Модифікація аналітичного гамма-алгоритму пласкої укладки графа

Богдан Вікторович Гребенюк

Криворізький національний університет,
вул. Віталія Матусевича, 11, м. Кривий Ріг, 50027, Україна
bogdan020699@gmail.com

Анотація. Планарність графів – це один із ключових розділів теорії графів. Хоча граф є абстрактним математичним об'єктом, найчастіше саме візуалізація графа спрощує вивчення або розробку у певній сфері, наприклад, інфраструктури міста, менеджменту компанії або веб-сторінки сайту. Взагалі у вигляді графа можна зобразити будь-які структури, що мають зв'язки між елементами. Але часто подібні структури збільшуються до таких розмірів, що важко визначити, чи можливо представити їх на площині без перетину зв'язків. Існує багато алгоритмів, що вирішують це питання. Одним із таких є гамма-алгоритм. У статті визначені його проблеми та запропоновані методи їх вирішення, а також досліджені шляхи їх досягнення.

Ключові слова: дискретні структури, теорія графів, планарність.

Modification of the analytical gamma-algorithm for the flat layout of the graph

Bohdan V. Hrebeniuk

Kryvyi Rih National University, 11, Vitalii Matusevych St., Kryvyi Rih, 50027, Ukraine
bogdan020699@gmail.com

Abstract. The planarity of graphs is one of the key sections of graph theory. Although a graph is an abstract mathematical object, most often it is graph visualization that makes it easier to study or develop in a particular area, for example, the infrastructure of a city, a company's management or a website's web page. In general, in the form of a graph, it is possible to depict any structures that have connections between the elements. But often such structures grow to such dimensions that it is difficult to determine whether it is possible to represent them on a plane without intersecting the bonds. There are many algorithms that solve this issue. One of these is the gamma method. The article identifies its

CEUR-WS.org/Vol-2292/paper03.pdf

problems and suggests methods for solving them, and also examines ways to achieve them.

Keywords: discrete structures, graph theory, planarity.

1 Вступ

Розвиток теорії графів в основному зобов'язаний великій кількості застосувань. Мабуть, з усіх математичних об'єктів графи займають найбільш чільне місце в якості формальних моделей реальних систем.

Графи знайшли застосування практично у всіх галузях наукових знань: фізиці, біології, хімії, математиці, історії, лінгвістиці, соціальних науках, техніці та ін. Найбільшою популярністю теоретико-графові моделі користуються при дослідженні комунікаційних мереж, систем інформатики, хімічних [1] і генетичних [2] структур, електричних ланцюгів [3] та інших систем мережевої структури.

Планарність графів – це один з ключових розділів теорії графів. Хоча граф є абстрактним математичним об'єктом, найчастіше саме візуалізація графа спрощує вивчення або розробку у певній сфері, наприклад, інфраструктури міста, менеджменту компанії або веб-сторінки сайту. Взагалі у вигляді графа можна зобразити будь-які структури, що мають зв'язки між елементами. Але часто подібні структури збільшуються до таких розмірів, що важко визначити, чи можливо представити їх на площині без перетину зв'язків.

Існує багато алгоритмів, які визначають планарність графа. Відома формула Ейлера [4] описує плаский граф наступним чином:

$$|V(G)| - |E(G)| + |F(G)| = 2,$$

де G – граф, |V(G)| – кількість вершин, |E(G)| – кількість ребер, |F(G)| – кількість граней.

Ключовим словом тут є «визначення»: за допомогою формули визначається, чи можливо взагалі укласти граф. Призначення аналітичних алгоритмів – практична допомога в укладці графа.

На практиці аналітичні алгоритми являють собою послідовність перетворень графа, що призводять до його укладки. Одним з найпоширеніших алгоритмів є гамма-алгоритм [5]. Без сумніву, простота формулювання забезпечила йому популярність, адже невеликі розрахунки можна виконати вручну, а ключові моменти легко реалізувати на аркуші паперу. Однак, протилежним боком простоти стає обмеженість можливостей. Докладно всі проблеми оригінального гамма-алгоритму ми визначимо нижче, також ми запропонуємо ідеї та методи усунення цих проблем. Розглянемо покрокову реалізацію оригінального гамма-алгоритму.

Отже, на вхід алгоритму подається граф з наступними властивостями:

1. Граф зв'язний.
2. Граф містить хоча б один цикл.

