ХИМИКОТЕХНОЛОГИЧЕН И МЕТАЛУРГИЧЕН УНИВЕРСИТЕТ

ДЕПАРТАМЕНТ ПО ФИЗИКО-МАТЕМАТИЧНИ И ТЕХНИЧЕСКИ НАУКИ

КАТЕДРА „ИНФОРМАТИКА”

маг. инж. Стефан Милчев Панов

АЛГОРИТЪМ ЗА РАЗПОЗНАВАНЕ НА ИНТЕРВАЛНИ ГРАФИ С ЕДНОВРЕМЕННО ПОСТРОЯВАНЕ НА ИНТЕРВАЛЕН МОДЕЛ

А В Т О Р Е Ф Е Р А Т

на дисертация

за придобиване на образователната и научна степен „доктор”
по научна специалност 4.6. Информатика и компютърни науки
(Информатика)

Научен ръководител: доц. д-р инж. Атанас Велков Атанасов

Научно жури:
1. проф. д-р Идилия Бачкова – председател, рецензент
2. доц. д-р Веска Ганчева - рецензент
3. доц. д-р Атанас Атанасов
4. доц. д-р Тодор Димов
5. доц. д-р Александър Ефремов

София, 2015

Дисертационният труд е написан на 116 страници, съдържа 34 фигури и 5 таблици.
Цитирани са 126 източника.

Представеният дисертационен труд е обсъден и приет за защита на заседание на разширен катедрен съвет на научното звено на катедра „Информатика”, състояло се на 10.06.2015 г.

Публичната защита на дисертационния труд ще се проведе на 29.09.2015 от 13:00 часа в зала 431, сграда „А” на ХТМУ.
Материалите са на разположение на интересуващите се на интернет страницата на ХТМУ и в отдел „Научни дейности”, стая 406, етаж 4, сграда „А” на ХТМУ.

Въведение

Един ненасочен граф е **Интервален Граф** (ИГ), ако неговите върхове могат да бъдат поставени в съответствие един към един с множество от интервали върху една линия така, че два върха в графа са съседни (т.е. има ръб между тях), ако и само ако техните съответни интервали се припокриват. Множеството от интервали се нарича **интервален модел** (ИМ) или **интервално представяне** (ИП) на графа. За първи път интервалните графи са въведени от Hajos през 1957 и разгледани в приложното изследване на S. Benzer през 1959 г. Те се появяват по естествен начин в процеса на моделиране на ситуации от реалния живот, особено когато са включени времеви зависимости или други ограничения, които са линейни по природа.

Самите интервални графи представляват специален клас на добре известните **хордни графи** – един граф е хорден, ако в него не се съдържа цикъл без хорди, състоящ се от 4 или повече върха. Съществуват десетки статии описващи приложението на интервални графи в най-разнообразни области – биология, археология, генетика, психология, управление на трафика, планиране на задачите и други.

ИГ са особено харесван клас графи заради факта, че различни алгоритмични техники продуцират ефикасни алгоритми върху тях. Едно друго приложение на интервалните графи е при работа с особено големи масиви от данни, какъвто е примерът с човешкия геном. Използването на ИГ формализмът позволява да се открият несъответствия в основните данни и както е подчертано в литературата нито една друга техника не е способна на това.

Разнообразни са опитите да се докаже, че един граф е интервален, или обратното, че не е такъв. От далечната 1965 г., когато е предложен първият - нелинеен и като следствие, особено бавен - алгоритъм за разпознаване на ИГ досега се представени различни парадигми за решение на проблема, някои от които в няколко модификации. В едни от тях се работи с доста сложни структури от данни. При тях практическата

реализацията на конкретни алгоритми е силно затруднена. Други са свързани с изключително тежки теоретични доказателства, с широко застъпена графична теория, в които в последствие нерядко се откриват грешки. Което от своя страна показва, че няма гаранция решението (доказателството) да е универсално.

В последните двадесетина години преобладават вариации на алгоритми, които работят на две основни стъпки. Първата от тях е да се докаже, че началният граф е хорден. Заслужава да се спомене, че наличието на предварителна стъпка влияе на бързодействието. *Lexicographic Breadth First Search* (LBFS) е парадигмата за ефикасно разпознаване на хордни графи. Крайният резултат от работата на LBFS е е едно специално подреждане, при което за всеки връх е в сила, че съседите му, намиращи се след него в подреждането и самият връх образуват клика.

Допълнително, заедно с ускореното навлизане на паралелното програмиране, все повече възниква нужда от алгоритми, които са подходящи за имплементация върху паралелни системи. Може да се заяви, че съществуващите класически алгоритми се оказват неподходящи за паралелизация. Това с особена сила важи за алгоритмите, базирани на LBFS.

Един разпознаващ алгоритъм е особено ценен, ако принадлежи към класа на т. нар. “Сертифициращи алгоритми” – такъв алгоритъм снабдява със сертификат всеки отговор, който произвежда. Сертификатът е свидетелство, на което може да се разчита и което доказва, че отговорът не е бил повлиян от грешка в имплементацията.

Глава 1. Литературен обзор

За да може да се навлезе достатъчно дълбоко в проблема на интервалните графи и разнообразните подходи за решение на отделните подзадачи е необходимо да се дадат някои основни понятия от теория на

графите, а също и да се дискутират различни характеристики на интервалните графи. В настоящата глава, с малки изключения няма да се дискутират доказателствата на теоремите.

1.1. Терминология и дефиниции

Броят на върховете в един граф обикновено се означава с *n*, а броят на ръбовете (също се използват, включително и в представената работа вместо ръбове и терминте дъги и ребра) с *m*. Ако съществува ръб между два върха *v1* и *v2* ще казваме, че *v1* е *съсед* на *v2* и обратно – *v2* е съсед на *v1*. За дефинирането на класическите интегралните графи като такива допълнителните ограничения са:

Графът да е ненасочен;

Два върха не могат да бъдат свързани с повече от един ръб;

Липсват цикли. Цикъл е дъга, свързваща даден връх със самия него;

Началният граф, за който ще се прави доказателството няма изолирани върхове;

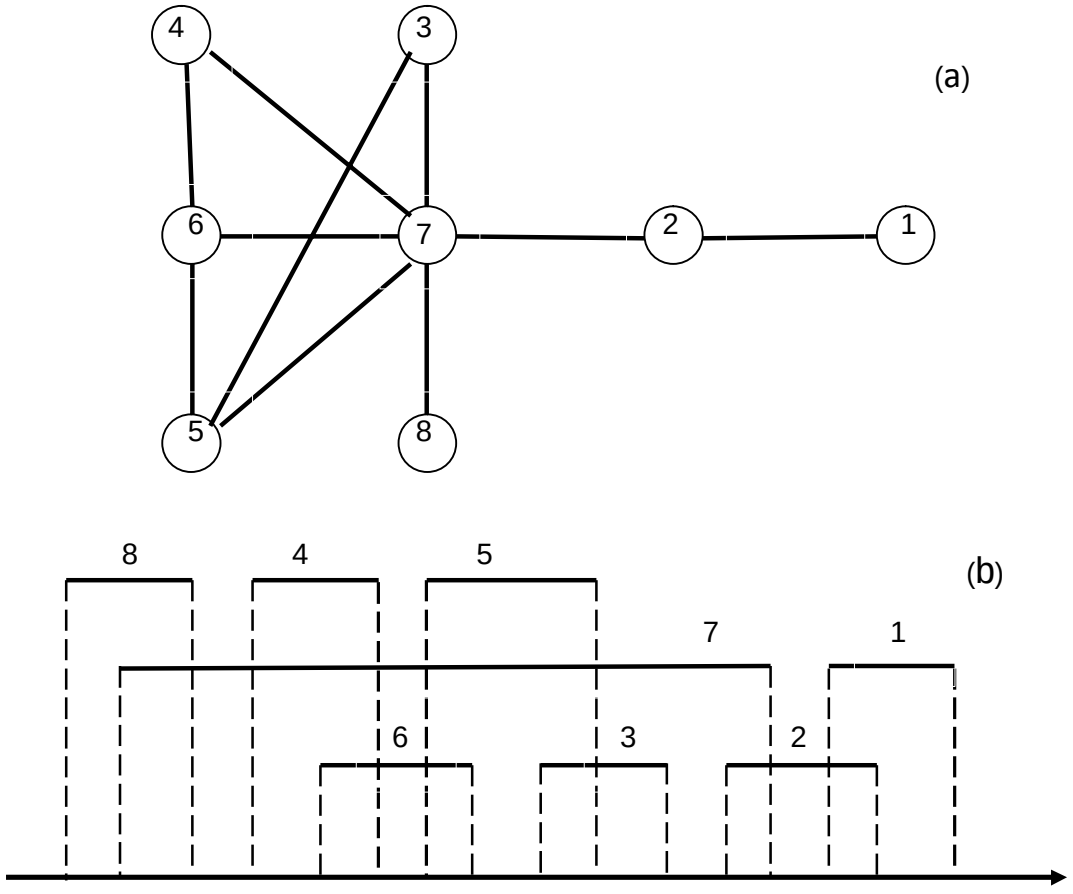
Следват добре познатите признаци на интервалните графи:

- *LE(u)* – левият край на интервала *u* на върха със същото име;
- *RE(u)* – десният край на интервала *u* на върха със същото име;
- *L-Block* – максималното непрекъснато множество от леви краища;
- *R-Block* – максималното непрекъснато множество от десни краища;

На Фиг. 1a е посочен един интервален граф, а Фиг. 1b показва едно интервално представяне от възможните за същия граф. Това представяне може удобно да се запише като подредена последователност от леви и десни краища:

LE(8), LE(7), RE(8), LE(4), LE(6), RE(4), LE(5), RE(6), LE(3), RE(5), RE(3), LE(2), RE(7), LE(1), RE(2), RE(1)

Отново ще се обърне внимание, че графът е интервален само ако е изпълнено условието, че когато два върха в графа са съседни, техните съответни интервали задължително се припокриват.



Фигура 1 : Интервален граф (a) и едно негово интервално представяне (b)

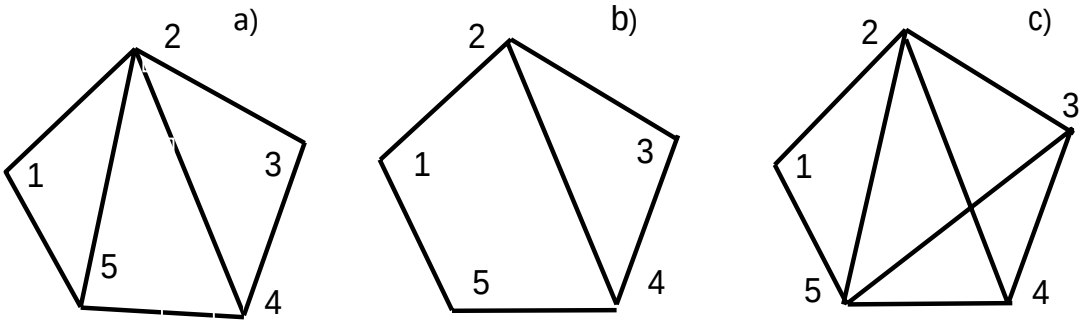
Тук е мястото да се спомене, че подреждането в рамките на един блок е без значение и каквито и да било пермутации не биха повлияли върху графа. Примерно, ако се удължи интервалът на връх 2 така, че *RE (2)* да дойде вдясно от *RE (1)* графът си остава неизменен. Това дава възможност да се дефинира следното важно свойство:

Един граф може да има много възможни интервални представяния. За един интервален граф ще се казва, че има уникален модел, ако всичките му интервални модели са еквивалентни.

При работа с ИГ задължително трябва да се споменат и хордните графи - един граф е хорден, ако не съдържа цикъл без хорди, състоящ се от 4 или повече върха. Интервалните графи са специален подклас на хордните графи. Левият граф на Фиг. 2а е хорден. Ако обаче се премахне

някоя от двете хорди, примерно *{5, 2}* графът престава да е хорден защото имаме цикъл *{1, 2, 4, 5}* (фигура 2b).

Симплициални върхове и клика – клика е пълен подграф в графа, т.е. всеки връх от подграфа е съсед на всички останали. Един връх *u (u ∈ V)* в графа *G = (V, E)* е симплициален, ако неговите съседи образуват клика.



Фигура 2 - Пример за хорден (a), нехорден (b) граф и клики (c)

В графа *G* показан на Фиг. 2c върховете *(2, 3, 4, 5)* образуват клика. Трябва да се отбележи, че множествата от върхове *(2, 3, 4), (2, 4, 5), (2, 3, 5)* и *(3, 4, 5)* също образуват клики. Кликата *(2, 3, 4, 5)* ще бъде дефинирана като *максимална клика* защото не се съдържа в друга, по-голяма като размер клика. Връх 4 е симплициален понеже неговите съседи *(2, 3, 5)* формират клика. Струва си да се отбележи, че условието за максималност не е задължително за симплициалните върхове.

Perfect Elimination Ordering (PEO) (Съвършена подредба за елиминация) за един граф е такова подреждане, при което за всеки връх *v* е в сила, че съседите му, намиращи се след него в подреждането и самият връх *v* образуват клика. Един граф е хорден тогава и само тогава, когато притежава поне едно PEO.

