Міністерство освіти і науки України

Національний університет “Києво-Могилянська академія”

Факультет інформатики

Кафедра математики

Магістерська робота

освітній ступінь — магістр

на тему: **“Опуклі структури на графах”**

Виконав: студент 2-го року навчання,
освітньо-наукової програми
“Прикладна математика”, 113
Гапоненко Владислав Олександрович

Керівник Козеренко С.О.,
Кандидат фіз.-мат. наук

Магістерська робота захищена
з оцінкою _____

Секретар ЕК _____
“_____”_____ 20__р.

Київ – 20__

Національний університет “Києво-Могилянська академія”

Кафедра математики факультету інформатики

ЗАТВЕРДЖУЮ
Зав. кафедри математики,
доц., к.ф.-м.н.
_____ Р.К. Чорней
(підпис)
“____”_____ 20__р. р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на магістерську роботу
студенту 2-го курсу факультету інформатики
Гапоненко Владиславу Олександровичу

Тема: Опуклі структури на графах.
Зміст магістерської роботи:

1 Вступ
2 Основні поняття
3 Геодетична опуклість
  3.1 Висновки
4 Digital опуклість
  4.1 Висновки
5 All-path опуклість
  5.1 Висновки
6 Класичні задачі
  6.1 Алгоритм перевірки на ap-опклість
  6.2 Алгоритм знаходження $I(S)$, $H(S)$ та $gin(G)$
  6.3 Алгоритм знаходження $c(G)$
  6.4 Алгоритм знаходження $i(G)$ та $h(G)$
  6.5 Висновки
7 Висновки
8 Література

Дата видачі “____”_____ 20__ р. Керівник _____
(підпис)
Завдання отримав _____
(підпис)

Тема: Опуклі структури на графах.

Календарний план виконання роботи:

Номер	Назва етапу магістерської	Термін виконання етапу	Примітка
1.	Отримання теми магістерської роботи.	01.08.2022	
2.	Ознайомлення з темою магістерської роботи.	01.09.2022	
3.	Розробка плану та структури роботи.	15.09.2022	
4.	Робота з науковою літературою, та отримання основних теоретичних результатів.	15.10.2022	
5.	Програмування алгоритмів розв'язку класичних задач в ар-опуклості.	01.01.2023	
6.	Робота над текстовим оформленням результатів.	01.04.2023	
7.	Попередній аналіз магістерської роботи. Виправлення помилок.	01.06.2023	
8.	Захист магістерської роботи.	13.06.2023	

Зміст

Вступ

Класичні концепції теорії опуклих структур отримали багато уваги в першій половині 20-го століття. Проте, з розвитком теорії, поняття опуклості почали розглядати в більш широкому сенсі, не прив'язуючись до векторних просторів. В нашій роботі ми спираємося саме на розширене поняття опуклості, базуючись на аксіоматичному підході представленому Ван дер Велем у книзі "Theory of convex structure"[20], оскільки нашим об'єктом дослідження є зв'язні прості графи. Найбільш дослідженими опуклостями на графах є монофонічна опуклість [5] [6] та геодетична опуклість [7] [8]. Загалом існує безліч опуклих структур на графах і в контексті кожної з них, досліджується набір класичних задач як знаходження алгоритму детермінації опуклості множини, знаходження опуклої оболонки множини, тощо. Нетривіальною є задача поширення інфекції [1], де вершини графа виступають у ролі потенційних реципієнтів інфекції, а сама інфекція поширюється в межах опуклих оболонок множин, що містять заражені вершини. Мета цієї задачи може формулюватися по-різному, єю може бути знаходження такої початкової конфігурації заражених вершин, щоб якнайшвидше "заразити" весь граф, або визначення часу ураження всього графу G за умови зараження певної його підмножини.

Метою роботи є знаходження нових критеріїв all-path опуклих множин та визначення алгоритмів розв'язку класичних задач із теорії опуклості. Зазначена мета встановлює такі завдання:

- Дослідити статті, що пов'язані із опуклостями на графах.
- Сформулювати та довести новий критерій all-path опуклих множин.
- Реалізувати алгоритми розв'язків класичних задач із теорії опуклості.
- Перевірити оптимальність запропонованих алгоритмів.

Об'єктом дослідження є опуклості на графах.

Методами дослідження є аналіз наукових робіт.

У ході виконання роботи було доведено дві нові характеристики ap-опуклих множин та деякі пов'язані з ними теоретичні результати.

Також, в роботі представлено новий алгоритм перевірки підмножини графа на ap-опуклість.

Магістерська робота пройшла апробацію на XI Всеукраїнській науковій конференції молодих математиків.

Основні поняття

В даній роботі ми досліджуємо поняття опуклості саме на графах. Графом G ми називаємо таку пару множин (V, E), що V є скінченною множиною елементів, які ми кличемо вершинами, а елементами E, або ребрами, є пари (a, b) елементів з V. Ми розглядатимемо скінченні прості ненапрямлені зв'язні графи графи. Тобто, V та E є скінченними; якщо $(a, b) \in E$ то таке ребро не є петлею $a \neq b$, ребра є ненапрямленими $(a, b) = (b, a)$ та немає кратних ребер. Граф G є зв'язним тоді і тільки тоді, коли між будь-якою парою вершин графа існує шлях. *Шляхом* $P(a, b)$ між вершинами $a, b \in V$ графа G є скінченна впорядкована послідовність вершин $P(a, b) = a, a_1, ..., a_i, a_{i+1}, ..., a_n, b$, що для всіх a_i, a_{i+1} існує $(a_i, a_{i+1}) \in E$. Також, P називають *простим шляхом* якщо для будь-яких $a_i, a_j \in P$ з $a_i = a_j$ слідує $i = j$.

Цього достатньо для означення *опуклості*. Сім'я підмножин C множини X називається опуклістю на X якщо наступні аксіоми правдиві:

1. Порожня множина $\emptyset$ та сам X входять в C.
2. C закрита відносно перетинів.
3. C закрита відносно вкладених об'єднань.

Тоді пару (X, C) називають *опуклим простором*, а елементи C – опуклими множинами.

Графовим опуклим простором є пара (G, C), де G є зв'язним графом з множиною вершин V, яка в парі з є опуклим простором, що задовольняє попередні аксіоми.

Опуклою оболонкою $H(S)$ підмножини $S \subseteq V$ називають найменшу опуклу множину, що містить S. Для графа G *ступенем опуклості* (*convexity number*) $c(G)$ називають найбільшу потужність нетривіальної опуклої $S \subset V$. Потужність найменшої такої $S \subseteq V$, що $H(S) = V$ є *ступенем оболонки* (*hull number*) і позначається як $h(G)$. Вершину x, що належить опуклій множині A називають *екстремальною* для A, якщо множина $A \backslash \{x\}$ є також опуклою. Граф G називають *опуклою геометрією* відносно певної опуклості, якщо будь-яка опукла підмножина є опуклою оболонкою своїх екстремальних вершин.

Покладемо на множині X функцію $I : X \times X \to 2^X$, що має наступні властивості:

1. Для будь-яких двох елементів $a, b \in X$ справджується $a, b \in I(a, b)$.

2. Функція є симетричною $I(a, b) = I(b, a)$.

Тоді I називають *інтервальним оператором* на X, а $I(a, b)$ є *інтервалом* між a та b. Пару (X, I) називають *інтервальним простором* і він природнім чином породжує опуклість C на X. Так, підмножину C ми називаємо *опуклою* тільки тоді, коли $I(x, y) \subseteq C$ для всіх $a, b \in C$. Для графа G *інтервальним числом* (*interval number*) $i(G)$ називають найменшу потужність такої $S \subseteq V$, що $I(S) = V$. *Числом геодетичних ітерацій* (*geodetic iteration number*) $gin(G)$ для графа G називають найменше i, що $H(S) = I^i(S)$ для всіх $S \subseteq V$.

Геодетична опуклість

Геодетична опуклість є прикладом класичної опуклості, що породжена звичайною метрикою на графах. Так, означимо функцію $d(a,b)$ для $a,b \in V$ як кількість ребер найкоротшого простого шляху, що з'єднує a та b. Тож $d(a,b) = |P(a,b)| - 1$, де $P(a,b)$ найкоротший простий шлях між a та b. Неважко переконатися, що d буде метрикою. Таким чином геодетичну інтервальну функцію для пари вершин x, y можна означити як об'єднання всіх таких простих шляхів $P(x,y)$, що $|P(x,y)| = d(a,b)+1$. Також, оскільки інтервальна функція, що породжує геодетичну опуклість є стандартною, зручно користуватися характеристикою геодетично опуклих множин через геодетичні інтервали: підмножина S графа G є геодетично опуклою, якщо $I[S] = S$.

Нехай $G = (V,E)$ є зв'язним графом та $v, u \in V$. Вершина v називається *граничною вершиною для* u, якщо жоден із сусідів v не має більшої відстані до u ніж сама v. Вершина v є *граничною вершиною графа* G якщо вона є граничною для хоча б однієї $u \in V$. *Границею* $\delta(G)$ *графа* G називають множину всіх граничних вершин для G, тобто $\delta(G) = \{v \in V | \exists u \in V, \forall w \in N(v) : d(u,w) \leq d(u,v)\}$.

Для множини $S \subseteq V$, *ексцентриситетом в* S вершини $u \in S$ є $ecc_S(u) = max\{d(u,v) : v \in S\}$. У випадку якщо $S = V(G)$ тоді кажуть, що розглядається *ексцентриситет* u *в графі* G, тож $ecc_S = ecc_G = ecc(u) = max\{d(u,v) : v \in V\}$. Менш формально, вершина v є *ексцентричною вершиною для* u, якщо вона знаходиться якнайдалі від u порівняно з іншими вершинами графа G, тобто $d(u,v) = ecc(u)$. Вершина v є *ексцентричною вершиною графа* G тоді, коли вона є ексцентричною для якоїсь $u \in V$. *Ексцентриситетом* $Ecc(G)$ *графа* G є множина всіх його ексцентричних вершин; з цього слідує, шо $Ecc(G) = \{v \in V | \exists u \in V, ecc(u) = d(u,v)\}$.