3. Граф не має мостів.

Якщо порушена властивість 1, то граф потрібно укладати окремо за компонентами зв'язності. Якщо порушена властивість 2, то граф – це дерево, і зобразити його пласку укладку тривіально.

Докладніше розглянемо ситуацію, коли в графі G порушена властивість 3. У такому випадку спочатку потрібно видалити всі мости. Наступним кроком буде окрема укладка всіх компонент за схемою: укладемо одну компоненту зв'язності, а наступну компоненту, що пов'язана з першою мостом в графі G, малюватимемо в тій грані, де лежить вершина, що належить мосту. Інакше можлива ситуація, коли кінцева вершина моста буде знаходитися всередині плаского графа, а наступна компонента – зовні. Таким чином ми зможемо з'єднати мостом потрібні вершини. Надалі будемо використовувати цей метод для кожної нової компоненти.

Перший крок алгоритму – ініціалізація. Обираємо будь-який простий цикл в G, укладаємо його на площину і отримуємо дві грані: зовнішню і внутрішню. Уже укладену під час роботи алгоритму частину будемо позначати G_{plane}.

Другий крок алгоритму – спільний крок. Будується множина сегментів S. Кожен сегмент S відносно вже побудованого графа може бути одним з двох:

- ребро, обидва кінці якого належать G_{plane}, але саме воно йому не належить;
- зв'язна компонента G/G_{plane}, що доповнена усіма такими ребрами графа G , у яких один з кінців належить зв'язній компоненті, а другий належить графу G_{plane}.

Вершини, які одночасно належать G_{plane} і будь-якому сегменту, назвемо контактними вершинами.

Нехай грань Г вміщує сегмент S, якщо номери всіх контактних вершин S належать цій грані, $S \subset \Gamma$. Вочевидь, таких граней може бути декілька. Множину таких граней позначимо $\Gamma(S)$, а їх кількість – $|\Gamma(S)|$.

Отже, розглянемо всі сегменти S_i і для кожного визначимо кількість $|\Gamma(S_i)|$. Якщо знайдеться такий номер i, для якого $|\Gamma(S_i)| = 0$, то граф є непланарним, і алгоритм завершує роботу. Інакше вибираємо такий сегмент S_i, для якого кількість $|\Gamma(S_i)|$ найменша. Якщо таких сегментів декілька, то можна вибрати будь-який з них.

У сегменті знайдемо довільний ланцюг між двома контактними вершинами і укладемо його в будь-яку з граней множини $\Gamma(S)$. При цьому дана грань розіб'ється на дві. Вже укладена частина графа G' після укладки ланцюга збільшиться, а сегмент, з якого вийнято ланцюг, зникне або розпадеться на менші з новими контактними вершинами, що ведуть до вершин G'.

Тепер необхідно повторювати укладку сегментів доти, поки ми не використаємо всі сегменти, або поки не буде отримана відповідь, що граф непланарний.

2 Проблеми традиційного гамма-алгоритму

Під час використання гамма-алгоритму було виявлено наступні недоліки:

1. Алгоритм не намагається мінімізувати перетини, а шукає лише «ідеальний» варіант.

Алгоритм, зіткнувшись із сегментом, який нікуди не вдається укласти, одразу припиняє свою роботу. Однак, практика показує, що часто сегменти не піддаються укладці через якийсь один ланцюг. Якщо такий ланцюг пропустити – визнати, що в цей момент ми не можемо його укласти, і просто відтермінувати рішення – граф можна буде укладати далі. Потрібно таку можливість реалізувати, оскільки здебільшого цей алгоритм нечасто використовується саме через цей недолік.

2. Алгоритм не вміє укладати граф різними конфігураціями.

В алгоритмі в якості сегмента використовується доволі «скута» модель – ціла компонента зв'язності. При цьому ланцюги, які ми знаходимо в компоненті, мають бути укладені лише в ті грані, в які дозволяє ця компонента, *хоча може існувати багато граней, що підходять для ланцюга, але не підходять для компоненти*. Граф, побудований за принципом оригінального алгоритму, матиме, в кращому випадку, одну конфігурацію, тоді як в реальній ситуації можуть існувати багато зауважень, що роблять поточну конфігурацію неприйнятною. Необхідна наявність можливості генерувати багато конфігурацій укладки графа.