Хордните графи могат да бъдат разпознати като такива за линейно време чрез използването на Съвършената Подредба за Елиминация.

Един граф G е интервален, ако и само ако максималните му клики $C_1, C_2, C_3, \ldots, C_5$ могат да бъдат подредени последователно, т.е. те могат да бъдат разположени така $C_1, C_2, C_3, \ldots, C_5$, че ако $v \in C_i$ и $v \in C_j$ то $v \in C_h$ за всяко $i < h < j$.

Модул - Модул *M* в графа *G = (V, E)* е множество *S* от върхове така че:

a) *S* е свързано множество;

b) за всеки връх $u \in S$ и $v \notin S$ е в сила $(u, v) \in E$, ако и само ако $(u', v) \in E$ за всяко $u' \in S$;

В графа, показан на Фиг. 1 множеството от върхове *M = {3, 5, 6, 4}* е модул. Ако $1 < |M| < |V(G)|$ модулът е **нетривиален**. Един граф се нарича **S-prime граф** ако:

a) има поне 4 върхове;

b) в него няма нетривиални модули;

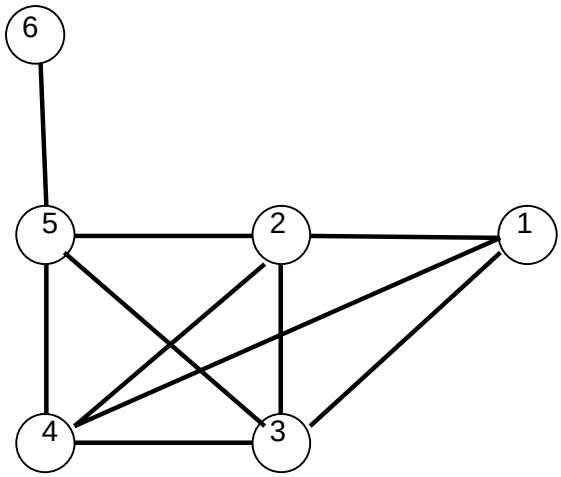
S-Декомпозиция – процесът на заместване на всеки нетривиален модул в графа с маркерен връх и същото направено рекурсивно за модула и за заместващия граф. Това определение още е наречено **модулна декомпозиция** или **заместителна декомпозиция**.

Процесът на модулна декомпозиция върху един граф води до конструирането на декомпозиционно дърво където всяко поддърво представя един нетривиален модул маркиран с неговия корен. Нека с *G*[M]* се означи резултатния граф. Ако *G*[M]* съдържа най-малко 4 върха тогава той трябва да бъде *S*-prime. Всеки такъв S-prime граф *G*[M]* се нарича **S-prime компонента** на G.

Един хорден граф *G*, несъдържащ подобни върхове е интервален граф, ако и само ако всяка *S-prime* компонента на *G* е също интервален граф.

Околност и подобни върхове – Околността *N[u]* за един връх *u* $(u \in V)$ в графа *G = (V, E)* е множеството от върхове, които са съседни на *u* в *G* плюс самия връх *u*. Околността записана с малки скоби означава: *N(u) = N[u] – {u}.* За графа от фигура 3 се получава:

N[1] = {1, 2, 3, 4} N(1) = {2, 3, 4} и подобно за останалите 5 върха.



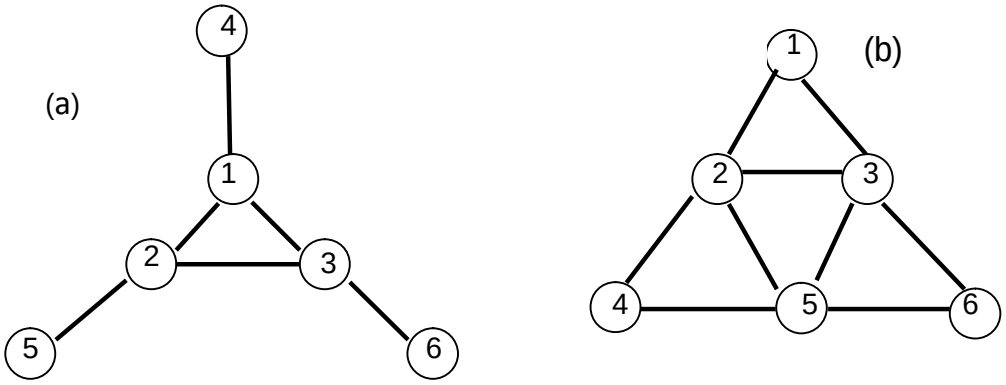
Фигура 3 – друг пример за хорден граф

Два върха *u* и *v* са **подобни,** ако *N[u] = N[v]*. Следователно за графа от фигура 3 върховете (*2, 3, 4)* са подобни. За едно подмножество от върхове *S (S ⊆ V)* с *N(M)* се означава множеството от върхове *V – S,* които са съседни на някой връх от *S*. С *G[S]* пък се означава индуцираният подграф от *G*, който включва върховете от *S*.

Астероидна тройка (AT) - такава тройка несвързани върхове в един граф, в която винаги съществува път между която и да е двойка върхове от тройката, който не минава през околността (съседите) на третия връх. Графите без AT (AT-free) са въведени от C. G. Lekkerkerker and J. C. Boland. Един AT-free граф е граф, който не съдържа нито една астероидна тройка. На фигура 4 са показани 2 класически примера за графи с AT. За графа на фигура 4а AT = *{4, 5, 6}.*

Един граф е интервален, ако и само ако е хорден и няма астероидни тройки (т.е. е AT-free граф).

***I* подредба (безчадърна подредба)** – Едно линейно подреждане $\curlyvee$ на множеството от върховете на един граф, такова, че за всяка тройка *u, v, w* за която е валидно *u* $\curlyvee$ *v* и *v* $\curlyvee$ *w, uw* $\in$ *E* да следва *uv* $\in$ *E.* Смисълът на *u* $\curlyvee$ *v* е, че *u* е преди *v* за дадената подредба $\curlyvee$.



Фигура 4 – Два примера за неинтервални графи с астероидни тройки

Всяка подредба на върховете на един граф, за която е вярно горното условие се нарича *I* подредба. Съответно, ако ръбът *uw* не отговаря на условието, т.е. налице е $uw \notin E$ то такъв ръб се нарича „чадър". Таблица 1 показва две LBFS подредби (разгледани по-късно) на графа от фигура 1.

Таблица 1 – пример за безчадърна подредба на втори ред

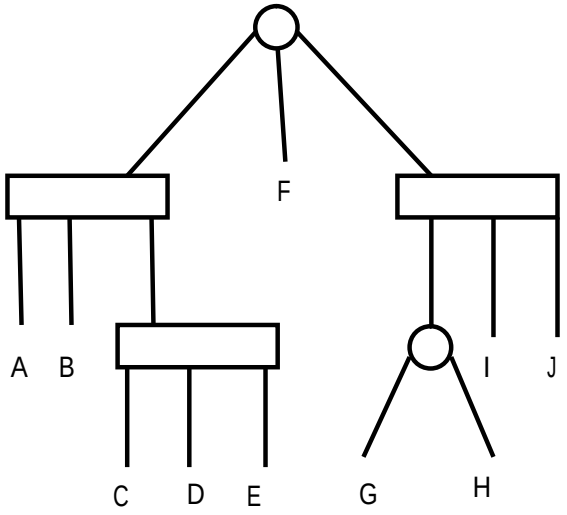
Подредба σ	**Безчадърна?**	**Забележки**
8, 7, 4, 6, 5, 3, 2, 1	Да	-
8, 7, 4, 5, 6, 3, 2, 1	Не	Съществува чадър между 4, 5

1.2. Основни структури от данни

PQ дървета - Едно *PQ* дърво е дървовидна структура от данни, която представя семейство от пермутации върху едно множество от елементи. Представлява етикирано дърво с корен, в което всеки елемент е представен посредством един от върховете-листа, а всеки връх, който не е листо е етикиран като *P* или *Q. P* върхът има най-малко два наследника, докато *Q* върхът има най-малко три наследника. Едно примерно дърво е показано на фиг. 5.

PQ дървото представя неговите пермутации чрез позволени пренареждания на наследниците на неговите върхове. Децата на един *P*

връх могат да бъдат преподредени по произволен начин. В отличие от тях, децата на един *Q* връх могат да бъдат преподредени единствено в обратен ред. Едно *PQ* дърво представя всички подреждания на върховете-листа, които могат да бъдат получени при произволна последователност на тия две операции. Едно *PQ* дърво с множество *P* и *Q* върхове е в състояние да представи сложни подмножества на множеството от всички възможни нареждания. На диаграмите е прието *P* връх да се се изобразява с кръгче, докато *Q* върхът се изобразява с правоъгълник.



Фигура 5 – Пример за PQ дърво

1.3. Lexicographic Breadth First Search (LBFS)

Повечето алгоритми за разпознаване на интервални графи първо проверяват дали графът е хорден. LBFS (лексикографското търсене първо в широчина) е парадигмата разработена от Rose et al. за ефикасно разпознаване на хордни графи като крайният резултат от търсенето е едно PEO. Следва LBFS алгоритъмът, както е даден в класическият му вариант:

(Забележка: ← съответства на „присвояване” в езиците за програмиране. Всеки връх има и атрибут „номериран” с две възможни стойности „ДА” или „НЕ”.)

Вход: свързан граф $G = (V,E)$ и зададен връх $u \in V$. С $|V|$ е означен броят на върховете в графа.

Изход: едно подреждане π_u на върховете от G, ако съществува.

1. Инициализирай $label(u) \leftarrow |V|$. Присвои на всички върхове атрибут „номериран" със стойност „НЕ".
2. За всеки връх v във $V - \{u\}$ прави label(v) $\leftarrow \lambda$.
3. За $i \leftarrow |V|$ надолу до 1

   3.1 Вземи неномериран връх v с лексикографско най-големия етикет.

   3.2 $\pi_u(|V| + 1 - i) \leftarrow v$. Задай за v атрибут „номериран" =„ДА".

Коментар: смисълът на ← тук е : постави v на позиция

$|V| + 1 - i$ във π_u

   3.3 За всеки неномериран връх w в $N(v)$ (в $N(v)$ са съседите на v) прави: добави i към label(w).

   3.4 Провери дали всички съседи на v, които са вдясно от него образуват клика. Ако „Да" – продължи, ако „Не" върни, че не съществува PEO, в което върхът u да е последен.

В стъпка 3.1 (често означавана и със „Стъпка ♦") може да се избере който и да е неномериран връх v, съвсем произволно. Известни са алгоритми, например, които не се нуждаят от тая стъпка, но все пак използват LBFS или някоя от неговите разновидности.

При друга разновидност на LBFS алгоритъма е използвана парадигмата на „подялбата". В началото всички върхове се намират в един начален раздел и се избира произволен връх от него. Често просто се взима първият връх в раздела. Това дава предимството да се допусне, че върховете имат вече някаква подредба. След като върхът е избран като опорен той се поставя в отделен, негов раздел и се пристъпва към разчленяване (подялба) на съществуващия раздел по следното правило – върховете, които са съседни на опорния образуват нов раздел, който предшества раздела съдържащ върховете, които не са съседни на опорния. След като текущата подялба е извършена се избира нов опорен

връх от раздела, непосредствено следващ раздела на стария опорен връх и процесът на разчленяване продължава. Едно от предимствата на показаната разновидност е, че непосредствено се получава разновидност на основния алгоритъм LBFS+ използвана по-нататък в настоящата глава.

1.4 Алгоритми за разпознаване на Интервални Графи

Първият алгоритъм за разпознаване на интервални графи е бил представен от D. R. Fulkerson and O. A. Gross, но неговото бързодействие е $O(n)^4$.

След тях K. S. Booth and G. S. Lueker предложиха *PQ* дърветата за разпознаването на интервални графи в реално време. Все пак трябва да се отбележи, че обработката на данните в *PQ* дърветата е доста заплетена. Десетилетие по-късно N. Korte and R. Möhring опростиха алгоритъма на Booth and. Lueker използвайки разновидност на *PQ* дърветата, която те нарекоха *MPQ дървета*. Отново, както и в първоначалния вариант, максималните клики играят ключова роля, но Korte и Möhring опростиха операциите върху *PQ* дърветата чрез намаляване на броя на шаблоните. Всички тия алгоритми разчитат на подреждането на максималните клики така, че за всеки връх *v* всички максимални клики, които го съдържат да са една до друга (последователно). Следователно, за същите алгоритми **предварителната компилация на всичките максимални клики е задължително условие**. Друг проблем е, подходът с *PQ* дървета изисква последователна обработка на данните, която не може да бъде паралелизирана. През 1991 г. се показа опростен декомпозиционен алгоритъм без използването на *PQ* дървета. Той се основава на *S*-декомпозицията на графа в реално време. В още по-ново време (1997 г.) M. Habib, C. Paul and L. Viennot разработиха друг линеен алгоритъм, относително по-лесен за имплементация. Същият използва LBFS подхода да определи дали даден граф е хорден и ако се окаже такъв да

състави едно „*дърво на кликите*”. Следващата стъпка е да се доведе това дърво до „*път на кликите*”. Ако това е възможно, то графът е интервален. Други нови техники (от 2003 г. насам) за разпознаване на интервални графи, с използването на PC дървета бяха предложени от W. L. Hsu and R. M. McConnell.