Вершина $v \in V$ називається *периферійною вершиною* G якщо вона має найбільший ексцентриситет в графі G. Ексцентриситет такої вершини називають *діаметром графа* G. Відповідно, *периферією* $Per(G)$ *графа* G є множина всіх його периферійних вершин $Per(G) = \{v \in V | ecc(u) \leq ecc(v), \forall u \in V\}$.

Вершина $v \in V$ називається *контурною вершиною графа* G якщо жодний із сусідів v не має ексцентриситету більшого за $ecc(v)$. Тоді *контуром* $Ct(G)$ *графа* G називають множину всіх його контурних вершин $Ct(G) = \{v \in V | ecc(u) \leq ecc(v), \forall u \in N(v)\}$.

Вершина $v \in V$ називається *симпліціалом графа* G якщо підграф поро-

джений його сусідами $G[N(v)]$ є клікою. Також будь-який симпліціал графа G є *есктремальною вершиною графа* G в геодетичній опуклості. *Екстремальною* $Ext(G)$ *множиною графа* G є множина симліціалів графа G, тобто $Ext(G) = \{v \in V | G[N(v)]$ є клікою$\}$.

Наступне твердження є відомим результатом для геодетично опуклих множин.

Теорема 1. *[3] Якщо у зв'язному графі* G *його зв'язний підграф* $W \subseteq V(G)$ *є геодетично опуклою множиною, тоді* W *є опуклою оболонкою своїх контурних вершин.*

Загалом, контур графа не завжди є геодетично опуклим, наприклад наступна теорема надає достатню умову для опуклості контуру.

Теорема 2. *[2] Якщо у зв'язному графі* G *справедлива рівність* $|Ct(G)| = 2$*, тоді* $Ct(G)$ *є геодетично опуклою*

Більш того, *розширений контур графа* завжди є геодетично опуклим.

Теорема 3. *[2] Розширений контур* $\Omega(G) = Ct(G) \cup Ecc(Ct(G))$ *будь-якого зв'язного графа є геодетично опуклим.*

Наслідок 1. *Границя будь-якого зв'язного графа є геодетично опуклою.*

Також, однією з актуальних задач є визначення степені k інтервальної функції або ж геодетичного замикання $I^k(Ct(G))$ при якому $I^k(Ct(G))$ є геодетично опуклою. Відомий наступний результат.

Теорема 4. *[2] Якщо у зв'язному графі* G *справедливе включення* $Ecc(Ct(G)) \subseteq I[Ct(G)]$*, тоді* $I^2[Ct(G)] = V(G)$*, а* $I[Ct(G)]$ *є геодетично опуклою множиною.*

Для будь-якого зв'язного графа справедлива наступна теорема, що оцінює геодетичне замикання його сонтуру.

Теорема 5. *[2] Нехай* G *є зв'язним графом з діаметром* D *та потужністю ексцентриситета контуру* n*. Тоді справедлива наступна рівність*

$$I^k[Ct(G)] = V,\ k \leq min\{|Ct(G)| - 1, \tfrac{D}{2} + 1, n\}$$

Висновки

У даному розділі була проаналізова геодетичну опуклість: критерій геодетичної опуклості, теоретичні результати пов'язані із геодетичністю контуру та границі графа.

Digital опуклість

В галузі інформаційних технологій ефективність способу зберігання інформації завжди є актуальною задачею. Оскільки більшість класичних структур даних можуть бути представленими як граф, питання оптимального способу зберігання графів є класичною проблемою на перетині дискретної математики та інформаційних технологій. Задача відновлення графа за його підграфом почала досліджуватися в середині 20-го століття. Тоді був отриманий один з найперших результатів цього напрямку: будь-який граф G щонайменше третього порядку, з точністю до ізоморфізму, може бути відновлений з підграфу $G[V(G) \setminus \{u\}]$ для $\forall u \in V$ ([12] та [19]). Також, Харарі вивів схожий результат але для ребер [10].

Digital опуклість (далі буде згадуватись як d-опуклість) була представлена в [16] як інструмент для обробки зображень, адже матриця, якою зазвичай репрезентується зображення, може бути представлена у вигляді графа, де кожна вершина є пікселем, а суміжним пікселям відповідають суміжні вершини. Для означення d-опуклості спершу потрібно ввести наступні означення: *окіл $N_G(v)$ вершини v* позначає множину сусідніх з v вершин, а $N_G[v]$ *є закритим околом*, тобто $N_G[v] = N_G(v) \cup \{v\}$. Якщо буде зрозуміло у контексті якого графа G розглядається окіл, використовуватиметься позначення $N(v)$ або $N[v]$. Для множини S *околом множини* $N[S]$ є об'єднання околів її вершин $N[S] = \cup N_G[v] | v \in S$. Множина вершин S графа G є d-опуклою якщо для $\forall v \in V$ із $N[v] \subseteq N[S]$ випливає, що $v \in S$. Наступна теорема є характеризацією d-опуклих множин.

Теорема 6. *[9] Множина вершин S графа G є d-опуклою тоді і тільки тоді, коли існує така підмножина $W \subseteq V(G)$, що $S = V(G) \setminus N[W]$.*

Зауважимо, що навідміну від геодетичної опусклості, d-опуклість не породжена інтервальною функцією. Наведемо приклад незв'язної d-опуклої множини.

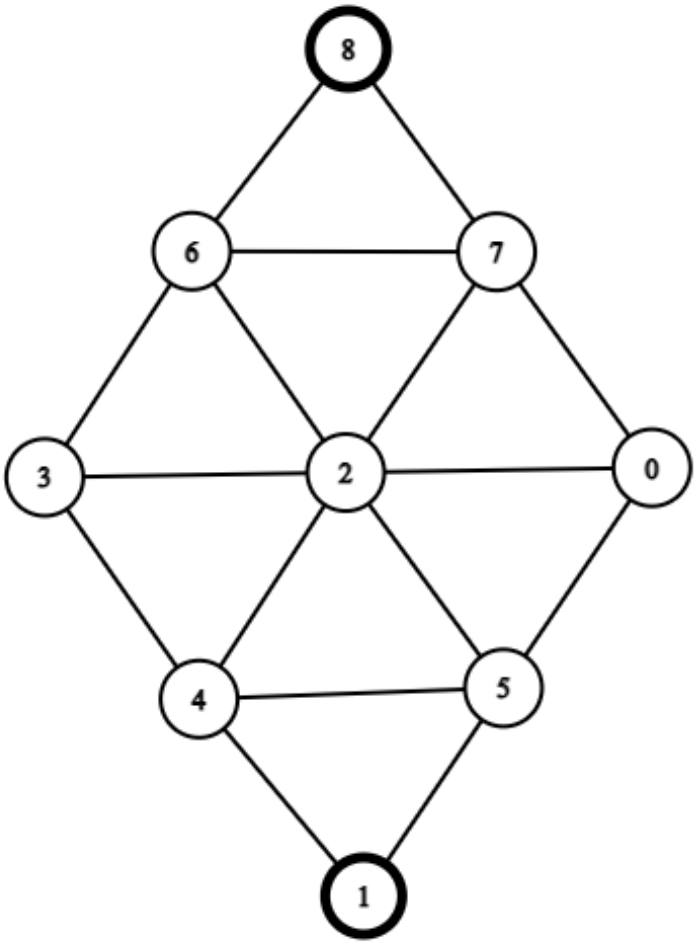


Рис. 1: Граф G з d-опуклою подмножиною $\{1, 8\}$

За допомогою критерія d-опуклості не важко переконатися, що підмножина $\{1, 8\}$ є d-опуклою як доповнення до околу підмножини $\{0, 2, 3\}$.

Для d-опуклого простору використовуватимемо позначення $(G, D(G))$. Наступна теорема описує загальні властивості d-опуклих множин, що випливають із означення.

Теорема 7. *[14]*

- *Якщо S є d-опуклою в деякому графі G, тоді результат дії функції $f(S) = V(G) \setminus N[S]$ є також d-пуклою, а f є бієкцією з $D(G)$ в себе.*
- *Будь-який граф G має парну кількість d-опуклих множин.*
- *Жодна з вершин $v \in V(G)$ не належить більшій половині d-пуклих множин.*

- *Якщо вершина $v \in V(G)$ належить рівно половині d-опуклих множин, тоді v є симпліціалом.*

- *Для дерева G вершина v належить рівно половині d-опуклих множин тоді і тільки тоді, коли v є листком.*

Для наступного результату необхідне ввести додаткові означення. Для вершини $v \in V(G)$ *2-околом вершини* v є $N_2[v] = \{u \in V | d_G(u, v) \leq 2\}$. Нехай G є нетривіальним зв'язним графом. Тоді, для вершини $v \in V$ множина $S \subseteq V$, що не містить v, є *мінімальною локально домінантною* множиною для v якщо $N[v] \subseteq N[S]$ та $\forall u \in S$ справджується $N[v] \nsubseteq N[S \setminus \{u\}]$.

Для дерева G для будь-якої вершини v множина її сусідніх вершин є мінімальною локально домінантною множиною для v. На основі цього факту в [13] доводиться наступна теорема.

Теорема 8. *За даною сім'єю d-опуклих підмножин $D(G)$ дерева G можна однозначно визначити окіл будь-якої вершини графа, тому така $D(G)$ однозначно визначає G.*

Висновки

У даному розділі була проаналізова d-опуклість: її критерій, та поле практичного використання опуклості.

All-path опуклість

All-path опуклість є прикладом опуклості, що породжена інтервальною функцією, де такою для пари вершин x, y будуть всі прості шляхи $P(x, y)$. Для надання характеристики all-path опуклих множин нам спершу треба ввести поняття *блока* та *породженого підграфа*. Зафіксуємо підмножину $S \subseteq V$, тоді породженим підграфом $G[S]$ буде граф вершинами якого є елементи з S, а ребрами, також, будуть пари вершин з S. *Двозв'язним* є граф, що залишається зв'язним після видалення будь-якої вершини, тобто, коли породжений підграф $G[V \setminus a]$ є зв'язним для всіх $a \in V$. Блоком графа G називають максимальний за включенням вершин його двозв'язний підграф.