3. Алгоритм не підтримує графи, що складаються з декількох компонент зв'язності та графи, які мають мости/точки зчленування.

У сучасному світі **час** є одним із пріоритетних ресурсів у розробці/вивченні певної сфери діяльності. Маючи масивну структуру з тисячі елементів, чи буде проектувальник ліній електропередач розбиратися, де мости, де точки зчленування, де компоненти зв'язності? Завдання – спроектувати ефективно, вигідно, у найкоротший термін. Навіть якщо розкласти граф на компоненти зв'язності, розбити мости, розділити точки зчленування і отримати множину укладених (а може і неукладених) компонент, як зібрати їх докупи? До речі, алгоритм дає хибну інформацію при роботі з такими графами, тобто доопрацювання його просто **необхідне**.

3 Запропоновані варіанти вирішення проблем

Щоб досягти мети, яку ми перед собою поставили, необхідно здійснити найбільш глобальну зміну. Як раніше було відзначено, представлення сегмента у вигляді компоненти зв'язності позбавляє алгоритм гнучкості. У сегменті можуть бути ланцюги, які неможливо укласти в деякі грані через те, що компонента забороняє

це робити. Звідси випливає висновок – компоненту потрібно скасувати, замінивши на щось інше. У запропонованій нами реалізації використано наступне рішення даної проблеми: **замінити компоненту зв'язності множиною всіх відсутніх ланцюгів**. При цьому ми застосовуємо однакові дії до наших «нових» і до «старих» сегментів (так само рахуємо кількість граней, куди ланцюг може бути укладений, все так же використовуємо ланцюг з мінімальною кількістю граней і т. ін.). Отже, з цією зміною ми матимемо:

1. Інформацію про кількість придатних для укладки граней для кожного ланцюга, тоді як компонента давала інформацію лише про спільні грані ланцюгів.
2. Множину унікальних конфігурацій графа шляхом перебору граней для укладки ланцюга.
3. Множину не унікальних конфігурацій графа шляхом перебору ланцюгів.
4. Можливість позначати ланцюг як «неможливий для укладки» для конкретної конфігурації і продовжувати алгоритм без нього.

За допомогою такої модифікації ми вирішили пункт 2 у нашому переліку проблем і частково пункт 1 – використовуючи ланцюги в ролі сегмента ми «розв'язали руки» нашого алгоритму. Тепер ми можемо укладати що хочемо і куди хочемо, а також спокійно відкладати сегменти, які не вдається укласти.

Що стосується пункту 3, то оригінальний алгоритм пропонує наступні дії з графом, у якому є мости: «... спочатку всі мости необхідно видалити, надалі виконати окрему укладку всіх компонент наступним чином: укладемо одну компоненту зв'язності, а наступну компоненту, пов'язану з першою в графі G мостом, будемо малювати в тій грані, в якій лежить вершина, що належить мосту».

Перш за все, необхідно зробити виправлення в умові: якщо є мости і **точки зчленування**. Незрозуміло, чому в алгоритмі немає ані слова про них, адже вони можуть привести алгоритм в таке саме невизначене становище, як і мости. Також необхідно дотримуватися певного порядку укладки компонент і слідувати певним правилам. Сформулюємо їх в наступному розділі.

4 Модифікований гамма-алгоритм

Отже, на вхід алгоритму подається будь-який граф. Після цього необхідно виконати наступні дії:

1. Знайти всі компоненти зв'язності.
2. У кожній з компонент знайти точки зчленування і мости.
3. Якщо такі є, розбити компоненту наступним чином (рис. 1).

На рис. 1 компонента {1, 2, 3, 4, 5} має точку зчленування – вершину 3. Після поділу у нас вийшло дві компоненти, які поділяють спільну вершину. Те ж саме відбувається і з компонентою, яка має міст (рис. 2).