1.4.1. Подходът с PQ-дърветата

Това беше първият алгоритъм за разпознаване на интервални графи с линейна сложност за бързодействие. В техния труд K. S. Booth and G. S. Lueker не само въведоха *PQ* дърветата, но и определиха както правилата, така и шаблоните за манипулирането им. Това позволява да се подредят максималните клики по начин, който дава възможност да се реши дали даден граф е интервален.

Стъпките на алгоритъма са следните:

1. Проверка дали графът е хорден чрез извикването на LBFS. Ако графът не е хорден, то край на алгоритъма – графът не е интервален.
2. Намери всички максимални клики.
3. Създай *PQ* дърво с един елемент за всяка максимална клика в графа.
4. За всеки връх *v* създай множество I_v състоящо се от всички индекси на максимални клики, които съдържат *v*.
5. Използвайки *PQ* дървото, с помощта на правилата и шаблоните потърси такова нареждане на върховете, за което всички елементи на I_v са последователни за всеки връх *v* от графа.
6. Ако такова нареждане не съществува, то графът не е интервален. Иначе, конструирай интервалите за върховете с помощта на нареждането на максималните клики.

Анализът за сложност на алгоритъмът доказа, че той е с линейно бързодействие със сложност $O(n + 2m)$. Включването на константа във формулата е направено за сравнителен анализ с другите алгоритми.

Предимства:

- проверен във времето алгоритъм;
- служи като отправна точка за други алгоритми и изследвания в областта на интервалните графи;

Недостатъци:

- трудна и тромава манипулация на *PQ* дърветата;
- последователен алгоритъм по своята същност;

1.4.2. Подходът с модулна декомпозиция

Разработен от професор Hsu въз основа на модулната декомпозиция. Тоя алгоритъм пропуска етапа на намирането на максималните клики, следователно и сложните *PQ* дървета. За изложението на алгоритъма е необходимо да бъдат въведени някои допълнителни дефиниции.

Cardinality Lexicographic Ordering (CLO) – Лексикографско нареждане, получено чрез разкъсване на връзките в Стъпка ♦ на LBFS алгоритъма в полза на върха с максимална степен.

Строго Съседен (Strictly Adjacent) – два съседни върха *u* и *v* са строго съседни, ако *N[u]* и *N[v]* не се съдържат една в друга.

ST-Подграф G' – ST-подграфът *G'* от *G = (V, E)* е подграф с множество върхове *V* с множество ръбове *E'* от всички строго съседни двойки в *G*. Основните стъпки на алгоритъма са следните:

1. Премахни всички подобни върхове в графа.
2. Провери дали новополученият граф е хорден или не с помощта на LBFS.
3. Конструирай CLO.
4. Конструирай специален подграф *G''*. Този подграф служи за основа на изграждането на декомпозиционното дърво и ИМ.
5. Конструирай диаграма на Хасе (Hasse) за ограничителните връзки на елементите на *G''*.
6. От диаграмата на Хасе конструирай S-Декомпозиционно дърво.

7. Конструирай интервалния модел за всяка S-prime компонента на *G*. Ако при това конструиране се открие несъответствие то графът не е интервален. Иначе (ако несъответствие не е открито) върни интервалния модел на графа.

Алгоритъмът се основава на принципа, че ако всички *S*-prime компоненти на графа *G* могат да бъдат представени като интервални графи, то и самият *G* е интервален граф. Предимства:

- бързодействието на алгоритъма е със сложност *O(n + m)*, т.е. по-добро от това на предишния метод;
- манипулацията на данните е по-проста в сравнение с подходът на *PQ* дърветата; Недостатъци:
- алгоритъмът се счита за много сложен даже в средите на най-добрите специалисти в теория на графите;
- сред критиките спрямо метода е посочено, че са наблюдавани случаи, когато при приготвянето на *G"[D],* където *D* е множеството от всички несимплициални върхове, някои от елементите на *G"[D]* са несвързани.
- Друга критика е относно начинът, по който несимплициалните върхове са подложени на пермутация. Липсва строго формално описание. Това е съществен проблем, който може да доведе до противоречиви резултати при определени входни данни.

1.4.3. Multi-Sweep LBFS подход

Опитите за Multi-Sweep LBFS (Многократно сканиране с използването на LBFS) датират от 1997 г. когато M. Habib, C. Paul and L. Viennot разработиха 3-сканиращ LBFS. Година по-късно от D. Corneil, S. Olariu, and L. Stewart беше предложен 4-сканиращ (4-Sweep) алгоритъм, но и за двата беше доказано, че са некоректни. След дълга пауза едва в края на 2009 отново от тях (Corneil, Olariu, and Stewart) се публикува 6-сканиращ алгоритъм. Същите учени твърдят, че и 5-сканиращият

алгоритъмът работи правилно, но доказателството му било много по-сложно от това на 6-сканиращия.

За описанието на метода от текущия подраздел също е наложително да се въведат нови понятия и определения.

Слайс (Slice) - при изпълнението си класическият LBFS алгоритъм позволява избор на произволен връх в Стъпка ♦. Множеството от взаимнообвързани (да не се бърка с понятието съседни) върхове, изброени в Стъпка ♦ се нарича слайс и ще се обозначава с *S*.

Добър връх – Един връх *v* е добър, ако съществува някаква LBFS подредба, която завършва на *v.* Съответно, един връх ще се счита лош, ако няма нито една LBFS подредба, която да завършва на *v.*

Добра LBFS Подредба – такава подредба, при която всеки слайс започва с връх, който е добър за слайса.

Ако се означи с *μ* една LBFS подредба и един връх с *v* то следващите две понятия – **Стойност на Индекса на Съседство i_μ** (**Neighbour Index Value**) и **A множество** (или B множество) – могат да се дефинират така:

- Стойност на Индекса на Съседство $i_\mu(v)$ се явява най-голямата стойност *μ* от всички върхове, съседни на *v*;
- *A(v)* или *B(v)* е множеството от съседите на *v,* които са преди *v* в *μ* и са съседни на връх след *v* в *μ* (т.е. стойността на $i_\mu(v)$ е по-голяма от *μ(v));*

Доказано е, че 4- сканиращият алгоритъмът работи добре само ако първото сканиране е било **Добра LBFS Подредба**. Това обаче не може да се гарантира. От друга страна, 6-сканиращият алгоритъм гарантира, че третото сканиране е Добра LBFS Подредба, а оттам и коректност на целия алгоритъм.

Сега ще бъдат представени два варианта на LBFS алгоритъма, които ще се използват в основния 6-сканиращ алгоритъм:

Алгоритъм ***LBFS+ (G, μ)***

Вход: Граф *G*, и LBFS Подредба *μ*

Изход: Друга LBFS Подредба

В LBFS процедурата на Стъпка ♦ нека *S* бъде множеството от върховете с лексикографски най-големия етикет. При това условие *v* се избира така сред върховете на *S,* че да е последен в *μ*.

Алгоритъм ***LBFS*(G, μ1, μ2);***

Тоя вариант на LBFS се нуждае от две предишни LBFS сканирания, *μ1* и *μ2*. Ако е даден слайс *S* (така, както е обозначен в Стъпка ♦ на LBFS), то се избират два върха α и β където α е последния S-връх в μ1 и β е последния S-връх в μ2. LBFS* избира между *α* и *β* чрез прилагане на следните правила:

ако $i_{\mu 1}(\alpha) > \mu_1(\alpha)$ (т.е. *α* „лети") тогава се избира *β*.

иначе ако $i_{\mu 2}(\beta) > \mu_2(\beta)$ (т.е. *β* „лети") тогава се избира *α*.

иначе ако $(|B(\beta)|) = \lambda$ или $|A(\alpha)| \neq \lambda$, се избира *β*.

иначе нека *y* е един произволен елемент от *B(β);*

ако $i_{\mu 1}(y) = \alpha$ тогава се избира *β*.

иначе се избира *α*.

Алгоритъмът за разпознаване на интервални графи, който следва използва гореописаните варианти на LBFS. Той неявно проверява за хордни графи и графи, несъдържащи астероидни тройки. Основните стъпки са:

1. Приложи LBFS за произволен връх *x* и нека *y* е последният връх посетен при това сканиране, π′.
2. Използвайки π′, приложи LBFS+ (от *y*) и нека *z* бъде последният връх посетен при това сканиране, π.
3. Използвайки π, приложи LBFS+ (от *z*) където *y* би бил последният връх посетен при това сканиране и получи σ.
4. Използвайки σ, приложи LBFS+ (от *y*, завършващо на z), създай сканирането σ_+.
5. Използвайки σ_+ изчисли i_+-тата подредба за LBFS σ_+ и множеството *A*.

6. Използвайки σ_{+}, приложи LBFS+ (от *z*, завършващо на *y*), създай сканирането σ_{++}.
7. Използвайки σ_{++} изчисли i_{++}-тата подредба за LBFS σ_{++} и множеството *B*.
8. Използвайки σ_{+} и σ_{++}, приложи LBFS*, създай сканирането σ_{*}.
9. Ако σ_{*} е безчадърен граф тогава *G* е интервален граф. В противен случай G не е интервален.
10. Ако *G* е безчадърен граф, то изчисли интервалния модел чрез представяне на връх *v* посредством интервала $[\sigma_{*}(v), i_{*}(v)]$.

Предимства: Алгоритъм с относително нетрудна имплементация. Бързодействието на алгоритъма е със сложност $O(n + m)$;

Недостатъци: Включва LBFS, паралелизацията на който е изключително трудна;

1.5 Паралелизъм при алгоритмите свързани с ИГ

Добре е да се подчертае, че когато говорим за паралелни алгоритми върху интервални графи се има предвид целия спектър от проблеми не само свързани с разпознаването им, но и връзката на ИГ с други видове графи както и конкретното им приложение в дадени предметни области. Последното налага някои нови ограничения или изисквания, които не са от интерес за класическите схеми, но са от значение при практическата реализация на проблем от дадена област и се радват на голям научен интерес в последните две десетилетия.

Освен това в практиката се наблюдават и усложнени разновидности като например теглови интервални графи. И накрая, но не и по значение е с какви хардуерни платформи и софтуерни варианти на паралелизъм разполагат изследователите.

При паралелизация за разпознаването на интервалните графи не се подразбират някакви строго определени, изцяло паралелни методи, които са алтернатива на класическите алгоритми за определяне дали

даден граф е интервален, просто защото такива не съществуват. По-скоро в зависимост от избрания класически метод се анализира кои негови стъпки могат да бъдат ускорени чрез използването на паралелизация. Колкото повече такива стъпки съществуват, толкова и крайният ефект е налице като повишено бързодействие. Последното е изключително важно за някои предметни област като генетиката, където размерът на графите, които се проверяват е много голям, а понякога и огромен - т.е. практически е невъзможно решението да бъде получено без прилагането на някаква форма на паралелизъм.

1.6 Изводи

Известни са няколко различни парадигми, всяка със своите модификации за разпознаване на интервални графи. Предимствата, ограниченията и проблемите, съпътстващи всяка от тях са добре известни и достатъчно анализирани в научните среди. На таблица 2 са резюмирани основните аспекти на всеки от трите подхода и техните недостатъци.

Към алгоритмите за разпознаване на интервални графи се появяват все по-високи изисквания по отношение на бързодействие, надеждност и възможност да работят в многонишкова, многопроцесна и/или многопроцесорна среда. **Засега не е показан базов алгоритъм, който да позволява ефикасна паралелизация на ниво задача на процеса на обработка на върховете.** От друга страна нарастващия обем данни в много предметни области, сред които задължително трябва да се упоменат генетиката и археологията, водят до графи с милиони върхове, при които класическите алгоритми за разпознаване работят прекалено бавно.

Друга особеност е, че досега не е известен **метод, при който самото интервално представяне да управлява процеса**, да му влияе или да налага ограничения по време на работата на конкретния алгоритъм. Това е една възможност, една ниша, която ще бъде опитано

Таблица 2 – основни характеристики на алгоритмите за разпозн. на ИГ.

Метод	Характеристики	Недостатъци	Имплементация
PQ-дървета	• Изисква шаблони • Дефинира правила върху шаблоните • Включва LBFS • Изисква предварително изчисление на всички максимални клики • Пресмята интервалите за всеки връх от подредби на максималните клики • Има няколко разновидности в опити да бъде подобрен	• Тромава манипулация върху дърветата • Не използва интервалното представяне при работата си • Правилата са сложни • Строго последователен алгоритъм, неподходящ за паралелизация	• Труден за имплементация
Модулна декомпозиция	• Включва LBFS • Изисква изчисляване на всички S-prime компоненти • Включва конструиране на специален подграф и конструиране на диаграма на Hasse за връзките в него • Конструиране на S-декомпозиционно дърво от диаграмата на Hasse	• Сложни манипулации • Тежки теоретични доказателства • Изисква предварително премахване на подобните върхове в графа. • Има критики относно теоретичната коректност при конструиране на специалния подграф	• Особено тежък за имплементация • Не предоставя сертификат за решението
Multi-Sweep LBFS	• Включва LBFS, при това в няколко разновидности • Търси за специална „безчадърна” подредба • Най-често използван от съществуващите досега	• Строго последователен алгоритъм, неподходящ за паралелизация • Сложно теоретично доказателство • Бавен заради многото сканирания. • Неподходящ при особено големи графи	• По-лесен за имплементация от другите 2 подхода

да се запълни с настоящия труд. Задаването на един много ограничен брой от правила позволява да се управлява както самото строене на

интервалния модел, така и ефикасното откриване на всички графи, които не са интервални.