Представимо новий критерій ap-опуклості множин.

Теорема 9. *Нехай G є зв'язним графом, $A \subset V(G)$ та $|A| \geq 2$. Тоді множина A є AP-опуклою тоді і тільки тоді, коли породжений підграф $G[A]$ є зв'язним об'єднанням блоків в G.*

Доведення. Доведемо достатність від супротивного. Покладемо $G[A] = \bigcup_{k=1}^{m} B_k$ є зв'язним підграфом, де $B_1 ... B_m$ є набором блоків з G. Припустимо, що існують такі вершини $u, v \in A$ із простим шляхом $P_1 = P(u, v)$, що $V(P_1) \nsubseteq A$. Без втрати загальності покладемо $V(P_1) \cap A = \{v, u\}$. Оскільки $G[A]$ є зв'язним, існує $P_2 = P(u, v)$ такий, що $V(P_2) \subseteq A$, але тоді існуватиме двозв'язна множина $S \supseteq P_1 \cup P_2$. Така S має належати $G[A]$ з чого отримали суперечність.

Доведемо необхідність. Нехай A є AP-опуклою множиною. Оскільки G є зв'язним, очевидно, що $G[A]$ також зв'язна. Оскільки $|A| \geq 2$, зафіксуємо пару суміжних вершин $u, v \in A$. Оскільки кожне ребро знаходиться в блоці, існує блок B графа G, що $u, v \in V(B)$. Якщо $|B| = 2$, тоді $G[A] \simeq K_2$, а A відповідно буде AP-опуклою. Розглянемо випадок коли $|B| \geq 3$ та зафіксуємо вершину $w \in V(B) - \{u, v\}$. З означення блока випливає, що ребро uv та w мають лежати на циклі в B. З цього слідує, що існує простий шлях $P = P(u, v) \subseteq B$, що $w \in V(P)$. Оскільки A є AP-опуклою, $V(P) \subseteq A$. Причому це справедливо для всіх вершин з $V(B)$, тому $V(B) \subseteq A$, що доводить твердження. □

Також, очевидно, що об'єднання AP-опуклих множин A, B, таких що $A \cap B \neq \emptyset$, є AP-опуклим за характеризацією AP-опуклих множин. Адже $A \cup B$ є зв'язним об'єднанням зв'язних об'єднань блоків. Проте, таке твердження не вірне для перетинів AP-опуклих множин, адже перетином таких

зв'язних множин може бути або спільний набір блоків графа G, або шарнірна вершина.

Щоби сформулювати другий критерій AP-опуклих множин, введемо посилення одного класичного означення з метричної теорії графів. А саме, нехай $A \subset V(G)$ множина вершин у зв'язному графі G та $x \in V(G)$. *Воротами* для x у A називається вершина $g \in A$ така, що для кожної вершини $a \in A$, деякий найкоротший шлях від x до a містить g.

Якщо в попередньому означенні посилити умову до "будь-який найкоротший шлях від x до a містить g", отримаємо означення *сильних воріт* для x у A.

Лема 1. *Нехай G є зв'язним графом, $A \subset V(G)$ та $|A| \geq 2$ та $x \in V(G)$. Тоді, якщо $g \in A$ є сильними воротами в A для x, це саме справджується для сусідніх з x вершин.*

Доведення. Доведемо від супротивного. Зафіксуємо $y \in V(G)$, що $yx \in E(G)$ та припустимо, що x та y мають різні сильні ворота в A, а саме g_x та g_y відповідно. Тоді має справджуватися рівність $d(x, g_x) = d(y, g_y) = k$. З цього слідує, що найкоротший шлях $P_x = P(x, g_x)$ не містить y та g_y, а $d(x, g_y) > d(x, g_x)$. Справедливим буде дуальне твердження, що найкоротший шлях $P_y = P(y, g_y)$ не містить x та g_x, а $d(y, g_x) > d(y, g_y)$. Тоді існуватиме простий найкоротший шлях $P(x, g(y)) = x + P_y$ та $|P(x, g_y)| = k+1$, що не міститиме g_x, але це суперечить твердженню, що g_x є сильними воротами для x у A. □

Теорема 10. *Нехай G є зв'язним графом, $A \subset V(G)$ та $|A| \geq 2$. Тоді множина A є AP-опуклою тоді і тільки тоді, коли кожна вершина $x \in V(G)$ має сильні ворота в A.*

Доведення. Необхідність. Від супротивного, припустимо, що для $x \in V(G)$ не існує сильних воріт у A. Тоді $x \in V(G) \setminus A$ та для будь-якої вершини $a_1 \in A$ що для найкоротшого простого шляху $P_{a_1} = P(x, a_1)$ виконується $A \cap P_{a_1} = a_1$ існує інша вершина $a_2 \in A$, що існує найкоротший простий шлях $P_{a_2} = P(x, a_2)$, що $A \cap P_{a_2} = a_2$. Тоді між a_1 та a_2 існуватиме простий шлях поза межами A, що суперечить AP-опуклості A.

Достатність також доведемо від супротивного. Припустимо, що між двома різними вершинами $u, v \in A$ існує простий шлях поза межами A, позначимо його $P_{u,v}$. Зафіксуємо вершини $x, y \in P_{u,v} \setminus A$, що $d(x, A) = d(y, A) = 1$ [Рис.2]. Зауважимо, що за попередньо доведеною лемою всі вершини з

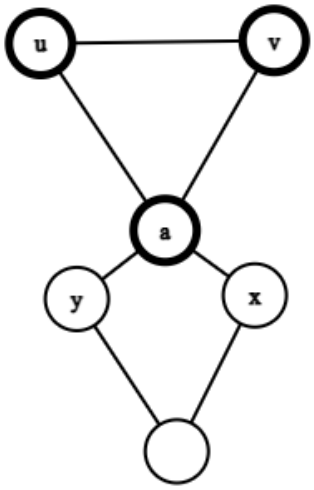


Рис. 2: Граф G та ар-опукла $S = \{u, v, a\}$

$P_{u,v} \setminus A$ матимуть однакові сильні ворота в A, позначимо їх g_P. З цього випливає, що x та y мають бути суміжними тільки з однією вершиною з A і це має бути g_P. Але це суперечить твердженню, що $P_{u,v}$ є простим шляхом. □

Наслідок 2. *Для кожного максимального зв'язного підграфу індукованого множиною $V(G) \setminus A$ існує єдина вершина з A, що буде сильними воротами для всіх вершин з відповідного підграфу.*

Представимо нову теорему, що є характеристикою графів блоків через ар-опуклість закритих околів його вершин.

Теорема 11. *Для зв'язного графа G наступні твердження еквівалентні:*

1. *G є графом блоків.*
2. *$N_G[u]$ є ар-опуклою для $\forall u \in V(G)$.*
3. *$N_G[u]$ є множиною із сильними воротами для $\forall u \in V(G)$.*

Доведення. $1 \Rightarrow 2$. Оскільки будь-який блок в G є клікою $N_G(u)$ буде об'єднанням блоків, що містять u, тому за першим критерієм ар-опуклих множин твердження є справедливим.

$2 \Leftrightarrow 3$. Справедливість цього твердження випливає з дрогого критерія ар-опуклих множин.

$2 \Rightarrow 1$. За першим критерієм ар-опуклих множин, множини $N_G[u]$ для $\forall u \in V(G)$ є блоками, або зв'язними об'єднаннями блоків. Без втрати загальності, зафіксуємо таке u, що $N_G[u]$ є блоком, і припустимо, що $N_G[u]$ не є клікою. Тоді існуватиме пара несуміжних вершин $v_1, v_2 \in V(N_G[u])$. Але тоді $N_G[v_1](N_G[v_2])$ не міститиме $v_2(v_1)$ і міститиме u, тому не буде максимальною двозв'язною компонентою. Отримали суперечність. □

Наступна теорема доводить деякі властивості ар-опуклих множин в деревах.

Теорема 12. *Нехай G є зв'язним графом з $|V(G)| = n$. Тоді наступні твердження еквівалентні.*

1. *G є деревом.*
2. *Існує послідовність множин $V(G) = V_n \supsetneq V_{n-1} \supsetneq ... \supsetneq V_1$, де для кожної i, V_i є AP-опуклою та $|V_i| = i$.*
3. *Будь-який зв'язний підграф графа G є AP-опуклим.*

Доведення. $1 \Rightarrow 3$. Оскільки будь-яка зв'язна підмножина G є однозв'язною, вона також буде AP-опуклою.

$2 \Rightarrow 1$. За характеризацією AP-опуклих множин, кожна така множина є зв'язним об'єднанням блоків. Розглянемо вершину $u \in V_{i+1} \setminus V_i$. Така u, очевидно не належить жодному блоку з V_i, проте кількість блочних підмножин в V_{i+1} має бути більшою за кількість цих множин в V_i. Таке можливо тільки коли u є листком в $G[V_{i+1}]$. Причому, оскільки це правдиво для кожного i, граф G має бути деревом.

$3 \Rightarrow 2$. Очевидно □

Загалом, знаходження кількості нетрівіальних опуклих множин графа є нетривіальною та складною задачею. Проте, для AP-опуклих множин наступне твердження є правдивим.

Теорема 13. *Якщо G є об'єднанням n блоків, кожний блок суміжний якнайбільше з двома іншими блоками та для кожної шарнірної вершини u виконується $deg(u) \leq 2$, то кількість нетривіальних AP-опуклих множин у графі G рівна $\frac{n(n+1)}{2}$.*

Доведення. Нехай блок B_i суміжний з блоками B_{i+1} та B_{i-1}. Оскільки об'єднання зв'язних блоків є AP-опуклою множиною, ця задача еквівалентна знаходження кількості j-елементних підпослідовностей n-елементної послідовності для $j = 1, ..., n$. Тож, кількістю нетривіальних ар-опуклих підмножин дорівнюватиме $\frac{n(n+1)}{2}$. □

Якщо з умови минулої теореми прибрати обмеження на кількість суміжних блоків, то для такого G кількість нетривіальних AP-опуклих множин дорівнюватиме кількості зв'язних підмножин дерева G' такого, що $|V(G')| = n$, кожному блоку B_i з G відповідає вершина v_i з $V(G')$ зі збереженням суміжності.

