J.: MutableGraph. 2001, [online] [cit. 2021-04-22]. Dostupné z: https://www.boost.org/doc/libs/1_74_0/libs/graph/doc/MutableGraph.html

[10] Faulhaber, J.: Chapter 1. Boost.Icl. 2010, [online] [cit. 2021-05-06]. Dostupné z: https://www.boost.org/doc/libs/1_64_0/libs/icl/doc/html/index.html

[11] Habib, M.; McConnell, R.; Paul, C.; aj.: Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing. *Theoretical Computer Science*, ročník 234, č. 1, 2000: s. 59–84, ISSN 0304-3975, doi:https://doi.org/10.1016/S0304-3975(97)00241-7. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0304397597002417

[12] Siek, J.; Lie-Quan, L.; Lumsdaine, A.: BFS Visitor Concept. 2001, [online] [cit. 2021-04-23]. Dostupné z: https://www.boost.org/doc/libs/1_54_0/libs/graph/doc/BFSVisitor.html

[13] Siek, J.; Lumsdaine, A.; Abrahams, D.: BoostConceptCheckingReference-1.66.0. 2007, [online] [cit. 2021-04-30]. Dostupné z: https://www.boost.org/doc/libs/1_66_0/libs/concept_check/reference.htm

[14] Maddock, J.; Cleary, S.: Chapter 29. Boost.StaticAssert. 2005, [online] [cit. 2021-04-30]. Dostupné z: https://www.boost.org/doc/libs/1_60_0/doc/html/boost_staticassert.html

[15] std::vector. 2021, [online] [cit. 2021-05-02]. Dostupné z: https://en.cppreference.com/w/cpp/container/vector

[16] std::pair. 2020, [online] [cit. 2021-05-02]. Dostupné z: https://en.cppreference.com/w/cpp/utility/pair

[17] CountingSort. 2021, [online] [cit. 2021-05-06]. Dostupné z: https://www.geeksforgeeks.org/counting-sort/

[18] std::equal. 2021, [online] [cit. 2021-05-02]. Dostupné z: https://en.cppreference.com/w/cpp/algorithm/equal

[19] std::insert_iterator. 2019, [online] [cit. 2021-04-29]. Dostupné z: https://en.cppreference.com/w/cpp/iterator/insert_iterator

[20] std::forward_iterator. 2021, [online] [cit. 2021-04-29]. Dostupné z: https://en.cppreference.com/w/cpp/iterator/forward_iterator

[21] std::deque. 2021, [online] [cit. 2021-05-02]. Dostupné z: https://en.cppreference.com/w/cpp/container/deque

[22] std::unordered_set. 2021, [online] [cit. 2021-05-02]. Dostupné z: https://en.cppreference.com/w/cpp/container/unordered_set

[23] Faulhaber, J.: Intersect. 2010, [online] [cit. 2021-05-03]. Dostupné z: https://www.boost.org/doc/libs/1_58_0/libs/icl/doc/html/boost_icl/function_reference/intersection.html

[24] Rozental, G.: Boost Test Library. 2007, [online] [cit. 2021-05-07]. Dostupné z: https://www.boost.org/doc/libs/1_45_0/libs/test/doc/html/index.html

[25] Test case fixtures. 2016, [online] [cit. 2021-05-07]. Dostupné z: https://www.boost.org/doc/libs/1_63_0/libs/test/doc/html/boost_test/tests_organization/fixtures/case.html

[26] BOOST_<level>. 2017, [online] [cit. 2021-05-07]. Dostupné z: https://www.boost.org/doc/libs/1_65_0/libs/test/doc/html/boost_test/utf_reference/testing_tool_ref/assertion_boost_level.html

[27] Heesch, D.: Doxygen. 2021, [online] [cit. 2021-05-10]. Dostupné z: https://www.doxygen.nl/index.html

Dodatok **A**

Zoznam použitých skratiek

BGL Boost Graph Library

ICL Interval Container Library

BTL Boost Test Library

Lex-BFS Lexografické prehľadávanie do šírky

HTML HyperText Markup Language

RTF Rich Text Format

PDF Portable Document Format

GNU GNU's Not Unix

LTS Long-term support

Dodatok **B**

Obsah priloženej SD karty

- `README.txt` ..... stručný popis obsahu SD karty
- `project` ..... priečinok obsahujúci implementáciu
  - `boost_src` ..... zdrojové kódy rozšírenia knižnice
  - `boost_tests` ..... zdrojové kódy testových súborov
  - `makefile` ..... makefile na spustenie testov
- `doc` ..... priečinom s dokumentáciou
  - `html` ..... dokumentácia vo formáte HTML
- `text` ..... text práce
  - `thesis.pdf` ..... text práce vo formáte PDF
- `thesis` ..... zdrojová forma práce vo formáte LaTeX