1.7 Цел и задачи на дисертационния труд

Направените по-горе обобщения относно характеристиките и недостатъците на съществуващите техники за разпознаване на интервални графи позволяват да се формулира основната цел на дисертационната работа:

Представяне и подробно доказателство на нов алгоритъм за разпознаване на интервални графи, който да дава еднозначно решение за всеки входен ненасочен свързан граф.

За постигането на поставената цел е необходимо да бъдат решени следните основни задачи като са спазени и посочените изисквания:

1. Интервалното представяне да участва в процеса на доказателство.
2. Да се извлекат правила от топологичните свойства на ИП, чието нарушение класифицира входния граф като неинтервален. Броят на тези правила да е възможно най-малък.
3. Да се изведе формално доказателство за валидността на всяко от предложените правила.
4. За всеки граф, разпознат като интервален да се връща едно ИП, където интервалните краища са цели положителни числа.
5. Да се докаже, че алгоритъмът е перспективен за паралелизация като се очертаят основните стъпки, позволяващи многонишкова имплементация;
6. Да се докаже, че полученото ИП се явява сертификат, което дава основание новият метод да бъде причислен към категорията на сертифициращите алгоритми.
7. Новият метод, за разлика от всички съществуващи никъде да не използва LBFS.

8. Новият алгоритъм да се имплементира на някой от езиците за програмиране и се тества върху достатъчно голям брой графи.
9. Да се анализират отделните стъпки и от гледна точка на бързодействието. Да се докаже, че алгоритъмът има линейна сложност и чрез резултатите от тестовете.

Глава 2. Новият алгоритъм за разпознаване на ИГ

Преди да се разгледат конкретните етапи на новия метод е необходимо, отчитайки всичко, което вече е казано в раздел 1.1, да се изброят и обяснят новите понятия, които ще бъдат използвани. Също така ще се въведат помощни алгоритми, явяващи се неотменими стъпки на цялостното решение.

2.1 Външен връх, вътрешен връх и Помощен Алгоритъм 1

v_adj – брой съседни върхове за връх *v*. Ако *v1* и *v2* от един ИГ са съседи и *v1* има поне един съсед, който не е съсед на v2 ще наричаме *v1* **външен спрямо** *v2*. Основание за това е, че интервалът на представяне на *v1* никога не може да бъде изцяло вложен в тоя на *v2* за който и да е ИМ. Обратно, ако *v1* няма нито един такъв съсед ще наричаме *v1* **вътрешен спрямо** *v2*. Сега е вярно *v1_adj* <= *v2_adj* и винаги съществува интервално представяне на *v1* изцяло вложено в интервала на *v2*. Следва помощен алгоритъм, с който се проверява в какво отношение са един спрямо друг два съседни върха - дали един връх *v1* е външен или вътрешен спрямо друг връх *v2.* За целта всеки връх от графа трябва да притежава един атрибут, да го означим *check* с две възможни стойности, примерно *COLOR_1* и *COLOR_2*. Алгоритъмът е следния:

Помощен алгоритъм 1

1. За всеки съсед на *v1* присвои *check* = *COLOR_1*
2. За всеки съсед на *v2* присвои *check* = *COLOR_2*
3. Ако *v1* има поне един съсед *v3* с атрибут *check* равен на *COLOR_1* (т.е. *v3* не е съсед на *v2*) то *v1* е външен спрямо *v2.*

Иначе *v1* е вътрешен спрямо *v2.*

2.2 Основни понятия и определения

Обработен връх – такъв връх, на който интервалните краища (ИК) са изчислени. Съответно **необработен връх**. За входният граф всички върхове са необработени.

NPV – брой обработени върхове в графа. По-надолу в основния алгоритъм винаги, когато се намери вторият интервален край на даден връх и се маркира като обработен се прави *NPV*++, но не се указва явно.

NGV – общ брой върхове в графа. Когато *NPV = NGV* графът е нареден. Ако за ИК на върховете се използват само цели положителни числа започващи от 1, то минималният брой числа необходими за подреждането на целия граф е *2* NGV.* Целта, която се поставя в настоящия труд е да се намери едно ИП точно в интервала *[1, 2* NGV].*

При определяне на интервалните краища ще се употребяват две глобални променливи *upper* и *lower* инициализирани така: *upper = 2*NGV* и *lower = 1*. Т.е. началото и краят на бъдещото интервално представяне. Единствените разрешени операции са декремент *upper*-- за първата и инкремент *lower*++ за втората. Кога да се ползва всяка от променливите ще се определя от логиката на алгоритъма.

Пълен връх – Такъв връх *v* е съсед на всички върхове в графа. Неговите ИК могат да се изчислят веднага.

Единичен връх – Такъв връх *v* е съсед само на пълните върхове в графа. ИК на единичен връх могат да се намерят веднага след като са обработени пълните върхове.

С обработените до даден момент върхове може да се постъпи по 2 начина – или да се извадят от текущия граф или за всеки връх да се въведе атрибут (примерно processed) с 2 възможни стойности true/false съответстващи на обработен/необработен. Ако се използва втората възможност то две от въведените до тук понятия може да се предефинират:

v_adj – брой съседни *необработени* върхове за даден връх *v*.

Пълен връх – Такъв връх *v* е съсед на всички *необработени* върхове. За него е в сила:

v_adj = NGV- NPV-1.

2.3 Помощен Алгоритъм 2 за избор на начален връх

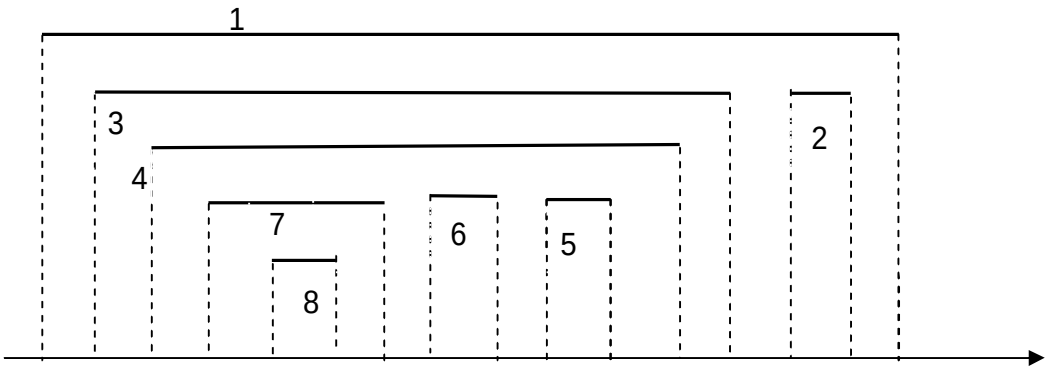
Ключов момент за алгоритъма е намирането на **начален връх** *SV* (още наричан стартов връх)– изискването за него е, че **той не трябва да е вътрешен спрямо друг необработен връх в графа**. Помощният алгоритъм (ПА) за намиране на *SV* представен тук обработва също така пълните и единични върхове:

Помощен алгоритъм 2 за SV (НамериSV)

NPV = 0 Прави

1. Взима се произволен връх *v* от множеството на необработените върхове и се задава *SV = v*.
2. За всеки *vn* от необработените съседи на *v* се проверява:
   Ако *vn_adj > SV_adj* то *SV = vn*.
3. Ако *SV_adj = NGV- NPV-1* (т.е *SV* е пълен връх) се прави:
   LE(SV) = lower++; RE(SV) = upper--; SV(Processed) = true; NPV++
   За всеки *v* с *v(Processed) = false* прави *v_adj--*
   За всеки *v* с *v_adj = 0* (открит е изолиран връх) прави:
   RE(v) = upper--;LE(v) = upper--;v(Processed) = true; NPV++;

Докато *(SV_adj = NGV- NPV-1)*



Фигура 6 – ИП на граф, който се нарежда само от *ПА 2*

От Фиг. 6 се вижда, че има случаи, когато *Помощен алгоритъм 2* може да намери ИП на един граф без да е необходима допълнителна обработка. За графът на Фиг. 6 последователно ще се наредят върховете 1, 2, 3, 4, 5, 6, 7 и 8. Заслужава да се отбележи, че при нареждането на единичните върхове може да използва както *upper,* така и *lower.* При използването на *upper*, координатите на краищата са *v1=[1, 16] v2=[14, 15] v3=[2, 13] v4=[3, 12] v5=[10, 11] v6=[8, 9] v7=[4, 7] v8=[5, 6].*

Друг основен аспект е когато *SV* е пълен връх **и началният граф може да се разпадне на два или повече несвързани подграфа**.

Ако с *ПодредиЕдинПодграф* се означи подпрограмата (алгоритъмът) за нареждане на един подграф, а *НамериSV* е *Помощен алгоритъм 2,* то основният алгоритъм (ОА) ще изглежда така:

Прави
  Извикай *НамериSV*
  Ако (*NPV* < *NGV*)
    Извикай *ПодредиЕдинПодграф*
Докато (*NPV* < *NGV*)

Надолу в предложената работа се разглеждат изискванията към и построяването на алгоритъма на *ПодредиЕдинПодграф.*

2.4 Група от правила, валидни за всяко ИП

Самата същност на интервалния модел на един интервален граф позволява да се дефинира **група от правила, които да са в сила за всеки ИГ**. Обратното също е вярно, нарушението на някое от правилата е свидетелство за това, че графът не е интервален.

2.4.1 Правило 1 за външните съседи на един връх

Първото правило е валидно за всеки връх, който има външни съседи:

Правило 1: За произволен интервален модел на един интервален граф всички външни съседи от едната страна на даден връх *v* са и съседи помежду си, т.е. образуват клика.

Доказателство: Интервалите на всички външни съседни върхове от едната страна включват единия край на *v*. Следователно, всички техни интервали се припокриват в тоя край. Съгласно определението за ИГ, всичките външни върхове (от единият край) трябва да са съседи помежду си.

Правило 1 ще е в сила и за *SV* когато той има външни съседи. Да не се забравя, че от друга страна, ако *SV* няма външни съседи, неговите интервални краища могат да се изчислят веднага и да се пристъпи към намирането на следващ начален връх. Т.е. наличието на съседи поне от едната страна на началния връх е естественото състояние на нещата при обработката на интервални графи.

2.4.2 Помощен Алгоритъм 3 за външните съседи на един връх

Следващият помощен алгоритъм намира съседите от едната страна на *SV* и ги поставя в множеството *SAV*.

SAV – (Множество на Активните Върхове) основно понятия за алгоритъма. Първоначално в него са всички върхове, които са съседи на началният връх от едната му страна и самият начален връх.

Помощен алгоритъм 3 (ПА 3) (*SV* е вече намерен)

Инициализира се *SAV* = Ø.

За всеки външен необработен съсед *vn* (намерен с *Помощен алгоритъм 1*) на *SV* се прави :

1. Ако *vn* е първият открит външен спрямо *SV* то:

   vn се поставя в *SAV*, запомня се в отделна променлива като *FL* (първи външен) и се запомня неговият *v3* (описан в *Помощен алгоритъм 1*) в друга променлива *first_v3*. (т.е. *v3* и *SV* не са съседи)

   Иначе:

   Ако *FL* и *vn* не са съседи, то *vn* е от другата страна на *SV* заради *Правило 1*

   Иначе

   Ако

   first_v3 и *v3* са съседи (*vn* и *v3* на Фиг. 7) или

   first_v3 и *vn* са съседи (*vn* и *v3* на Фиг. 8) или

   FL и *v3* са съседи (*vn* и *v3* на Фиг. 9) или

   first_v3 и *v3* съвпадат (Фиг. 10)

   то постави *vn* в *SAV.*

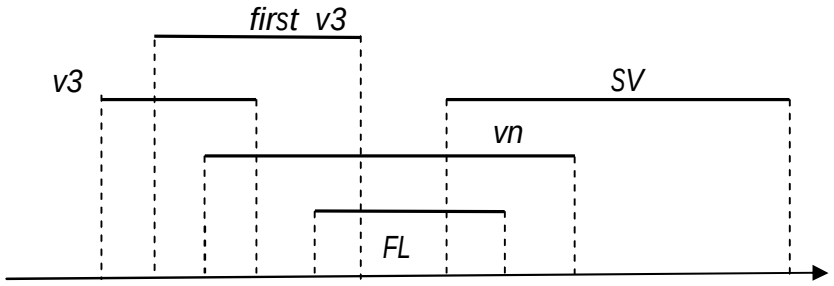
   Иначе *vn* е от другата страна *SV* .

2. Ако *vn* е поставен в *SAV* и не е от първите 3 поставени то се проверява дали той е съсед на всички предишни от *SAV* – т.е. прилага се *Правило 1*. Ако *vn* не е съсед на някой от *SAV* - край на основния алгоритъм, входният граф не е ИГ.

В точка 1 са изброени **всичките 4** възможни отношения между *first_v3* и *v3* когато *vn* е от страната на *FL* **за който и да е** *ИМ.* Ако нито един от тях не в сила, то *v3* е от другата страна на *SAV*, следователно и *vn* е от другата страна.