Наступне твердження є характеризацією екстремальних вершин для AP-опуклих множин.

Теорема 14. *Нехай множина $S \subset G$ є AP-опуклою та, що $u \notin S$. Тоді $S \cup \{u\}$ є опуклою тоді і тільки тоді, коли існує $v \in S$, що суміжна з u та u,v не лежать на циклі.*

Доведення. Для доведення необхідності припустимо, що u та v лежать на циклі. Тоді, за характеристикою AP-опуклих множин S є блоком або об'єднанням блоків, тож u має належати блоку з S, що суперечить умові.

Достатність. З умови випливає, що $\{u, v\}$ є AP-опуклою множиною, адже $\{u\}$ та $\{v\}$ суміжні і не лежать на циклі. Тому $S \cup \{u, v\}$ є AP-опуклою як об'єднання зв'язних AP-опуклих множин. □

Наслідок 3. *Кожна вершина блоку $B \simeq K_2$ є екстремальною.(можна переписати попередню теорему як є екстремальною тоді і тільки тоді коли лежить в K_2 блоці)*

Висновки

У даному розділі була досліджена ар-опуклість. Було надано два нових критерії ар-опуклих множин, а вже відомі теоретичні результати були передоведені користуючись новими критеріями. Також, була представлена нова характеристика графів блоків через ар-опуклість всіх закритих околів вершин графа.

Класичні задачі

Класичними задачами в контексті будь-якої опуклості вважаються наступні:

1. Для $S \subseteq V$ графа G визначити чи є S опуклою.
2. Для $S \subseteq V$ графа G визначити $I(S)$.
3. Для $S \subseteq V$ графа G визначити $H(S)$.
4. Для графа G визначити $c(G)$.
5. Для графа G визначити $i(G)$.
6. Для графа G визначити $h(G)$.
7. Для графа G визначити $gin(G)$.

Розглянемо розв'язки вище наведених задач у контексті all-path опуклості. Зауважимо, що в роботі "All-path convexity: Combinatorial and complexity aspects"[17] доведено існування алгоритму лінійної складності що пропорційна сумі вершин та ребер графа, для кожної згаданої задачі. Їх алгоритмічний розв'язок базується на двох алгоритмах вперше представлених в роботі [11], що в свою чергу є модифікаціями відомого алгоритму на графах *пошуку в глибину* (*DFS*). DFS алгоритм полягає в наступному:

- Фіксуємо будь-яку вершину графа.
- Якщо можливо, переходимо до невідвіданої сусідньої вершини.
- Повертаємося до попередньої вершини, якщо не залишилося невідвіданих сусідніх вершин для поточної вершни.

Даний алгоритм дає спосіб обходу всіх вершни (або ребер) за лінійний час. Також, DFS природнім чином надає орієнтацію ребрам, направляючи їх в напрямку від поточної вершини до досі невідвіданого сусіда. Наведемо псевдокод алгоритма:

Оскільки кожна вершина може бути відмічена як пройдена тільки одного разу, відповідний код відпрацює таку кількість разів, що пропорційна потужності множини вершин графа. Друга чатина алгоритма полягає в

Algorithm 1 Алгоритм пошуку в глибину

```
function DFS(v)
    visited[v] ← True                          ▷ Відпрацює |V| разів
    for u ∈ N(v) do                            ▷ Відпрацює 2|E| разів
        if u is not visited then
            DFS(u)
        end if
    end for
end function
```

тому, щоб перевірити чи всі сусідні вершини є відвіданими. Це можна інтерпретувати як перевірку того, чи відповідне ребро є пройденим. Оскільки даний блок відпрацює один раз для кожної вершини, кожне ребро буде перевірено двічі. З цього випливає, що друга частина коду загалом відпрацює кількість разів, що пропорційна подвоєній кількості ребер. Якщо припустити що всі перевірки та маніпуляції мають константну часову складність $O(1)$, з цього випливає, що мінімальна часова складність DFS алгоритму складає $T(G) = |V| \times O(1) + 4 \times |E| \times O(1)$ або ж $O(G) = |V| + |E|$. Наведена часова складність обумовлена одним "відмічанням" для кожної вершини та двома перевірками для кожного ребра : чи є ребро пройдем та чи є це твердження правдивим.

Версія цього алгоритма для обходу всіх вершин графа може бути використана для вирішення задачі знаходження зв'язних компонент графа [11].

Алгоритм перевірки на ap-опуклість

В роботі "All-path convexity: Combinatorial and complexity aspects"[17] був представлений наступний критерій all-path опуклості множини.

Теорема 15. *Зафіксуємо $S \subset V$. Тоді S є AP-опуклою тоді і тільки тоді, коли $S = V$, або для кожного зв'язної компоненти G_i, що $V(G_i) \subset V(G - S)$, існує тільки одна сусідня вершина з S.*

Цей критерій надає зручну умову перевірки множини на ap-опуклість для алгоритму визначення чи є множина $S \subseteq V$ ap-опуклою. Автори статті також запропонували алгоритм перевірки множини S на ap-опуклість, що полягає у знаходженні неперетинних зв'язних компонентів у графі породженому $V \setminus S$. Даний алгоритм наводиться в [11] та його основою є уже

згаданий DFS, тому мінімально можливою часовою складністю такого алгоритма є $T(G) = |V| \times O(1) + 4 \times |E| \times O(1)$ або ж $O(G) = |V| + |E|$. Далі автори пропонують використати свій критерій для перевірки S на ар-опуклість. Тобто, для кожної зв'язної компоненти C_i графа $V \setminus S$ необхідно перевірити кількість сусідніх вершин в S. Для цього необхідно перебрати кожне суміжне ребро C_i, що обумовлює використання, ще одного DFS для ефективного проходу по всім ребрам. З цього випливає, що в статті "All-path convexity: Combinatorial and complexity aspects"[17] запропонований алгоритм перевірки множини S на ар-опуклість вимагає два відпрацювання DFS алгоритму, а тому має мінімально можливу часову складність $T(G) = 2|V|O(1) + 8|E|O(1)$ або ж $O(G) = |V| + |E|$. Далі ми представимо алгоритм, максимальна швидкість якого буде меншою за мінімально можливу швидкість запропонованого в [17] алгоритма.

Наш алгоритм спиратиметься тільки на одне відпрацювання DFS. Він полягатиме у використанні запропонованого нами критерію ар-опуклої множини через сильні ворота. Алгоритм полягає в наступному: з кожної вершини $v \in S$ ініціюємо DFS обхід вершин не з S; якщо під час такого обходу ми наштовхуємось на таку вершину $u \in S$ та $u \neq v$, то S не буде ар-опуклою. У псевдокоді алгоритму E_A позначатиме множину напрямлених ребер, що починаються із вершин, що належать підмножині A. Представимо псевдо код алгоритму:

Оцінимо часову складність алгоритму. Припустиму, що всі дії та перевірки відбуваються із константною складністю. Звернемо вашу увагу, що над вершинами (ребрами), що належать S(починаються з вершини з S), проводяться маніпуляції тільки в середині функції $triggerDFS$, а над вершинами (ребрами), що належать $V \setminus S$(починаються з вершини з $V \setminus S$), проводяться маніпуляції тільки в середині функції DFS. Тоді, враховуючи три присвоєння на початку функції $triggerDFS$ максимальна часова складність алгоритму дорівнюватиме

$$T(G) = |V \setminus S|O(1) + |S|O(1) + 2|E_S| + 2|E_{V\setminus S}| + 4\frac{|E_{V\setminus S}|}{2} + 3O(1) \leq$$
$$|V|O(1) + 4|E| + 4|E| + 3O(1) = |V|O(1) + 8|E| + 3O(1)$$

або ж $O(G) = |V| + |E|$. Така часова складність як $|V|O(1) + 8|E| + 3O(1)$ є максимально можливою для нашого алгоритму та досі є меншою за мінімально можливу часову складність попереднього алгоритму.

Оцінимо кількість необхідної пам'яті для алгоритму. Для нього використовується список ребер графа, список відвіданих вершин та список вершин з S. Тому справедливо оцінити необхідну пам'ять для алгоритма як

Algorithm 2 Алгоритм перевірки на ap-опуклість

```
function triggerDFS(S)
    visited ← S
    blocked ← S
    ap-convex ← True
    for v ∈ S do
        strongGates ← v                                  ▷ Відпрацює |S| разів
        for u ∈ N(v) do                                  ▷ Відпрацює |E_S| разів
            if u is not visited then
                DFS(u)
            end if
        end for
    end for
end function
function DFS(v)
    visited[v] ← True                                    ▷ Відпрацює |V \ S| разів
    for u ∈ N(v) do                                      ▷ Відпрацює |E_{V\S}| разів
        if u is not visited then
            DFS(u)
        else                                             ▷ Відпрацює |E_{V\S}|/2 разів
            if u is in blocked and u ≠ strongGates then
                ap-convex ← False
            end if
        end if
    end for
end function
```

$M(G) = |E| + |V| + |S| \leq |E| + 2|V|$.