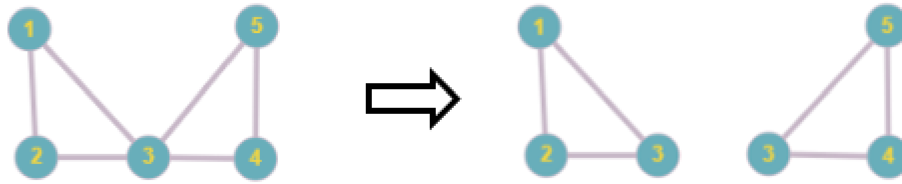

Рис. 1. Поділ по точці зчленування

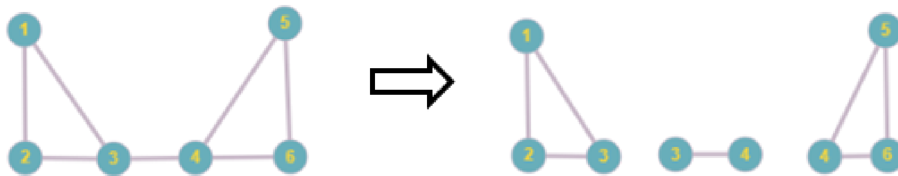

Рис. 2. Поділ по мосту

Також потрібно відзначити, що наступний граф розіб’ється в такий спосіб (рис. 3):

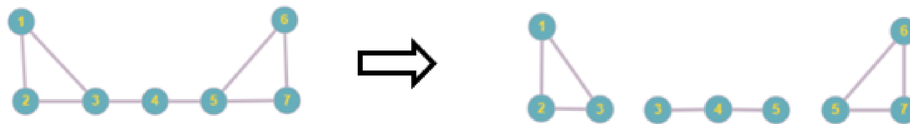

Рис. 3. Поділ компонентів

Хоча у даного графа є 3 точки зчленування – {3, 4, 5}, ми можемо їх об’єднати в дерево. Тобто фактично нам необхідно знайти всі подібні дерева в графі і «роз’єднати» компоненту.

4. Після того, як були знайдені всі компоненти зв’язності і проведено їх роз’єднання, необхідно укласти їх у певному порядку.

Ми пропонуємо наступний алгоритм укладання компонент.

a. Укласти деяку компоненту зв’язності C.
b. Серед компонент, що залишилися, визначити «зв’язані» (не плутати зі зв’язними компонентами), тобто ті, які мають спільні вершини з компонентою C.
c. Якщо таких немає, повернутися до 1 пункту.
d. Інакше кожну зі знайдених компонент необхідно укласти так, щоб спільна вершина мала доступ до зовнішньої грані. Після цього ми зможемо укласти побудовані нами компоненти у відповідні грані, де знаходяться спільні вершини.

e. Для кожної зі знайдених компонент знов визначити неукладені компоненти, що мають спільні вершини. Повторювати пункти 4-5 доти, поки не вичерпаємо всі зв'язані компоненти.
f. Якщо все зв'язані компоненти укладені, але не всі компоненти вичерпані, перейти до пункту 1.

Зупинимося на моменті вибору «початкової» компоненти C. Існує багато критеріїв, за якими можна оцінювати оптимальний варіант: це може бути компонента з найбільшою кількістю вершин, з найбільшою кількістю ребер, з найбільшою кількістю загальних вершин і т. ін. У загальному випадку краще вибирати найбільш «складну», наймасивнішу компоненту, тому що при першому укладанні не існує ніяких обмежень, отже, можна генерувати **всі можливі конфігурації** даної компоненти. Коли б така компонента укладалася не першою, ми могли б отримувати лише ті конфігурації, у яких загальна вершина має доступ до зовнішньої грані. У нашій реалізації в якості критерія ми будемо використовувати кількість вершин компоненти (дерева не братимуть участі у розрахунках). Але дуже легко обрати інший критерій пошуку, змінивши буквально один рядок коду.

5. Отже, перейдемо безпосередньо до укладки компоненти. Потрібно:
   a. Ініціалізувати будь-який простий цикл в компоненті.
   b. Якщо такого немає, то компонента – дерево, і укладка її не потребує особливих зусиль.
   c. Інакше знаходимо всі відсутні ланцюги компоненти. Ви можете помітити, що деякі ребра повторюються в різних ланцюгах. При укладанні кожного ланцюга необхідно буде позбавлятися від уже укладених ребер. Маючи цикл і сегменти, можна почати генерацію конфігурацій. Генерувати ми будемо рекурсивно, тому для простоти пояснень за допомогою псевдокоду коротко, **поверхово** опишемо цю функцію.

```
function RecurSearch(cycles, segments)
  if segments.length == 0 then
    someComplexThings()
    return 1
  facesOfSegments ← getFacesOfSegments(cycles, segments)
  minSegment ← getMinSegment(facesOfSegments)
  if minSegment == 0 then
    anotherComplexThings()
    return 0
  For each segment in facesOfSegments with length == minSegment
do
    For each face in facesOfSegments[segment] do
      Data ← getPreparedData(cycles, segments, segment, face)
      if RecurSearch(Data) != 0 do
        return 1
  return 0
```