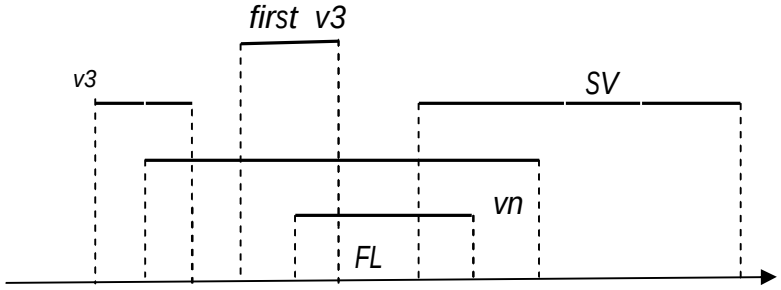
Ако *vn* е върхът, за който трябва да се реши и неговият съсед *v3* е съсед на *first_v3* (Фиг. 7) то не е възможно *vn* да бъде от другата страна. Защото при допускане на това интервалът на *vn* би обхванал *SV* - оттук следва, че *SV* е вложен във *vn,* а това противоречи на изискването за намиране на началния връх.

Ако *vn* е върхът. за който трябва да се реши и неговият съсед *v3* не е съсед на *first_v3,* когато *vn* е от същата страна, както и *FL* то може да има само 2 случая:



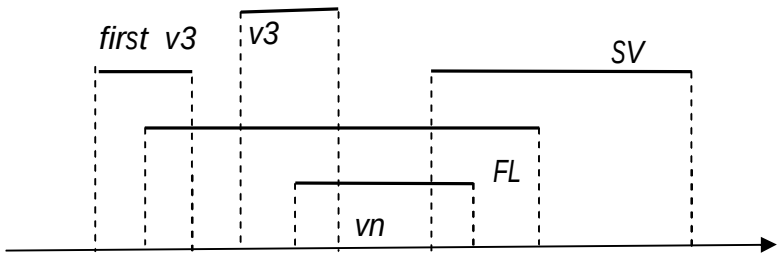
Фигура 7 – първоначално попълване на върховете в SAV, първи случай

- когато *first_v3* се намира между *SV* и *v3.* В тоя случай *first_v3* и *vn* винаги са съседи (Фиг. 8). Може да се каже и по друг начин: ако *first_v3* и *vn* са съседи то *vn* е от същата страна, както и *FL*. Допускането на противното води до това, че *SV* е вложен във *vn,* а това противоречи на изискването за намиране на началния връх.



Фигура 8 – първоначално попълване на върховете в SAV, втори случай.

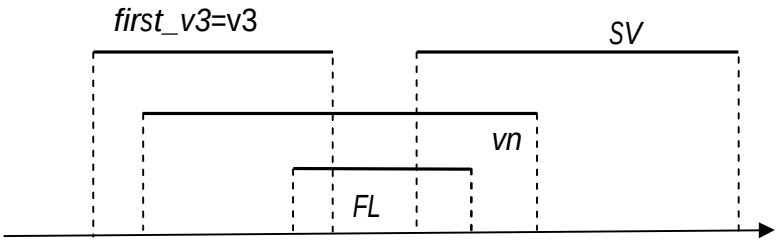
- *vn* е от другата страна



Фигура 9 – първоначално попълване на върховете в SAV, трети случай

- Когато *v3* се намира между *SV* и *first_v3.* В тоя случай *v3* и *FL* винаги са съседи (Фиг. 9). Може да се каже и по друг начин: ако *v3* и *FL* са съседи то *vn* е от същата страна, както и *FL*. Допускането на противното води то това, че *SV* е вложен във *vn,* а това противоречи на изискването за намиране на началния връх.

Възможно е върхът *v3,* който да прави *vn* външен спрямо *SV* да съвпада с *first*_v3*.* Отново *vn* (показан на Фиг. 10) е от същата страна, както и *FL*. Допускането на противното води то това, че *SV* е вложен във *vn,* а това противоречи на изискването за намиране на началния връх.



Фигура 10 – първоначално попълване на върховете в SAV, четвърти случай

2.4.3 Правило 2 за съседите на най-близкия връх

За да се въведе второто правило отново ще се разгледат външните съседи от едната страна на началния връх *SV*. Нека множеството *SAV* да включва всички тези съседи заедно със самия *SV.* За $\forall v \in$ първоначалното *SAV* може да се намери отделно множество *SCV(v)* така, че за $\forall v2 \in SCV(v)$ да са в сила следните три условия:

a) *v* и *v2* са съседи;

b) v2 и *SV* са съседи;

c) v2 $\notin$ *SAV*;

Нека с *num_SCV(v)* се означи броят на върховете в *SCV(v).* Нека с $NV \in SAV$ означим тоя връх, за който множеството *SCV* е най-малко, т.е. *num_SCV(NV) = minimum.* Ако има няколко върха с намерен минимум

вместо единичен *NV* те се поставят в множеството *SNV* и следващото правило важи за всеки от тях. Посоченият начин за пресмятане на *SCV* се използва **само за първоначалните елементи** на *SAV* и то **в първата посока**.

Правило 2: За произволен интервален модел, ако *v2* ∈ *SCV(NV)* то *v2* е съсед на ∀*v* ∈ *SAV.* Самото доказателство се пропуска тук в реферата.

2.4.4 Правило 3 за зависимостите на ИК между 2 върха

Третото и последно правило отново е свързано с върховете в *SAV*.

Правило 3: За произволен интервален модел където в *SAV* са левите съседи на *SV* и самия *SV*: ако *v* ∈ *SAV*, *v* е от първоначалните в *SAV, v* ≠ *SV, v2* е съсед на *v* и *v2* не е съсед на *SV,* то *RE(v2)* < *LE(SV)*.

За всеки *v2* за който е в сила Правило 3 може да се присвои атрибут „*LEFT_FROM_SV*". Това присвояване може да стане едновременно с първоначалното намиране на *SAV.* Помощен алгоритъм 4 намира всички върхове с атрибут „*LEFT_FROM_SV*".

Помощен алгоритъм 4 (прилага се само за първоначално поставените върхове в *SAV* в първа посока)

За всеки *v* ∈ *SAV* и *v* ≠ *SV* прави,

За всеки *v2* съсед на v

Ако *v2* не е съсед на *SV* то

присвои на *v2* атрибута *LEFT_FROM_SV*

2.5 Алгоритъмът на ПодредиЕдинПодграф

По надолу в алгоритъма за всяко от множествата *SIV,* и *SOV* броят на върховете в тях се означава съответно с *num_SIV* и *num_SOV.* Следват основните стъпки на алгоритъма:

1 Намери *SAV (ПА 3).* Запомни върховете от *SAV* без *SV,* т.е.: *KEEP = SAV - SV*

direction = 1 (променлива за посоката, 1 - първа посока)

2 Намери върховете с атрибут *LEFT_FROM_SV*

3 За $\forall v \in SAV$ намери *SCV(v)* (Както е описано преди Правило 2)

4 Намери $NV \in SAV$, за който

num_SCV(NV)=minimum

5 Раздели *SCV(NV) на две:*

SCV(NV) = SIV(NV) + SOV(NV), където в *SIV (NV)* са вътрешните, а в *SOV (NV)* външните върхове спрямо *NV.* Външните са спрямо десния интервален край на *NV*.

6 Ако num_*SCV(NV) <=3,* то изчисли координатите на върховете от *SIV(NV)* и левите координати на върховете от *SOV(NV).*

Иначе:

6.1 Създай нов граф G' само от *SCV(NV)*

6.2 Извикай рекурсивно ОА за него.

Ако за стария граф G *SOV(NV)* != Ø

a) firstSubgraphHasOuterVert = true

b) Ползвай модифициран вариант на *НамериSV* (разгледан по-късно) за подграфа от G' където участват само върховете от *SOV(NV)*

Иначе *firstSubgraphHasOuterVert = false*

6.3. Изчисли координатите на върховете от *SCV(NV)* въз основа на координатите на върховете от *G'*. За върховете от *SOV* се изчисляват само левите координати.

7. Изчисли *RE* за *NV* и всички върхове *v*, за които num_*SCV(v)*= num_*SCV(NV).*

8. Преизчисли *SAV = SAV - SNV + SOV.* За $\forall v \in SAV$ се преизчислявя num_*SCV(v).* Тоест за $\forall v \in SAV$ прави:

Ако $v \notin SOV$, то

num_*SCV(v)* = num_*SCV(v)*- num_*SOV(v)*- num_*SIV(v)*

Иначе ($v \in SOV$) се използва следния алгоритъм:

num_*SCV(v) = 0.*

За всеки *v2* съсед на *v*

Ако

v2 не е обработен и

v2 не е съсед на *NV* и

v2 не е самия *NV*

то num_*SCV(v)* ++

Едновременно с намирането на всички num_*SCV* се намира и *NV* (*num_SCV(NV) = minimum)*

9. Ако *SAV* != Ø отиди на т. 5

10. Ако *direction = 1* то

a) Отмести всички изчислени дотук ИК надясно така, че най-десния интервален край да стане равен на 2* *NGV*

b) Ако *KEEP* = Ø отиди на т. 11

Иначе *SAV = KEEP.*

c) За всеки $v \in SAV$ прави

LE(v) = RE(v)

d) За всеки $v \in SAV$ намери *SCV(v)*

(Подробностите са описани в гл. 3)

e) *direction++*

f) отиди на т. 4 (за да се намерят ИК във втората посока)

Иначе (т.е. *direction = 2)*

a) Преизчисли координатите на върховете от *KEEP*

b) Преизчисли координатите на останалите върхове, обработени във втората посока

(a) и b) са разгледани в глава 3)

11. Ако *firstSubgraphHasOuterVert == true* прави

a) Ако текущият граф не е нареден (т.е. има и други подграфи освен първия) то текущият подграф се премества най-отдясно, ако все още не е там.

b) Разместват се, ако е необходимо, интервалните краища за текущият подграф така,
че за върховете, които съответстват на тия от *SOV* десните краища да отидат най-вдясно;

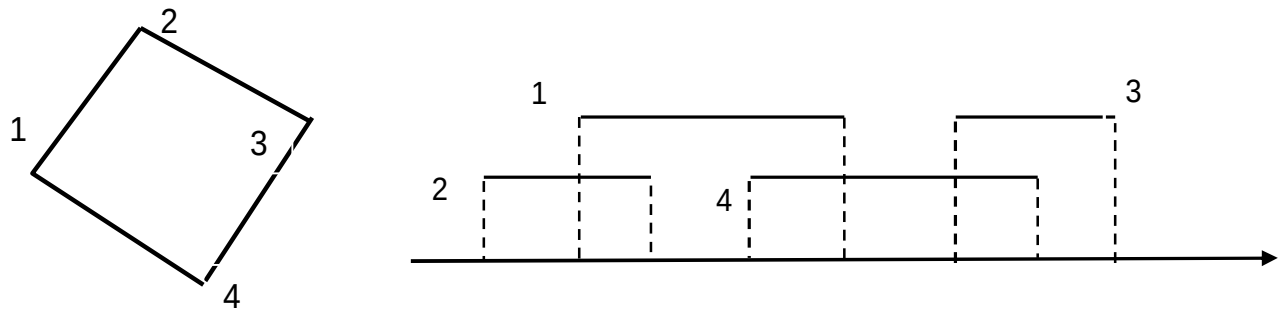
c) *firstSubgraphHasOuterVert = false*;

12. Сертификация – построява се граф от интервалното представяне и се сравнява с началния граф.

2.6 Приложение на Правила 1, 2 и 3

И трите правила се прилагат по време на построяване на интервалното представяне. Правило 1 се прилага при намиране на първоначалните върхове в *SAV* и при намиране на външните върхове спрямо текущия *NV*. Правилото ще сработи винаги, когато има поне един липсващ ръб между някои членове на *SOV* или *SAV*.

Правило 2 също се прилага при обработка на съседите на *NV* с цел разделянето им на външни и вътрешни – т.е в стъпка 5 от основния алгоритъм от предишния раздел. Правилото ще сработи винаги, когато има поне един липсващ ръб между някой съсед на *NV* и поне един от върховете в текущото *SAV*.



Фигура 11 – Възможно най-простият неинтервален граф и грешното му ИП

Употребата на Правило 3 е по-сложна. Да разгледаме следния пример от фигура 11: Граф с върхове *{1, 2, 3, 4}* и ръбове *{1, 2}, {2, 3}, {3,*

4}, и *{1, 4}.* Това е един нехорден граф заради липсата на хорда (ръб) между върхове 1 и 3 или между върхове 2 и 4.