Порівняємо роботу реалізації нового алгоритму. Порівняння проводилося з нашими реалізаціями цього ж алгоритму але за описом з [17]. Оскільки вони не є максимально оптимальними реалізаціями, тому варто сприймати цю перевірку, як порівняння з алгоритмом повного перебору. Порівняння проводилося наступним чином: для кожної кількості вершин i з проміжку $10 \leq i \geq 300$ генерувалося 40 випадкових зв'язних графів та 40 відповідних випадкових підмножин; та замірялася середня тривалість роботи кожного алгоритма для кожної i. Результати перевірки зображені на Рис. 3

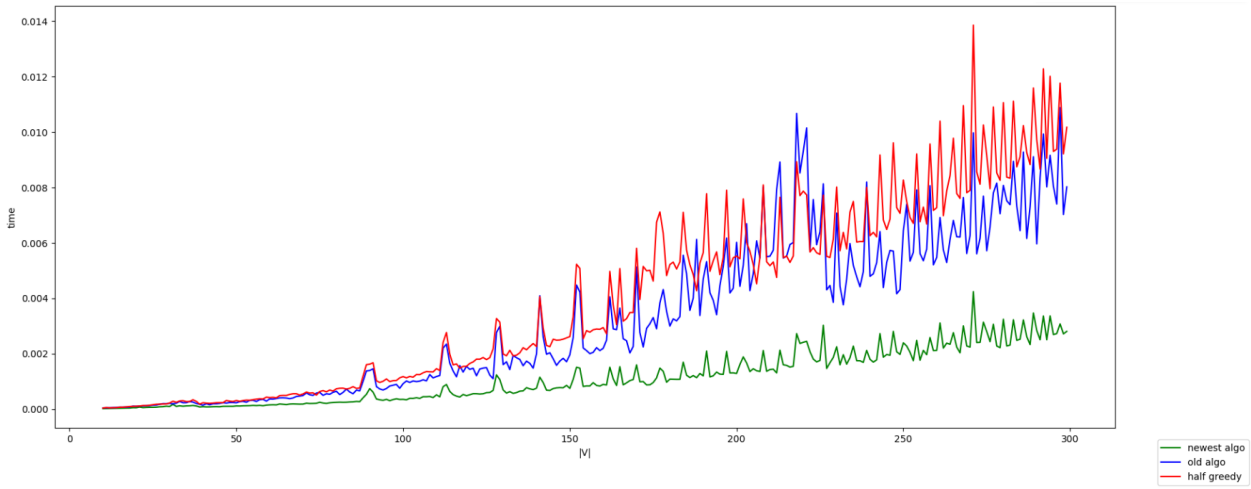
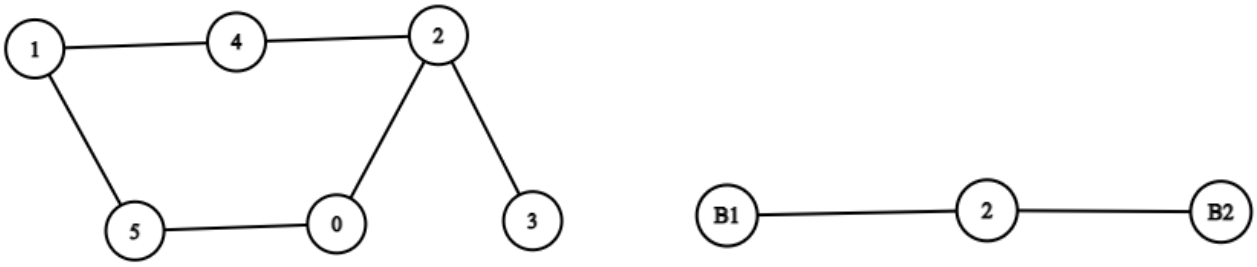


Рис. 3: Порівняння роботи алгоритмів

Ми також реалізували даний алгоритм за допомогою мови програмування Python. Код реалізації наведено в додатках.

Алгоритм знаходження $I(S)$, $H(S)$ та $gin(G)$

Задача визначення $I(S)$. Розглянемо декомпозицію графа G за допомогою шарнірно-блочного дерева T_G де кожна вершина в T_G є образом блока B_j або шарнірної вершини z_i графа G. Причому між вершинами-образами B_j та z_i існує ребро в T_G коли $z_i \in B_j$ в графі G. Очевидно, що листки T_G завжди є образами блоків, такі блоки називають кінцевими. На Рис. 4 (а) зображений граф G з двома блоками $B_1 = \{0, 1, 2, 4, 5\}$ та $B_2 = \{2, 3\}$, а поруч – $T(G)$ Рис. 4 (б).



(а) G (б) $T(G)$

Рис. 4: Граф G та його T_G

Для множини $S \subseteq V$ покладемо T_S максимальним піддеревом T_G таким чином, що кожний листок T_S є образом блока графа G, що вміщає вершину з S, що не є шарнірною вершиною підграфа G_S породженого $\bigcup_{B_j \in V(T_S)} B_j$.

Теорема 16. *Для множини $S \subseteq V$ покладемо u, w є різними вершинами в S, що належать блокам B_u та B_w відповідно. Без втрати загальності припустимо, що u та v не є шарнірними вершинами в G_S. Нехай $B_{j1}z_1B_{j2}z_2...z_{k-1}B_{jk}$ є шляхом в T_S між $B_{j1} = B_u$ та $B_{jk} = B_w$. Тоді для всіх $v \in \bigcup_{i=1}^{k} V(B)$ існує простий шлях P в G з кінцевими точками u та w, що проходить через v.*

Доведення. Доведемо твердження користуючись властивістю блоків, що для кожної пари вершин існує простий шлях через будь-яку іншу вершину. Для випадку $B_u = B_w$ з цієї властивості одразу випливає вірність твердження. Розглянемо інший випадок. За цією властивістю між кожною наступною парою вершин $\{u, z_1\}$, $\{z_i, z_{i+1}\}$ та $\{w, z_2\}$ має існувати шлях через кожну вершину відповідного блоку B_{j1}, B_{ji+1} та B_{jk}, що доводить твердження. □

Наступна теорема доводить значення інтервальної функції для множин графа G.

Теорема 17. *Покладемо $S \subseteq V$ із $|S| \geq 2$. Тоді $I(S) = \bigcup_{B_j \in V(T_S)} B_j$*

Доведення. Із умови побудови T_S випливає, що кожний блок з T_S має містити вершину з S, або містити частину шляху між якимись вершинами з S. Причому, для будь якого $B_j \in T_S$ завжди існують такі дві вершини $u, v \in S$, що існує такий простий шлях $P_{u,v}$, що $|P_{u,v} \cap B_j| \geq 2$. Правдивість цього твердження разом із властивістю блока, що між кожною парою вершин блока існує простий шлях через будь-яку іншу вершину блока, доводить це твердження. Доведемо його правдивість від супротивного. Припустимо, що

для зафіксованого блоку $B_j \in T_S$ такої пари вершин не існує. Тоді перетин будь-якого простого шляху P між вершинами S та B_j має щонайбільше одну вершину. Ця вершина завжди має бути шарнірною, тому B_j не містить точок з S. Але B_j також є листком графа T_S тому він має містити вершину з S, отримали суперечність. □

Наслідок 4. *З характеристики AP-опуклої множини випливає, що $I(S)$ завжди є опуклою множиною, а $H(S) = I(S)$.*

Загальний алгоритм знаходження $I(S)$ полягатиме у знаходженні $T(G)$, та подальшого обрахунку $T(S)$. Для знаходження $T(G)$ ми користуємся класичним лінійним алгоритмом знаходження двозв'язних компонент графа G представленим в [11]. Проте, зауважимо, що алгоритм знаходження двозв'язних компонент досі утримує увагу дослідників та активно покращується. Наприклад, в праці [21] представили алгоритм знаходження двозв'язних компонент на GPU з використанням паралельних обчисоювань, що, як стверджується, щонайменше в четверо швидше будь-якої іншої імплементації цього алгоритму.

Розглянемо алгоритм знаходження двозв'язних компонент з [11]. Цей алгоритм виконує пошук в глибину по ребрам графа. Кожна нова досягнута вершина додається до стеку вершин та нумерується, також для кожної вершини зберігається найменший номер вершини яку з неї можна досягти через шлях з недосягнутих вершин, у [11] даний запис називають lowpoint. Після проходження по ребру, воно також видаляється та додається до стеку ребер. Якщо найперша вершина в стеку не має суміжних з нею ребер, вона видаляється зі стеку вершин, а пошук продовжується з нової найпершої вершини стека.

Алгоритм закінчує свою роботу, коли стек ребер графа стає пустим, це знаменує кінець дослідження всіх двозв'язних компонент однієї зв'язної компоненти. Ізольовані вершини не вважаються двозв'язними компонентами, тому вони пропускаються алгоритмом, оскільки не мають вихідних ребер. Наведемо алгоритм більш детально за допомогою діаграми на Рис. 5

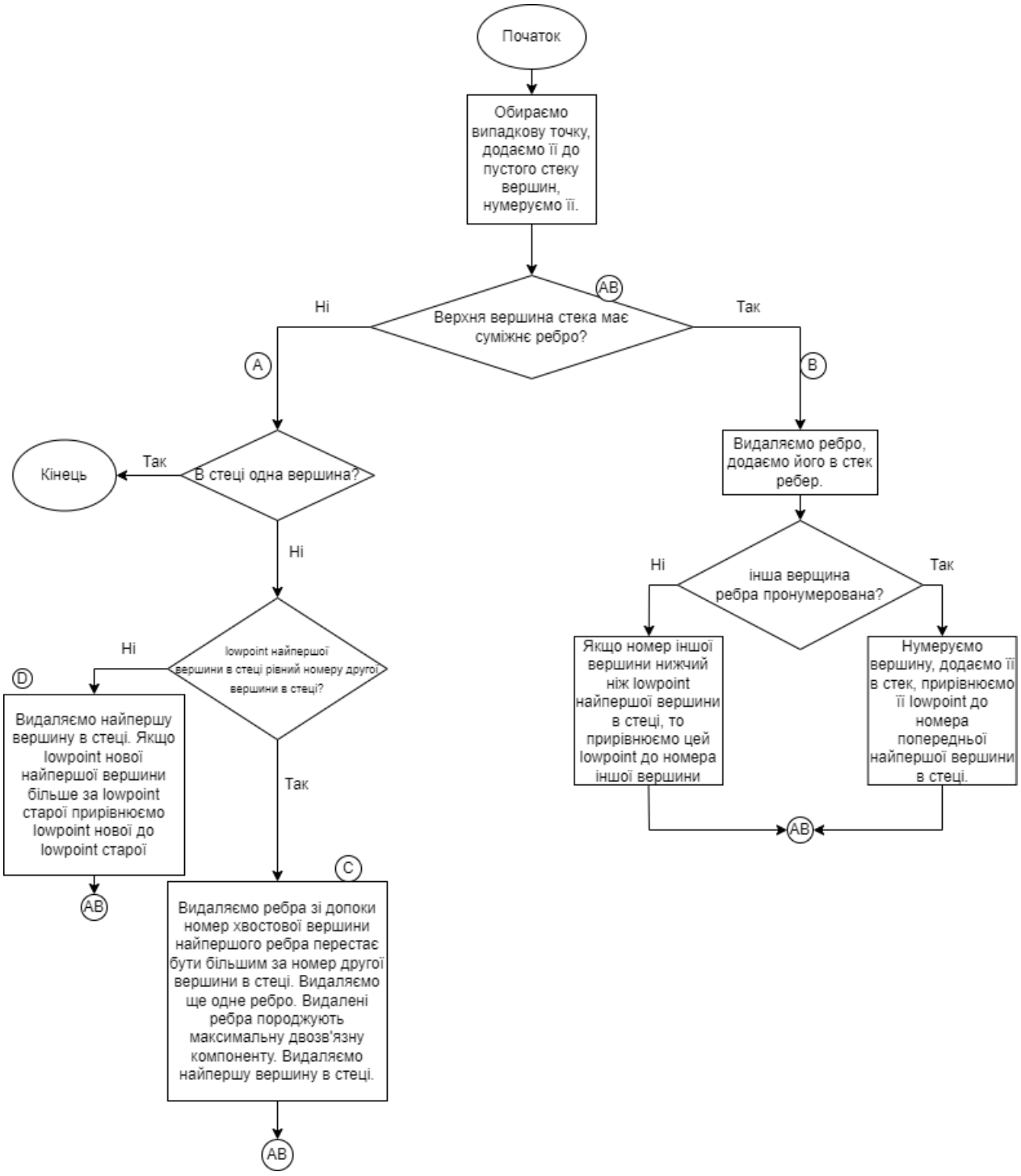


Рис. 5: Діаграма алгоритму знаходження двозв'язних компонент

Хочемо зауважити, що дана діаграма на Рис. 5 є майже повною копією діаграми з [11], проте ми виправили деякі помилки. Так, наприклад, блок C діагарами дещо відрізнявся від наведеного нами: умовою зупинки процесу видалення ребер зі стеку було досягання ребра, що пов'язаний із

третьою вершиною в стеку вершин, проте дана умова здатна провокувати помилку, що залежить від прядку обходу вершин. Ми можемо довести це продемонструвавши два варіанти відпрацьовування алгоритму з різним порядком обходу ребер. Ми наводимо хід роботи алгоритма наступним чином. Ми наводимо опис послідовних кроків у наступному форматі: граф чиї ребра видаляються в ході роботи алгоритма, стек ребер куди потрапляють видалені ребра, що отримують природню орієнтацію в напрямку новішої вершини, стек вершин, що зберігає пари (вершина, lowpoint) та список порядку відвідування вершин.

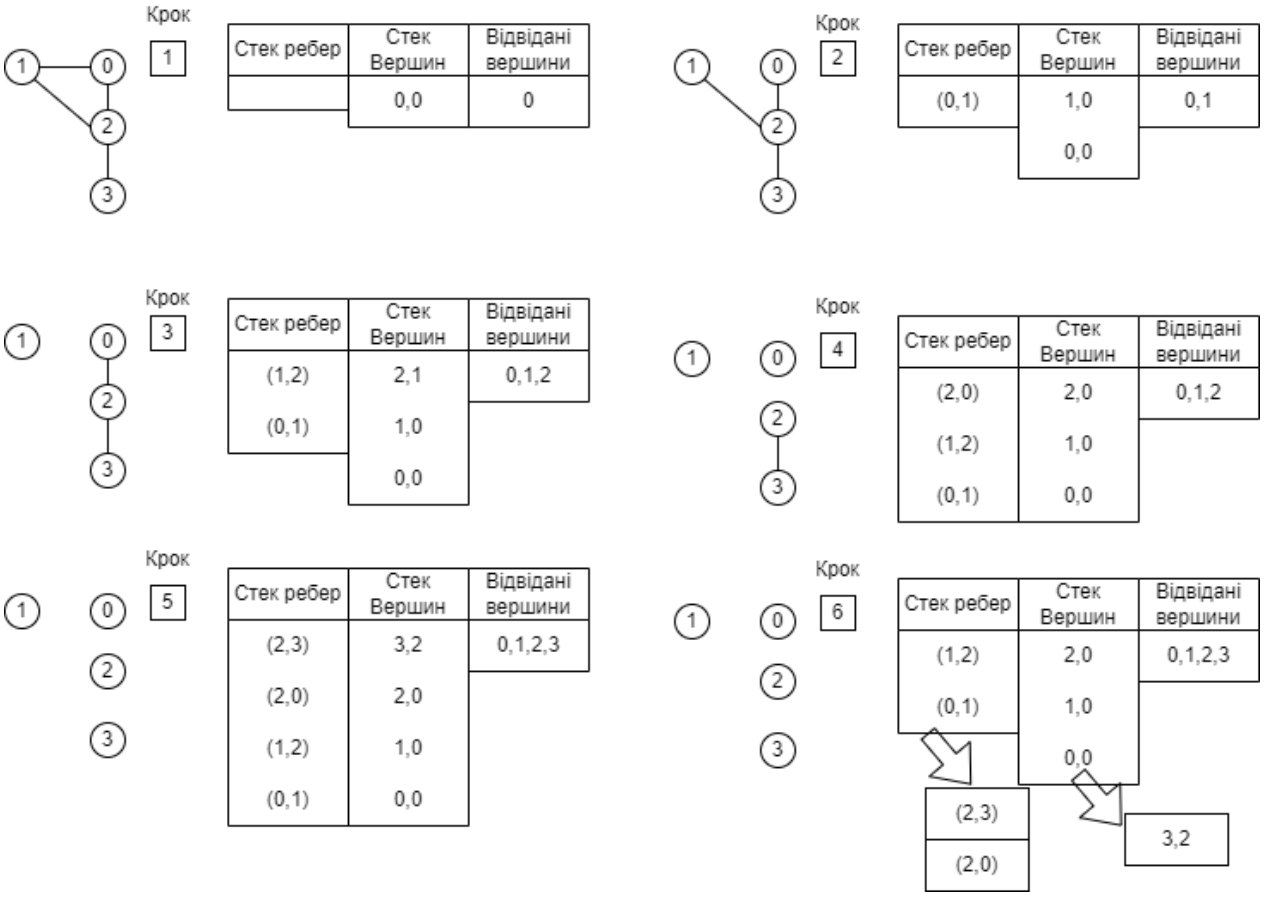


Рис. 6: Порядок роботи алгоритму з неправильною умовою

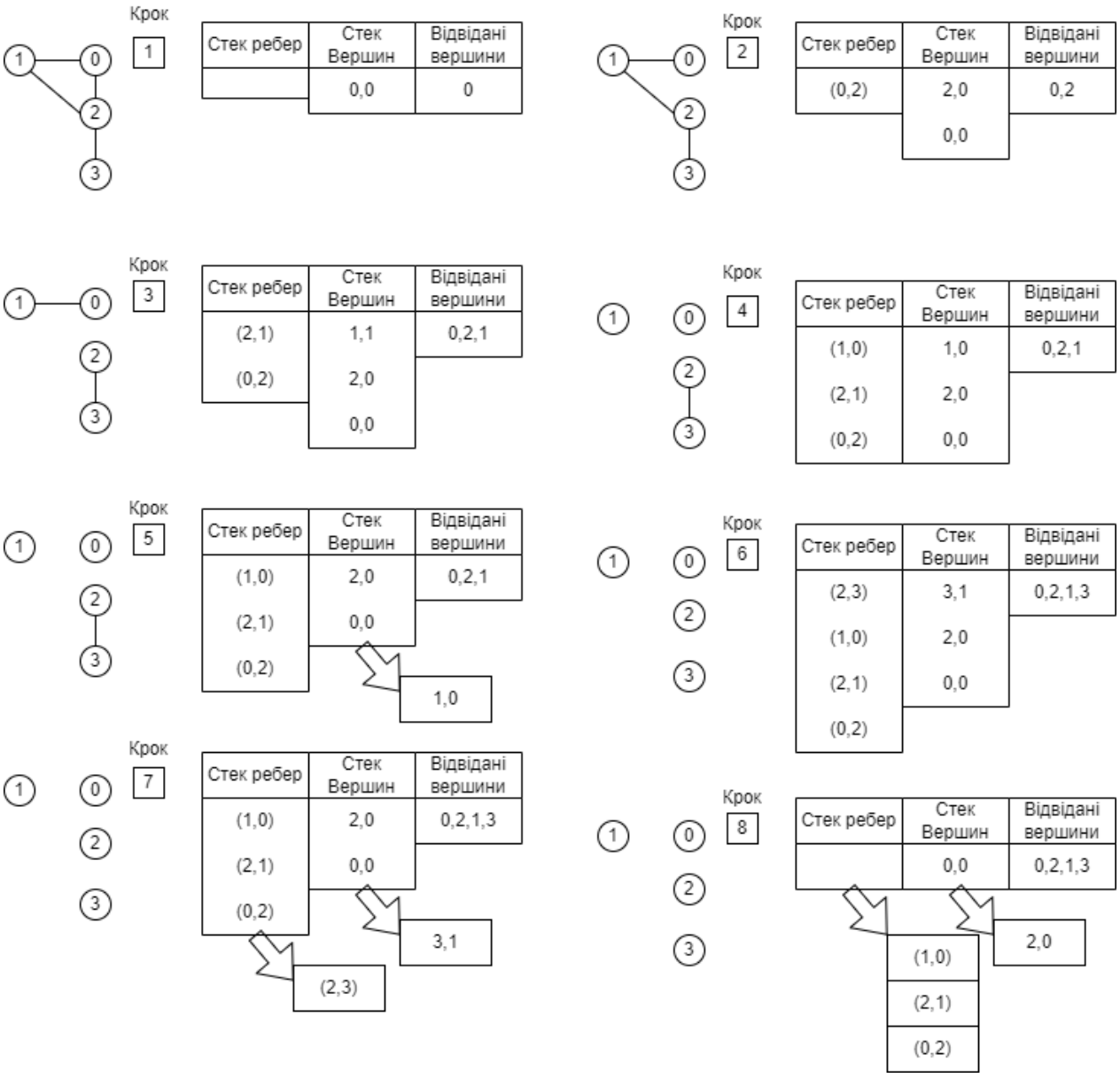


Рис. 7: Порядок роботи алгоритму з правильною умовою

Так Рис. 6 демонструє відпрацювання алгоритму із неправильною умовою і на шостому кроці повертає двозв'язну компоненту породжену ребрами $(2,3)$ та $(2,0)$, проте даний породжений підграф не є максимальною двозв'язною компонента графа G з першого кроку. Натомість на Рис. 7 продемонтсровано роботу алгоритму з правильною умовою зупинки видалення ребер. Також в оригінальній роботі допущена ще одна помилка в блоці C. Там не зазначалась необхідність видалення верхньої вершини стека в кінці відпрацювання блока C. Проте не важко зрозуміти, що це б провокувало безкінечну реактивацію блока C.