Ако липсва правило 3 нека имаме в *SAV*: *SV = 1, a* другият член на началното *SAV* е връх 2. Върхът, заради който връх 2 е обявен за външен е връх 3 и неговото интервално представяне трябва да е отляво на интервала на *SV*. Алгоритъмът ще подреди интервалните краища в следната последователност: *LE(2), LE(1), RE(2), LE(4), RE(1), LE(3), RE(4), RE(3)* (фигура 11). Обаче при наличие на Правило 3 при опит връх 3 да бъде обработван в дясна посока правилото ще сработи и графът ще се класифицира като неинтервален. Правилото трябва да се прилага, както и предишното винаги на стъпка 5, но само в първата посока за съседите на текущия *NV.*

Да резюмираме казаното дотук – ако един граф не е интервален заради липсващ ръб ще сработи Правило 1 или 2. Но ако имаме обратното, т.е. графът да не е интервален заради допълнителен ръб възможните варианти са повече и може да сработи всяко от трите правила. Общото за разгледаните случаи е, че Правило 3 ще сработи само ако имаме цикъл, т.е. графът не е хорден. Тоест, ако знаем, че графът е хорден няма нужда от това правило.

2.7 Допълнителни пояснения по т.6 за num_SCV(NV) <=2

Когато броят на съседите на *NV* е малък интервалните краища на върховете може да се изчислят веднага. Най-лесен е случаят, когато *NV* няма съседи. В тоя случай веднага се слага десният край на *NV*.

Когато *NV* има само един съсед *v* съществуват само два случая:

a) v е вътрешен спрямо *NV*; *b) v* е външен спрямо *NV*;

Ако *f* (съответства на описаната по-рано *lower*, но се ползва тук за удобство като по-кратка) е целочислена променлива с някаква текуща стойност преди поставяне на интервалните краища на *v*, то за първият случай е сила:

LE(v) = f++; RE(v) = f++; RE(NV) = f++; а за втория:

LE(v) = f++; RE(NV) = f++;

Когато *NV* има два съседа *v1* и *v2* съществуват пет случая, ако се подсигури, че когато единият връх е външен, а другият вътрешен то *v1* е вътрешният, а *v2* – външният. Т.е. ако имаме обратния случай върховете просто се разменят в метода изчисляващ интервалните краища. Случаите са подобни на тия от следващия раздел и не се разглеждат тук.

2.8 Допълнителни пояснения по т.6 за num_SCV(NV) =3

Когато общият брой на върховете от *SIV* и *SOV* е равен на три възможните случаи, макар и значително повече в сравнение с предишната подточка могат да бъдат обработени посредством въвеждането на допълнителни правила, които ще се разгледат тук.

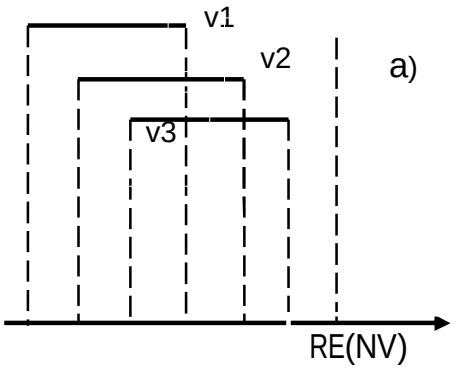
Правило 1: Ако върховете от *SIV* и *SOV* са в променливите *v1, v2,* и *v3* и приемем, че основната подредба е *v1, v2, v3* то върховете се разместват така, че вътрешните да са отляво, а външните отдясно. Ако с *i* се означи вътрешен връх, а с *o* – външен, то след прилагането на *Правило 1* може да има само 4 варианта - *i,i,i* или *i,i,o* или *i,o,o* или *o,o,o.*

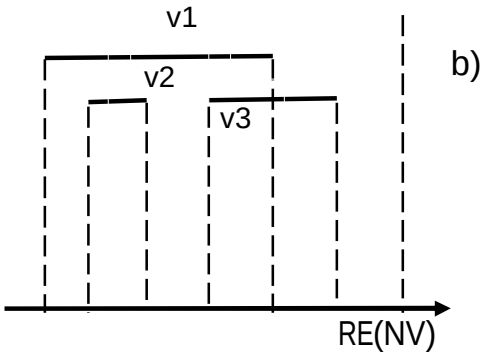
Правило 2: Ако имаме варианта *i,i,i* и само един връх е съседен на другите два, то тоя връх трябва да бъде *v1.*

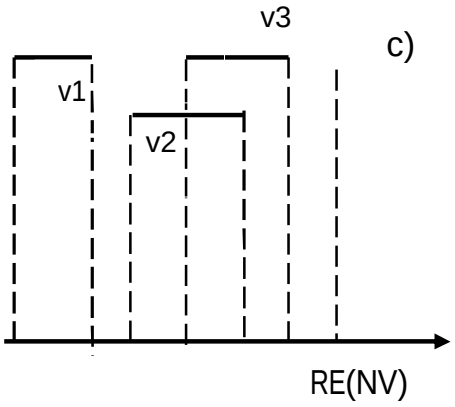
Правило 3: Ако имаме варианта *i,i,i* и един връх няма съседи, а другите два са съседи, то върхът без съседи трябва да бъде *v1.*

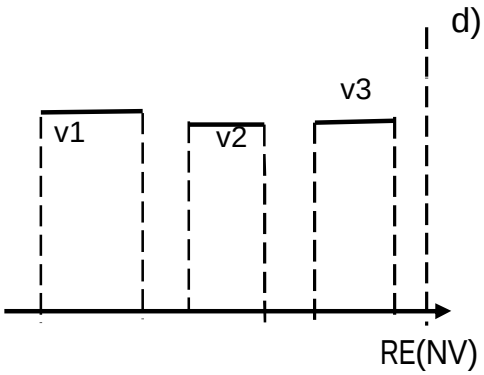
Правило 4: Ако имаме варианта *i,i,o* и само единият от вътрешните върхове е съседен на външния, то тоя връх трябва да бъде *v2.* Тоест, ако *v1* е съседен на *v3, а v2* не е, то върховете във *v1* и *v2* се разместват.

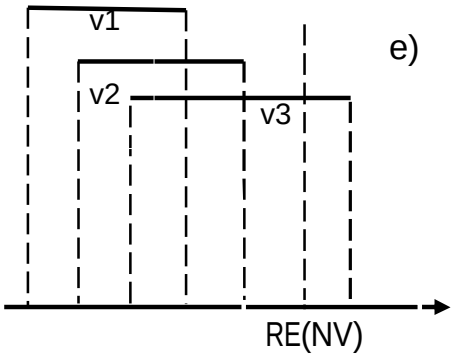
Правило 5: Ако имаме варианта *i,o,o* и само единият от *v2* и *v3* е съседен на *v1*, то тоя връх трябва да бъде *v2.* Тоест, ако *v3* е съседен на *v1, а v2* не е, то върховете във *v2* и *v3* се разместват.

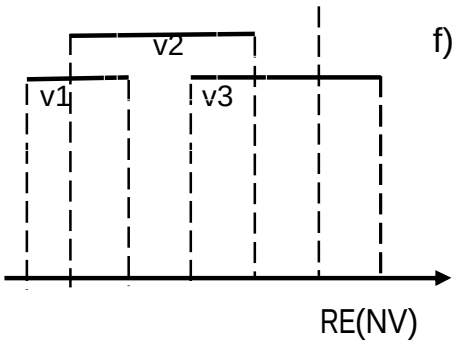

v1
v2
v3
a)
RE(NV)

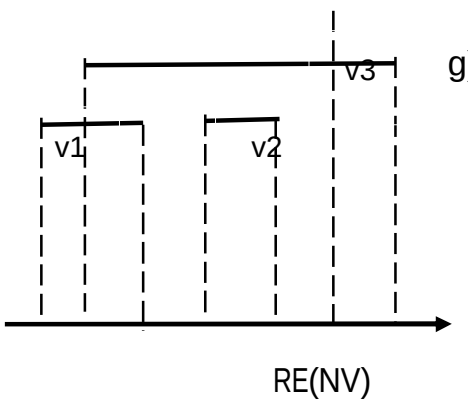

v1
v2
v3
b)
RE(NV)

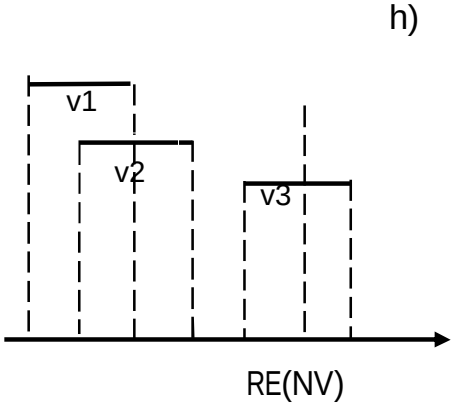

v3
c)
v1
v2
RE(NV)

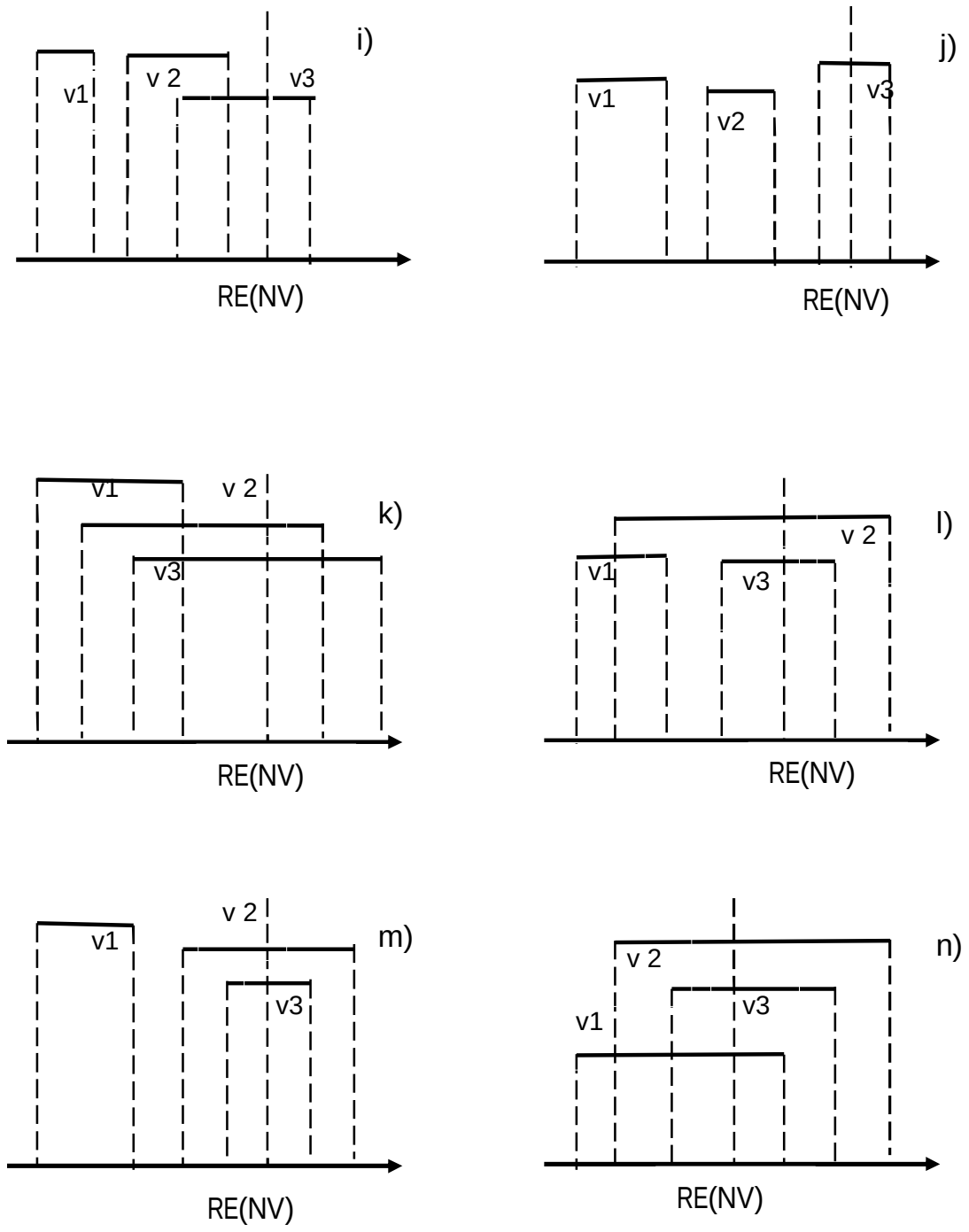

d)
v3
v1
v2
RE(NV)


v1
e)
v2
v3
RE(NV)


v2
f)
v1
v3
RE(NV)


v3
g)
v1
v2
RE(NV)


h)
v1
v2
v3
RE(NV)



Фигура 12 – показани са всичките 14 допустими възможни комбинации на граф с 3 върха по отношение на десния край на връх *NV*

Петте правила, приложени едно след друго намаляват броя на обработваните случаи, а самите случаи покриват всички възможни допустими конфигурации от 3 върха в множествата *SIV* и *SOV*.

На фигура 12 са изобразени 14 случая от всички възможни отношения между върховете, като те могат да бъдат обособени в 4 групи, съответстващи на вариантите от Правило 1. За всеки от примерите е даден и десният край на интервала на най-близкия връх *RE(NV).* Така всеки външен връх има десен интервален край вдясно от *RE(NV),* а всеки вътрешен вляво от него. При първите 4 случая - “a-d” - няма външен връх (т.е. *i,i,i)* при вторите 6 , “e-j” има един външен връх (т.е. *i,i,o)* докато третият вариант *i,o,o* обхваща само 3 случая ”k-m”. По-малкият брой се дължи на Правило 1 от раздел 2.4. Същото правило е причина последният вариант *o,o,o* да има само един случай, последният ”n”. Но защо са избрани точно 14 - ако се абстрахираме от пермутациите **разглежданите 14 случая са всички възможни комбинации по отношение на съседство между върховете. Подробното доказателство е дадено в пълната версия на дисератционния труд.**

2.9 Допълнителни пояснения по т.6 за num_SCV(NV) > 3

Когато общия брой на върховете от *SIV* и *SOV* е повече от три възможните случаи стават много и подходът описан в предходния раздел не работи. Решението е рекурсия, т.е. да се построи и подреди нов граф, където всеки негов връх съответства на връх от *SIV* или *SOV*. Два върха от новия граф са съседи **тогава и само тогава, когато съответните върхове от първоначалния граф са съседи**. За новия граф се използва основният алгоритъм от настоящата статия, но с допълнителни изисквания, които ще се опишат сега:

- В общия случай новообразуваният граф може да се състои от няколко несвързани подграфи. Това е важна особеност. Докато

началният граф е винаги свързан и може да се превърне в несвързани подграфи когато имаме пълни върхове то за подграфите няма никакво ограничение и може да има несвързани подграфи от самото начало. Заради Правило 1 само един от тях може да има върхове от *SOV*. Обработката на новия граф трябва да започне именно от тоя първи подграф - по този начин интервалните краища на върховете от подграфа ще бъдат най-вляво или най-вдясно в ИМ на графа. Когато се прави мепинг (трансформация) на координатите от ИМ на новообразуваният граф към координатите от ИМ на първоначалния граф десните интервални краища на върховете на новия граф, съответстващи на тия от *SOV* не участват.

- Ако за новия граф в даден момент се получи, че общия брой на върховете от *SIV* и *SOV* е повече от 3 следва нова рекурсия, както описаната до тук и така докато броят на върховете падне до 3 или по-малко и се наредят веднага.

Заради рекурсивното извикване към входните данни на програмата трябва да се добави кои върхове участват в мепинга заедно с информация кой от върховете е външен за викащата функция.

Важно е да се подчертае, че в точка 8 на *ПодредиЕдинПодграф* при добавянето на *SOV* към *SAV* Правило 1 продължава да бъде в сила, т.е. върховете в *SAV* отново образуват клика заради Правило 2 и фактът, че върховете в *SOV* също образуват клика.

2.10 Сертификация на решението, че един граф е ИГ

Сертифициращ алгоритъм (СА) за даден проблем е такъв алгоритъм, който снабдява със сертификат всеки отговор, който произвежда. Сертификатът е надеждно свидетелство, доказващо, че отговорът не е бил повлиян от грешка в имплементацията. Естествено изискване е един СА да бъде с линейна сложност за бързодействие. Освен формалното доказателство за линейност такова може да бъде и

проверката за големи графи където в експеримента нарастват подходящо както броя на върховете, така и тоя на дъгите. Последната точка от основния алгоритъм е посветена именно на сертификацията. Тя не е формално задължителна, но е от огромно значение, тъй като доказва, че ОА е верен.

Струва си да се отбележи, че всяко от трите правила, разгледани досега, в случай, че сработят по същество също е сертификат, че графът не е интервален. Това е така, защото е дадено формално и ясно доказателство за резултата от неговото действие. Тоест, когато всяко от правилата, приложени по време на изграждане на ИП „отсява" неинтервалните графи накрая ще имаме едно интервално представяне.

По време на изпълнение на алгоритъма също така се строи и едно ИП т.е. всички върхове в него образуват интервален граф. **Процесът на генериране на граф въз основа на същото ИП е напълно независим от получаването му.** Ако новогенерираният граф, който е интервален по дефиниция е напълно еднакъв с входния, то следва, че и входният граф е интервален. И обратното, ако двата графа са различни **това е сигурно доказателство за грешка в имплементацията**..

Основните стъпки на сертифициращия алгоритъм за ОА от настоящия труд са само две:

- Генерирай граф въз основа на интервалното представяне;
- Сравни входния и новогенерирания граф и върни решение „съвпадат"/"не съвпадат".

  Детайлите могат да бъдат разучени в пълната версия на дисертационния труд.

2.11 Възможности за паралелизация

Предложеният алгоритъм открива широки възможности за паралелизация, като в настоящия раздел ще се изброят основните от тях:

- След като е получено първоначалното *SAV* обработката в двете посоки може да е едновременна. С използването на споделена памет, където се записват интервалите на обработените до момента върхове от двете нишки може да се реши и по-рано в сравнение с класическата еднонишкова имплементация, че графът не е интервален - **когато се получи връх, който е обработен в двете посоки**.
- Когато има подграф с брой върхове над определен праг рекурсивното намиране на неговите интервали може да бъде в отделна нишка. Същата нишка ще изпрати съобщение, съдържащо интервалния модел, когато е готова към съответната от двете главни нишки.
- За големи графи когато и множеството *SOV* е голямо намирането на *SCV* може да бъде направено в няколко нишки.
- За първоначалния граф съседите на всеки връх са в списък. Не се препоръчва използването на матрица на съседство вместо списък защото при големи графи обемът на данните става огромен. Но ако списъкът се подреди следващата обработка съществено се ускорява. Например, проверката дали два върха са съседни вместо последователно търсене в списъка се свежда до двоично търсене. При многоядрени процесори списъците може да бъдат сортирани в паралел.
- Проверката, която трябва да се извърши след намирането на *SV,* именно дали даден връх е външен или вътрешен и от коя страна е трябва да се направи за всички необработени съседи на *SV*. Ако броят на тия съседи е над определен праг отново може да се използват отделни нишки или ядра за по-бързо изчисление.

2.12 Заключение

В сегашният си вид разгледаният алгоритъм дава сертификат за коректност само ако графът е разпознат като интервален, но не дава сертификат във вид на АТ за хорден не ИГ или сертификат във вид на върхове, образуващи цикъл без хорди, по-голям от 3 за нехорден граф. Беше доказано също така, че Правило 3 сработва само за нехордни графи. Начинът на работа на алгоритъма не предполага намирането на АТ в общия случай, но **сработването на някое от трите правила имплицитно показва за наличието на поне една такава**. Обратното, неприлагагането на нито едно от правилата позволява построяването на едно ИП, което е и доказателство (сертификат), че графът е интервален.

Създаването на ефикасни паралелни алгоритми за разпознаване на ИГ все още е открит проблем. За разлика от тях в предложения тук алгоритъм принципно има поле за паралелизация – едновременната обработка в двете посоки, а също и отделни нишки за рекурсивното извикване на алгоритъма за редене на подграфи. Това е така, защото в момента на рекурсията не се знае какви са отношенията вътре в подграфа, но се знае размерът и мястото на интервала, който подграфът ще заеме в създавания ИМ. Следователно имплементацията на разновидност на алгоритъма за паралелно подреждане е също обещаващо направление за бъдеща работа. Вместо правило 3 при паралелизация е по-удобно да се прилага друго правило за всички върхове, които не принадлежат на първоначалното *SAV*: ако даден връх е обработен и в двете посоки, това означава, че графът не е интервален.

Глава 3. Софтуерна реализация на алгоритъма

Имплементацията на алгоритъма до завършена и тествана срещу грешки програма не е тривиален проблем. От началния граф, зададен като файл, състоящ се от списък на върховете и дъгите между тях, до получаване на интервалното представяне на същият граф има дълъг процес от избор на основните типове данни, дизайн на програмата, организация на рекурсията, основна за новия алгоритъм и преодоляване на всички подводни камъни, които биха повлияли на линейният характер на алгоритъма. Имплементацията е извършена на C++, поради което в някои от примерите в настоящата глава неявно се подразбира синтаксисът на езика.

Част от помощните алгоритми за софтуерната реализация са вече разгледани в предходната глава. Пак там са въведени някои означения и понятия, които ще се използват и сега, например множеството *SAV*, където първоначално се намират външните съседи от едната страна на началния връх *SV* в едно интервално представяне.

3.1 Използвани структури

Интервалното представяне се състои от цели положителни числа започващи от 1 така, че за граф с *n* върха интервалните краища са в обхвата *[1 ÷ 2*n]*. Задаването на интервалното представяне може да се реализира по два принципни начина:

- Или за всеки връх интервалните краища се задават посредством структурата

```
typedef struct interval {
      ulong_t  left;
      ulong_t  right;
} interval_t;
```

където *ulong_t* е типът *unsigned long int.*

- Или самото интервално представяне е масив от *2*n* елемента от типа:

```
typedef struct intervalends {
      ulong_t  vert;
      unsigned char  l_or_r;
} intervalends_t;
```

където *vert* съдържа номера на върха, а *l_or_r* може да приема само две стойности *LEFT_END* и *RIGHT_END*. Това е същата структура, която се използва и при сертификацията на графа.

В програмата са използвани и двата начина като връзката между тях е следната: Ако *m_iga[]* е масивът от върховете от типа *struct interval*, а *m_intervalEnds[]* е масивът от типа *struct intervalends_t* и е дадено: *m_intervalEnds[k].vert = x; m_intervalEnds[k].l_or_r = RIGHT_END,* то е в сила *m_iga[x].right = k*. Инициализацията на полетата *left* и *right* е с нули, което показва, че за даден връх интервалните краища още не са намерени.

Основната структура, съдържаща атрибути за всеки връх е следната:

```
typedef struct attributes {
      unsigned char putInSAV;
      bool initialForSAV;
      unsigned char inside;
      unsigned char checkVert;
      bool processed;
      ulong_t  left;
      ulong_t  right;
      ulong_t  numOfAdjInSAV;
      ulong_t  numOfInputAdj;
} attributes_t;
```

Елементите на структурата имат следния смисъл:

- Двата елемента *left* и *right* са вече разгледани и са част както на основната структура, така и на по-малката структура *struct interval* в някои от подпрограмите.

- *putInSAV* - Може да приема само три стойности - *NOT_SAV_VERT*, *SAV_VERT* и *LEFT_FROM_SV*, втората от която означава, че съответният връх е член в даденият момент на множеството *SAV*. Първата стойност означава точно обратното, че не е член. Последната стойност е за специални нечленове на *SAV* и е разгледана в основното Правило 3 в глава втора.
- *initialForSAV* – има стойност *true* само за първоначалните съседи на *SV* в първата посока. Същите върхове се нуждаят от по-специална обработка в сравнение с останалите от *SAV*.
- *inside* – атрибут, разграничаващ вътрешните и външните върхове спрямо текущият *NV* или *SV*. Съответно стойността на атрибута е *INNER_VERT* или *OUTER_VERT*.
- *checkVert* – атрибут с който се проверява дали един връх е външен или вътрешен спрямо друг в *Помощен алгоритъм 1* от втора глава.
- *processed* – атрибут, с който се означава дали интервалът на даден връх е вече намерен, т.е. върхът е вече обработен. Тоя атрибут е от особено значение, защото съществуват принципиално два подхода към вече обработваните върхове – или да се маркират с *processed = true* или такъв връх да се премахне от самия граф. В имлементацията, свързана с представения тука труд е предпочетен първият вариант като по-бърз и по-удобен. При втория вариант всеки път би се налагало да се преизчисляват върховете и дъгите в текущия граф.
- *numOfAdjInSAV* – Когато се търси *NV*, изхождайки от предпоставката, че текущият граф може да е интервален, това става като се преброят съседите на всеки връх в *SAV*. При това самите върхове от *SAV* не участват в броенето, както и върхове, които имат интервали преди левият интервал на текущият *NV*. Същото е в сила и за върховете имащи интервали преди левият интервал на *SV* при намирането на първоначалните членове на *SAV*.

- *numOfInputAdj* – броят на съседите на текущия връх във входния граф. Този елемент не се променя в алгоритъма, за разлика от описания по-надолу *m_numRealAdj.*

Следващата структура, се използва при рекурсивното викане на основния алгоритъм за подграфи, имащи повече от 3 върха:

```
typedef struct mapping_t{
            ulong_t  vertex;
            unsigned char inside;
       } mapping_t;
```

Елементът *vertex* има за стойност връх от текущия граф, който ще участва в образуването на подграфа, чиято обработка ще се осъществи чрез рекурсията. Вторият елемент *inside* има същия смисъл, както и в основната структура защото трябва да се знае кой от върховете е външен за викащата функция – такива върхове ще се поставят първи в *SAV*.