Також під час знаходження двозв'язних компонент ми маємо змогу конструювати шарнірно-блочне дерево графа. Спосіб яким ми реалізовуємо цю побудову спирається на наступний факт: при створенні двоз'язної компоненти в блоці C всі її шарнірні вершини є відомими, адже всі її вершини є повністю дослідженими окрім однієї, що знаходитиметься другою в стеку і тому вважатиметься шарнірною, навіть якщо не всі її суміжні ребра є дослідженими. Тому у ході відпрацювання алгоритму ми зберігаємо інформацію щодо уже підтверджених шарнірних вершин і при створенні двозв'язної компоненти додаємо ребро між неї та всіма її шарнірними вершинами у списку суміжності. Таким чином ми створюємо шарнірно-блочне дерево за лінійний час. Ми презентуємо реалізацію цього алгоритму мовою програмування Python у додатках.

Після знаходження двозв'язних компонент для графа G та створення його шарнірно-блочного дерева останнім кроком для знаходження $I(S)$ є знаходження $T(S)$. Алгоритм знаходження $T(S)$ із $T(G)$ полягає в наступному: для кожного листка B_l із $T(G)$ ми проводим перевірку чи $|S \cap B_l| = 0$ або $|S \cap B_l| = 1$ та чи $S \cap B_l = v$ є шарнірною вершиною вершиною для B_l. Якщо перевірка спровджується B_l видаляється з $T(G)$. Цей процес видалення листків триватиме доки жоден з листків не пройде перевірку. В результаті цього, буде знайдено $T(S)$, що співпадатиме з $I(S)$. Оскільки алгоритм знаходження $T(S)$ є лінійним відносно суми $|V| + |E|$ за часом виконання та за використанням пам'яті, алгоритм $I(S)$ також буде лінійним. Ми наводимо його реалізацію мовою Python у додатках.

Задача знаходження $H(S)$ співпадає із задачею знаходження $I(S)$ для ар-опуклості.

Також задача знаходження числа геодетичних ітерацій є тривіальною для ар-опуклості адже $I(S)$ є опуклою для будь-якої S.

Задача знаходження $c(G)$

Задача знаходження $c(G)$ полягає у пошуку найбільшої нетривіальної опуклої множини. Також із представленого нами критерія ар-опуклості множини, такою є підмножина, що є блоком або зв'язним об'єднанням блоків. З цього випливає алгоритм знаходження $c(G)$. Для тривіального випадку коли $|V| = 2$ або G є двозв'язним $C(G)$ очевидно дорівнюватиме 1. Інакше нам достатньо знайти кількість вершин у блоках-вершинах $T(G)$ без найменшого за потужністю блока-листка. Тому даний алгоритм є лінійним за часом та за використанням пам'яті відносно суми $|V|+|E|$. Ми презентуємо

його реалізацію мовою програмування Python в додатках.

Задача знаходження $i(G)$ та $h(G)$

Далі розглянемо задачу знаходження $i(G)$. Для тривіального G очевидно, що $i(G) = 1$. Якщо G є двозв'язним то, оскільки між кожною парою вершин блока існує простий шлях через будь-яку іншу вершину блока, $i(G) = 2$. Для загального випадку розглянемо $T(G)$. Нам достатньо розглянути множину вершин S що її потужність $|S|$ рівна кількості листків $T(G)$ і для кожної $u \in S$ існує листок $T(G)$ що йому належить u, що також не є шарнірною вершиною. Тоді $I(S)$ очевидно проходитиме через кожний блок, більш того будь-який простий шлях, що з'єднує всі вершини з S матиме щонайменше по дві вершини зі всіх блоків тому $i(G) = |S|$. Така S є мінімальною адже, ящо не додати не шарнірну вершину з листка-блока B_l з шарнірно-блочного дерева до S тоді не існуватиме простого шляху, що поєднує дві вершини з S і містить хоча б дві вершини з B_l, тому для такої S інтервальне число $i(G) > |S|$.

З цього слідує алгоритм знаходження $i(S)$. Він полягає у побудові $T(G)$ та підрахунку кількості блоків-листків $T(G)$. З цього випливає, що даний алгоритм є лінійним за часом та за використанням пам'яті відносно суми $|V| + |E|$. Також, ми представляєм його реалізацію мовою програмування Python в додатках.

Для ар-опуклості $h(G) = i(G)$ тому алгоритми їх знаходження співпадають.

Висновки

У даному розділі були досліджені класичні задачі в контексті ар-опуклості. Також, був представлений новий алгоритм детермінації ар-опуклості множини, що теоретично є швидшим за аналоги. Також,були реалізовані алгоритми розв'язання класичних задач в контексті ар-опуклості.

Висновки

В роботі було досліджено та проаналізовано низку опуклостей: all-path опуклість, digital опуклість та геодетичну опуклість. Також були висвітлені численні теоретичні результати пов'язані зі згаданими опуклостями. Для all-path опуклості було запропоновано два нових критерії all-path опуклості підмножини, та на їх основі був представлений новий критерій графів блоків. В контексті деяких опуклостей була висвітлена тема класичних задач для опуклостей на графах. Для ap-опуклості були реалізовані алгоритми розв'язків всіх класичних задач, та був запропонований новий алгоритм детермінації того чи є підмножина ap-опуклою. Даний алгоритм виявився теоретично швидшим за попередників.

Література

[1] Benevides, Fabricio, et al. "Complexity of determining the maximum infection time in the geodetic convexity."Electronic Notes in Discrete Mathematics 50 (2015): 403-408.

[2] Cáceres, José, et al. "On geodetic sets formed by boundary vertices."Discrete Mathematics 306.2 (2006): 188-198.

[3] Cáceres, José, et al. "Rebuilding convex sets in graphs."Discrete Mathematics 297.1-3 (2005): 26-37.

[4] Changat, Manoj, Iztok Peterin, and Abisha Ramachandran. "On gated sets in graphs."(2016): 509-521.

[5] Dourado, Mitre C., Fábio Protti, and Jayme L. Szwarcfiter. "Complexity results related to monophonic convexity."Discrete Applied Mathematics 158.12 (2010): 1268-1274.

[6] Duchet, Pierre. "Convex sets in graphs, II. Minimal path convexity."Journal of Combinatorial Theory, Series B 44.3 (1988): 307-316.

[7] Everett, Martin G., and Stephen B. Seidman. "The hull number of a graph."Discrete Mathematics 57.3 (1985): 217-223.

[8] Farber, Martin, and Robert E. Jamison. "On local convexity in graphs."Discrete Mathematics 66.3 (1987): 231-247.

[9] González, Lucía M., et al. "Covering graphs with convex sets and partitioning graphs into convex sets."Information Processing Letters 158 (2020): 105944.

[10] F. Harary, On the reconstruction of a graph from a collection of subgraphs, Proc. of the Symp. Theory of Graphs and its Appl., Prague (ed. M. Fiedler); reprinted by Academic Press, New York, 1964, pp. 47–52.

[11] Hopcroft, John, and Robert Tarjan. "Algorithm 447: efficient algorithms for graph manipulation."Communications of the ACM 16.6

(1973): 372-378.

[12] Kelly, Paul Joseph. On isometric transformations. Diss. 1942.

[13] Lafrance, Philip, Ortrud R. Oellermann, and Timothy Pressey. "Reconstructing trees from digitally convex sets."Discrete Applied Mathematics 216 (2017): 254-260.

[14] Lafrance, Philip, Ortrud R. Oellermann, and Timothy Pressey. "Generating and enumerating digitally convex sets of trees."Graphs and Combinatorics 32 (2016): 721-732.

[15] Oellermann, Ortrud R. "On domination and digital convexity parameters."J. Combin. Math. Combin. Comput 85 (2013): 273-285.

[16] Pfaltz, J. L., and A. Rosenfeld. Pattern recognition-iv. sequential operations in digital picture processing. No. NASA-CR-63948. 1965.

[17] Protti, Fábio, and Joao VC Thompson. "All-path convexity: Combinatorial and complexity aspects."Ars Combinatoria 148 (2020): 77-87.

[18] Sampathkumar, E. "Convex sets in a graph."Indian J. pure appl. Math 15.10 (1984): 1065-1071.

[19] Ulam, Stanislaw M. "A collection of mathematical problems."New York 29 (1960).

[20] van De Vel, Marcel LJ. Theory of convex structures. Elsevier, 1993.

[21] Wadwekar, Mihir, and Kishore Kothapalli. "A fast GPU algorithm for biconnected components."2017 Tenth International Conference on Contemporary Computing (IC3). IEEE, 2017.

Додатки

Алгоритм перевірки на ар-опуклість

- Необхідні модулі.

```
from collections import defaultdict
from copy import deepcopy
from functools import reduce
```

- Репрезенація простого графа.

```
class SimpleGraph:

    def __init__(self):
      # default dictionary to store graph
        self.graph = defaultdict(set)


    def addEdge(self, u, v):
        """
        This function adds an edge to indirected graph
        """
        self.graph[u].add(v)
        self.graph[v].add(u)

    def vertexes(self):
      return self.graph.keys();

    # Outputs adjacency matrix of a graph
    def __str__(self):
        """
        This function outputs adjacency matrix of a graph
        """
        verts = self.vertexes()
        string = ""
        for v in verts:
            string += f"{v} : {self.graph[v]} \n"
        return string
```

- Приклад створення графа.

```
g = SimpleGraph()

g.addEdge(0, 1)
g.addEdge(1, 2)
g.addEdge(1, 3)
g.addEdge(2, 3)
g.addEdge(3, 4)
g.addEdge(3, 5)
g.addEdge(3, 6)
g.addEdge(5, 6)
```

- Алгоритм визначення чи є підмножива S ap-опуклою.

```
class Graph(SimpleGraph):

    def __init__(self):
        SimpleGraph.__init__(self)

    def triggerDFS(self, s):
        """
        This function starts DFS traversal in order to
        determine whether a set s is ap-convex
        """
        self.visited = s.copy()
        self.blocked = s
        self.blocked_calm = True

        # triggering DFS traversal
        for v in s :
            self.start = v
            for u in self.graph[v]:
                if not u in self.visited:
                    self.__DFS(u)

        return self.blocked_calm

    def __DFS(self, v):
        """
        This function does DFS traversal
        """
        self.visited.add(v)

        # continue traversal if neighdor of v
        # is not visited and not from s
        for u in self.graph[v]:
            if not u in self.visited:
                self.__DFS(u)
            elif (u in self.blocked and
                  not u == self.start):
                self.blocked_calm = False
                return;
```

- Приклад використання алгоритма.

```
# creation of a graph
g = Graph()

g.addEdge(0, 1)
g.addEdge(1, 2)
g.addEdge(1, 3)
g.addEdge(2, 3)
g.addEdge(3, 4)
g.addEdge(3, 5)
g.addEdge(3, 6)
g.addEdge(5, 6)
```

```
12
13 #checking whether a set is ap convex in the graph
14 subset ={1,2,3}
15 g.triggerDFS(subset)
```

Алгоритм знаходження $I(S)$ та $H(S)$

- Алгоритм знаходження шарнірно-блочного дерева.

```
1
2  class BlockCutTree(SimpleGraph):
3
4      def __init__(self, g):
5          """
6          The constructor which expects to get
7          edge default dict of the graph G
8          """
9          SimpleGraph.__init__(self)
10         # initiation of fields
11         self.art_verts = set()
12         self.preimage_vertexes = g.vertexes()
13         self.edges = deepcopy(g.graph)
14         self.numbered_verts = defaultdict(int)
15         self.visited_edges = defaultdict(set)
16         self.vertex_stack = []
17         self.edge_stack = []
18         self.bi_components = []
19
20         # start of dfs traversal
21         self.__blocka()
22         # fill adj list in case G is biconnected
23         if len(self.bi_components) == 1:
24             self.graph[tuple(self.preimage_vertexes)] = []
25
26
27     def addEdge(self, u, v):
28         """
29         This function adds edge to adj list
30         """
31         self.graph[u].add(v)
32         self.graph[v].add(u)
33
34     def __isNumbered(self, v):
35         """
36         This function checks whether v is a numbered vertex
37         """
38         return v in self.numbered_verts.keys();
39
40     def __getNumber(self, v):
41         """
42         This function returns a number of a vertex v
43         """
44         return self.numbered_verts[v];
45
46     def __numberVert(self,v):
47         """
48         This function numbers a vertex v
49         """
50         num = len(self.numbered_verts.keys())
```

```
51          self.numbered_verts[v] = num
52          return num;
53
54      def __addEdgeToComp(self, s, edge):
55          """
56          This function ads edge to s
57          """
58          for i in edge:
59              s.add(i)
60          return;
61
62      def __deletePreImageEdge(self, u, v):
63          """
64          This function delets edge from adj list
65          """
66          self.visited_edges[u].add(v)
67          self.visited_edges[v].add(u)
68          return
69
70
71      # algorithm of each block is described
72      # on the scheme
73      def __blocka(self):
74          self.vertex_stack = []
75          next_num = len(self.numbered_verts.keys())
76          unnumbered_verts =[x for x in self.preimage_vertexes if not
    self.__isNumbered(x)]
77          start_point = unnumbered_verts[0]
78          self.vertex_stack.append([start_point, next_num])
79          self.numbered_verts[start_point] = next_num
80
81          self.__blockAB()
82
83
84      def __blockAB(self):
85
86          top = self.vertex_stack[-1][0]
87          edges = self.edges[top]
88          for e in edges:
89              if not e in self.visited_edges[top]:
90                  self.__blockB(e)
91          self.__blockA()
92
93      def __blockA(self):
94          if len(self.vertex_stack) == 1:
95              if len(self.numbered_verts.keys()) == len(self.
    preimage_vertexes):
96                  return
97              else:
98                  self.__blocka()
99          else:
100             self.__blockD()
101
102     def __blockB(self, e):
```

```
103         top = self.vertex_stack[-1]
104 
105         edge = [top[0], e]
106         self.__deletePreImageEdge(edge[0], e)
107         self.edge_stack.append(edge)
108 
109         if self.__isNumbered(e) :
110             head_num = self.numbered_verts[e]
111             if head_num < top[1]:
112                 self.vertex_stack[-1][1] = head_num
113         else:
114             self.__numberVert(e)
115             self.vertex_stack.append([e, self.numbered_verts[top
    [0]]])
116 
117         self.__blockAB()
118 
119 
120     def __blockC(self):
121         verts = set()
122         art_vertex = self.vertex_stack[-2][0]
123         if len(self.vertex_stack) > 2:
124             art_num = self.numbered_verts[self.vertex_stack[-2][0]]
125             while self.numbered_verts[self.edge_stack[-1][0]] >
    art_num:
126                 edge = self.edge_stack.pop()
127                 self.__addEdgeToComp(verts, edge)
128             edge = self.edge_stack.pop()
129             self.__addEdgeToComp(verts, edge)
130             self.art_verts.add(art_vertex)
131         else:
132             for i in self.edge_stack:
133                 self.__addEdgeToComp(verts, i)
134 
135         self.bi_components.append(verts)
136 
137         self.__make_component(tuple(verts))
138 
139         self.vertex_stack.pop()
140         self.__blockAB()
141 
142 
143     def __blockD(self):
144         top = self.vertex_stack[-1]
145         if top[1] == self.numbered_verts[self.vertex_stack[-2][0]]:
146             self.__blockC()
147         else:
148             lp_next = self.vertex_stack[-2][1]
149             if lp_next > top[1]:
150                 self.vertex_stack[-2][1] = top[1]
151             self.vertex_stack.pop()
152             self.__blockAB()
153 
154     def __make_component(self, comp):
```

```
        """
        This function connects blocks to it's cut vertexes in adj list
        """
        for i in comp:
            if i in self.art_verts:
                self.addEdge(comp, (i,))


    def __str__(self):
      blocks = self.graph.keys()
      string = ""
      for b in blocks:
        string += f"{b} : {list(self.graph[b])} \n"
      return string

```

- Приклад використання алгоритму.

```
bct = BlockCutTree(g)
```

- Алгоритм знаходження $I_G(S)$, що співпадає із алгоритмом знаходження $H_G(S)$.

```
def find_interval(subset, g):
    """
    This function finds ap-interval between all vertices of a subset
    """
    bct = BlockCutTree(g)
    graph = deepcopy(bct.graph)
    bct_leafs = [x for x in graph.keys() if len(x)>1 and len(graph[x])==1]

    # deleting leafs of the block cut tree until
    # all of them has a vertex from the subset
    for l in bct_leafs:
        leafs_delation(subset, l, graph)

    return graph

def leafs_delation(subset, l, graph):
    """
    This function deletes block-leaf until it has a vertex form the subset
    """
    verts = graph.keys()
    inter = check(l, subset)
    if (inter[0] == 0) or (inter[0] == 1 and inter[1] in verts):
        next = delete_leaf(l, graph)
        if next:
            leafs_delation(subset, next, graph)
    else:
        return;
```

```
28
29
30 def delete_leaf(block, graph):
31     """
32     This function deletes leaf from adjacency list
33     """
34     art_vertx = list(graph[block])[0]
35     graph.pop(block)
36     graph[art_vertx].remove(block)
37
38     # deletes cut vertex from adj list if its a leaf
39     if len(graph[art_vertx]) > 1:
40         return None;
41     else:
42         block = list(graph[art_vertx])[0]
43         graph.pop(art_vertx)
44         graph[block].remove(art_vertx)
45         if len(graph[block])>1:
46             return None;
47         else:
48             return block
49
50 def check(block, subset):
51     """
52     This function finds intersection of block and subset
53     """
54     counter = 0
55     art_vert = 0
56     for i in block:
57         if i in subset and counter<2:
58             art_vert = (i,)
59             counter += 1
60     return [counter, art_vert]
```

- Приклад використання алгоритму.

```
1 s = {5,0}
2 find_interval(s, g)
```

Алгоритм знаходження $c(G)$

- Алгоритм знаходження $c(G)$.

```
def convexity_number(g):
    """
    This function returns convexity number of graph G
    """
    verts = g.vertexes()
    if len(verts) == 2:
        return 1

    bct = BlockCutTree(g)
    if len(bct.bi_components) == 1:
        return 1

    verts = bct.graph.keys()

    #finds blocks-leafs of the block cut tree
    bct_leafs = [x for x in verts if len(x)>1 and len(bct.graph[x])
   ==1]
    minimal_leaf = len(reduce(lambda a, b: a if len(a)<len(b) else b,
    bct_leafs))

    return len(bct.preimage_vertexes) - minimal_leaf + 1
```

- Приклад використання алгоритму.

```
convexity_number(g)
```

Алгоритм знаходження $i(S)$ та $h(S)$

- Алгоритм знаходження $i(S)$, що співпадає із алгоритмом знаходження $h(S)$.

```
1
2  def interval_number(g):
3      """
4      This function returns interval number (hull number) of Graph G
5      """
6      verts = g.vertexes()
7      # trivial case
8      if len(verts) == 1:
9          return 1
10
11     bct = BlockCutTree(g)
12     # trivial cases
13     if len(bct.bi_components) == 1 or len(verts) == 2:
14         return 2
15
16     # interval number of G = number of leafs of block cut tree of G
17     return len([x for x in bct.graph.keys() if len(x)>1 and len(bct.
    graph[x])==1])
```

- Приклад використання алгоритму.

```
1 interval_number(g)
```