3.2 Основен клас – конструктор и членове

Основният клас *CPanGraph* има един конструктор със следните параметри:

CPanGraph(ulong_t numVert,
ulong_t numOfOuterVert,
*ulong_t** numRealAdj,*
*ulong_t*** vert,*
*interval_t** intervals,*
*intervalends_t **intervalEnds,*
ulong_t numOfSubgraphs,*
mapping_t mapping,*
*ulong_t *resOK);*

Параметрите са избрани така, че да позволяват използването на рекурсия, съгласно основната идея за обработка на алгоритъма,

разгледан в предишната глава. Типът *ulong_t* е *unsigned long int.* Допълнителна информация за параметрите:

numVert - входен, размер на входния граф, Съответства на ***NGV*** от втора глава;

vert – входен, отново от *readGraphFile,* разгледан подробно в 3.1;

numRealAdj – входен, също от *readGraphFile,* разгледан в 3.1;

resOK – изходен. Стойност 0 означава, че е намерено интервално представяне на входния граф. Останалите възможни стойности са AT_TYPE_UNKNOWN1, AT_TYPE_UNKNOWN2 и AT_TYPE_UNKNOWN3, ако са сработили съответно Правило 1, Правило 2, или Правило 3 ;

intervals – изходен, намерените интервали са в структура *interval_t*;

intervalEnds - изходен, намерените интервали са в структура *intervalends_t*; *И*ма смисъл (както и *intervals),* само ако *resOK е 0;*

mapping – входен. Има смисъл само при рекурсия. Указва кои върхове от текущия граф участват в подграфа, който ще се обработва при рекурсивното извикване на алгоритъма. При първоначалното извикване на конструктора за входния граф има винаги стойност *NULL*;

numOfOuterVert - входен. Има смисъл само при рекурсия. При първоначалното извикване на конструктора за входния граф винаги има стойност *0*. Дава броя на външните върхове спрямо текущия най-близък връх на графа на викащия обект, осъществяващ рекурсията;

numOfSubgraphs – изходен. Дава броят на несвързаните подграфи, от които се състои текущия граф. Тъй като входният граф по условие е свързан, за него стойността е винаги 1. Но при рекурсивното извикване броят на несвързаните подграфи може да бъде теоретично произволно голям. Съгласно Правило 1 само един от подграфите обаче може да има върхове, които са външни спрямо текущия най-близък връх на графа на викащия обект, осъществяващ рекурсията;

Следва описание на всички членове на класа *CpanGraph:*

ulong_t m_numVert – брой на върховете на текущия граф. Получава стойност от параметъра *numVert* на конструктора на класа;

ulong_t m_SV – началният връх, посочен като *SV* във втора глава. Пак там е описан и начинът на получаване му;

ulong_t SAV* – множество на активните върхове. Реализацията му е като едномерен масив с размер *m_numVert,* което подсигурява, че елементите поставени в него няма да надхвърлят границите му;

ulong_t m_indSAV – текущ брой на елементите в *SAV*;

*ulong_t** m_vert* – двумерен масив. Получава стойностите си от параметъра *vert* на конструктора на класа, без да ги променя, т.е. тука се съхранява входният граф в удобен за работа вид;

ulong_t m_numRealAdj* – получава стойностите си от параметъра *numRealAdj* на конструктора на класа. В началото се явява броя на елементите за всеки ред на *m_vert* ;

interval_t m_intervals* – изчислените интервали съхранявани в тоя едномерен масив се предават на изходния параметър *intervals* на конструктора;

intervalends_t m_intervalEnds* – изчислените интервали съхранявани в тоя едномерен масив като второ представяне, алтернативно на това в *m_intervals* се предават на изходния параметър *intervalEnds* на конструктора;

attributes_t m_iga* – основният работен масив за програмата. Атрибутите, които могат да се присвоят на всеки от върховете в графа са вече описани при представянето на структурата *attributes_t*;

ulong_t m_sizeSubgraph – брой на върховете в текущо-обработвания подграф;

ulong_t m_upper – помощна променлива за изчисляване на ИК на върховете, описана подробно в глава втора под името *upper*;

ulong_t m_lower – с подобно предназначение като *m_upper*;

ulong_t NPV – брой на обработените върхове, описан в глава втора;

ulong_t m_numOfSubgraphs – съдържа броя на несвързаните подграфи, от които се състои текущият граф. Параметърът *numOfSubgraphs* на конструктора получава стойността си от него;

ulong_t num_SOV – брой на външните върхове спрямо текущия най-близък връх *(NV*);

ulong_t num_SIV – брой на вътрешните върхове спрямо текущия най-близък връх *(NV*);

ulong_t num_SNV – брой на най-близките върхове. Показва колко други върха имат същите множества *SIV* и *SOV* в даден момент от изпълнение на програмата. Вече разгледан в глава втора;

ulong_t m_sizeOfArrayOuterVert – размер на масива, реализиращ множеството *SOV.* Когато имаме *num_SOV* > *m_sizeOfArrayOuterVert* избраното като реализация решение е масивът да нарасне два пъти чрез извикването на функцията *realloc*;

bool m_firstSubgraphHasOuterVert – в случай на рекурсивно извикване на програмата дава информация дали текущият граф има върхове, които са външни спрямо *NV* от по-горното ниво;

3.3 Особености на обработката на върховете в SAV

Една от основните стъпки в *ПодредиЕдинПодграф* е обработката на върховете в *SAV* след като са наредени съседите на най-близкия връх. За целта е написан метода *processSAV*. Неговите главни задачи са да премахне от *SAV* вече наредените върхове, да постави нови, ако има такива и да намери следващия *NV* от множеството *SNV.* Елементът *numOfAdjInSAV* (съответства на *num_SCV* от глава 2) от основната структура е тоя, който определя решението за *NV* и затова в настоящия раздел ще се разгледа подробно как се изчислява неговата стойност във всеки от трите варианти които съществуват.

Методът *FindNumOfAdjForOneVertInSAV* реализира два от споменатите варианти. Първият е само за първоначалните елементи в *SAV* и то само за първата посока. Разгледан е в началото на раздел 2.4.3. и чрез него се пресмятат броя на върховете, чиито интервали ще са вдясно от *LE(SV)*. Разграничението между първите върхове в *SAV* в

първа посока и останалите става с елемента *initialForSAV* от основната структура *attributes_t*.

За първоначалните върхове във втората посока, както и за първите върхове поставени в *SAV* при рекурсия се прилага вторият вариант за пресмятане на *numOfAdjInSAV* - намира се броят на необработените съседи на даден връх, които не са част от *SAV*. В случай на рекурсия в началото всички върхове са необработени и всички са съседни на текущия *NV* от викащия конструктор. Един тънък момент е, че се подсигурява *initialForSAV* да има стойност *false* във втора посока и при рекурсия.

Третият вариант за пресмятане на *numOfAdjInSAV* се извършва в метода *FindNumOfAdjForNewVertInSAV* и е за върховете, които се добавят в *SAV* - това са върховете от текущото *SOV*. Методът работи по начина описан в точка 8 от главния алгоритъм за намирането на *num_SCV* в раздел 2.5. Броят се върхове, които не са съседи на текущия *NV.* Сега вече могат да се посочат основните стъпки за метода *processSAV*:

1. Всички върхове от *SNV* се преместват на края на масива SAV. Прави се по две съображения:

- първо, за всички върхове, които са преди тях много лесно се пресмята *numOfAdjInSAV = numOfAdjInSAV - (num_SIV + num_SOV):*

- второ, удобно може да се добавят новите членове на *SAV* в точка 3 на мястото на върховете от *SNV.* Самото преместване е линейна операция като бързодействие.

2. За всички от множеството *SNV* се пресмята *RE* и се задава *putInSAV = NOT_SAV_VERT;*

3. Добавят се нови членове към *SAV*, това са върховете от *SOV*. Съществуват 2 подслучая:

а) ако *num_SOV* <= *num_SNV* - всички външни върхове се слагат на мястото на тези от *SNV;*

б) ако *num_SOV* > *num_SNV* последните (*num_SOV* - *num_SNV*) върха се добавят като нови елементи на *SAV*.

4. За всички новодобавени се извиква *FindNumOfAdjForNewVertInSAV* за пресмятане на *numOfAdjInSAV.* За всички стари върхове в *SAV* се изчислява *numOfAdjInSAV* по начина описан в стъпка 1. И в двата случая след пресмятане на новото *numOfAdjInSAV* се проверява дали то не е минимално от всички изчислени и ако е така, то става минимум. Ако текущото е равно на временния минимум се инкрементира друга променлива, която на края на метода *processSAV* се присвоява на *num_SNV.* Казано с други думи се намира следващия *NV* и колко са най-близките върхове.

В заключение за метода *processSAV* е добре да се споменат две особености:

- Новата дължина на масива *SAV* винаги се пресмята така:
  m_indSAV = m_indSAV+num_SOV- num_SNV;
- Понеже всички изчисления в метода са линейни самият метод е линеен като степен на бързодействие;

3.4 Експериментални резултати

Целта е да се покаже, че имплементацията на описания алгоритъм наистина води до успешното и бързо разпознаване на един граф като интервален. Тестовете бяха извършени върху над 4 хиляди графа с размерност от 4 върха (минималният брой върхове, когато един свързан граф може да не е интервален) до графи със 100 000 върха. За нуждите на експериментите беше написана и допълнителна програма за генерирането за интервални графи въз основа на алгоритъма от предишния раздел. От тях чрез добавяне на допълнителни дъги между двойки върхове лесно се получават графи, които не са хордни или са хордни, но не са интервални. Другият начин на получаване на неинтервални графи е да се премахне дъга между два върха, които

участват в клика от Правило 1. Всички тестови ИГ правилно бяха сертифицирани като такива и не се откри граф, който алгоритъмът да разпознае като интервален, но да не се сертифицира. Самият факт обаче, че алгоритъмът принадлежи към класа на така наречените „сертифициращи алгоритми" се оказа голямо предимство за бързото откриване на грешки в имплементацията.

При големи графи десеткратното/стократното увеличаване както на броя на върховете, така и на броя на дъгите доведе и до също около десеткратното/стократното увеличаване на времето за обработка (виж таблица 3). Така линейният характер на алгоритъма беше потвърден и експериментално като неговата времева сложност е $O(n+m)$. **Трябва задължително да се направи уговорката, че се наблюдава линейност само ако плътността на графа (т.е. средният брой съседи на един връх) не се променя**.

Таблица 3 – време за разпознаване на един граф при различен брой върхове и ръбове

Брой върхове n	Брой ръбове m (закръглени до 1000)	Време за разпознаване в сек.	Среден брой съседи на един връх
1000	100000	0.12	200
10000	1000000	1.2	200
100000	10000000	13	200
1000	48000	0.034	96
1000	210000	0.64	420

От друга страна при двукратно увеличение (намаление) на средния брой на съседите на един връх времето за обработка се увеличава (намалява) нелинейно, за тестовете от таблица 3 около 4-5 пъти. Подобна нелинейност се наблюдава и при намиране на PEO чрез LBFS и оттам за другите методи за разпознаване на ИГ. Това е лесно обяснимо

защото времето за изчисляването на клики (използвани при Правило 1) има сложност $O(m^2/2)$. В случая с m е означена плътността на графа. При клики трябва да проверим дали всеки връх е съсед на останалите в кликата. Същото е в сила и за други обработки срещащи се както в новия алгоритъм, така и в другите известни методи, оттам и наблюдаваната нелинейност.

За най-големия граф със 100 000 върха и общ брой на дъгите малко над 10 000 000 времето за разпознаване (с точност до секунда) е 13 секунди, а времето за сертификация е 4 секунди. За сравнение, времето за четене на върховете и дъгите между тях от файл за същия граф заедно с приготвянето на масива от списъци със съседите на всеки връх отне 24 секунди. Получените резултати са при имплементация на C++ на лаптоп с процесор 2.4 Гх и RAM памет 4 ГБ без никакви специални оптимизации на кода за бързина. Наличието на ефикасен и лесно конфигурируем лог дава възможност за всеки граф да се проследят стъпките при изграждането на неговото интервално представяне както и да се наблюдават мястото и условията за сработване на някое от трите правила при неинтервален граф.

Приноси

1. Направен е анализ на съществуващите основни методи за разпознаване на ИГ като са посочени характеристиките и недостатъците на всеки от тях.
2. Разработен е нов алгоритъм за разпознаване на интервални графи посредством построяване на ИП без използване на LBFS.
3. Извлечени са три правила от топологичните свойства на ИП, като нарушението на което и да е от тях класифицира входния граф като неинтервален.
4. Изведени са формални доказателства за валидността на всяко от предложените правила.
5. За всеки интервален граф се връща едно ИП, като е подсигурено интервалните краища да са цели положителни числа.
6. Доказани са възможностите за паралелизация на алгоритъма като са посочени основните стъпки, позволяващи многонишковата му имплементация, което в пъти да увеличава бързодействието му;
7. Доказано е, че полученото интервално представяне е сертификат, което дава основание новият метод да бъде причислен към категорията на сертифициращите алгоритми.
8. Разработено е програмното осигуряване на C++, имплементиращо новия алгоритъм. Обсъдени са тънкостите и детайлите в процеса на реализацията му. Програмното осигуряване е тествано върху над 4000 графи.
9. Анализирани са отделните стъпки на алгоритъма от гледна точка на бързодействието и на основата на резултатите от тестването му е доказано, че той има линейна сложност.

Публикации по дисертационния труд

Статии

1. Ст. Панов, Нов алгоритъм за разпознаване на интервални графи посредством група от правила, наложени върху интервалния модел, Computer & Communications Engineering, vol. 8 No 1 (2014) 37-43, ISSN 1314-2291.
2. Ст. Панов, В. Николов, Генерация и сертифициране на интервални и хордни графи, Computer & Communications Engineering, vol. 8 No 2 (2014) 34-38, ISSN 1314-2291.

Доклади на научни конференции, публикувани в пълен текст

1. S. Panov, Efficient Interval Representation Via A New Algorithm For Recognition Of Interval Graphs, Challenges in Higher Education & Research, vol. 12 eds. T. Tashev, R. Deliyski, B. Lepadatescu, Heron Press, Sofia, 2014.