де:

- **cycles** – відображення, що характеризує укладку, елементи якої утворюють пари «цикл»: «ланцюг»;
- **segments** – послідовність ще не укладених ланцюгів;
- **getFacesofSegments** – функція, яка повертає відображення, в якому елементи утворюють пари «ланцюг»: «послідовність циклів» (цикл у контексті графа);
- **getMinSegment** – функція, яка повертає число, яке є мінімальним з кількості граней, що підходять для укладки;
- **getPreparedData** – функція, що виконує укладку сегмента, розбиття грані і т. ін.;
- **someComplexThings**, **anotherComplexThings** – деякі дії, до яких ми ще не готові, але поки ми про них і не повинні замислюватися.

Насправді, навіть з таким поверхневим оглядом ми вже досить глибоко занурилися в деталі реалізації алгоритму, тому давайте словесно його опишемо. На кожному кроці рекурсії ми для кожного ланцюга знаходимо такі грані, куди ланцюг можна укласти. Знаходимо ланцюги, для яких кількість таких граней **однакова** і **мінімальна**. Тепер запускаємо рекурсивний пошук у кожну грань кожного ланцюга, перед цим розділивши грань, уклавши сегмент і т. ін. Тут також необхідно позбутися від уже укладених ребер, які ще можуть бути в ланцюгах.

Зверніть увагу, що при такому підході ми будемо отримувати результати, які з першого погляду можуть здатися однаковими. Відрізнятися вони будуть лише порядком укладки сегментів у грані, що теж іноді буває важливим. Щоб генерувати тільки унікальні конфігурації, необхідно запускати рекурсивний пошук у кожну грань тільки для одного «мінімального» сегмента.

Якщо є деякі ланцюги, що неможливо укласти в жодну грань, потрібно пропустити їх. Фактично це ми і будемо робити в функції **anotherComplexThings**, але іншим шляхом, про який ми поговоримо вже у самій реалізації. Тобто при кожному виклику рекурсії ланцюгів ставатиме все менше і менше, і в кінцевому результаті алгоритм їх повністю вичерпає.

Як тільки будуть вичерпані всі сегменти (а це означає, що алгоритм здійснив деяку укладку графа), необхідно перевірити, чи відповідає укладена компонента умовам, які зазначені в пункті 4. Для цього призначена функція someComplexThings. Якщо дана конфігурація пройшла всі перевірки – компонента успішно укладена.

Повний опис алгоритму та програмна реалізація мовою Python подані у [6].

Список використаних джерел

1. Семеріков С. О. Застосування методів машинного навчання у навчанні моделювання майбутніх учителів хімії / С. О. Семеріков // Технології навчання хімії у школі та ЗВО : збірник тез доповідей Всеукраїнської науково-практичної Інтернет-конференції / заг. ред. Т. В. Старова (вид. 1-е). – Кривий Ріг : КДПУ, 2018. – С. 10-19.

2. Komarova O. V. Computer Simulation of Biological Processes at the High School [Electronic resource] / Olena V. Komarova, Albert A. Azaryan // Augmented Reality in Education : Proceedings of the 1st International Workshop (AREdu 2018). Kryvyi Rih, Ukraine, October 2, 2018 / Edited by : Arnold E. Kiv, Vladimir N. Soloviev. – P. 24-32. – (CEUR Workshop Proceedings (CEUR-WS.org), Vol. 2257). – Access mode : http://ceur-ws.org/Vol-2257/paper03.pdf
3. Modlo Ye. O. Modernization of Professional Training of Electromechanics Bachelors: ICT-based Competence Approach [Electronic resource] / Yevhenii O. Modlo, Serhiy O. Semerikov, Ekaterina O. Shmeltzer // Augmented Reality in Education : Proceedings of the 1st International Workshop (AREdu 2018). Kryvyi Rih, Ukraine, October 2, 2018 / Edited by : Arnold E. Kiv, Vladimir N. Soloviev. – P. 148-172. – (CEUR Workshop Proceedings (CEUR-WS.org), Vol. 2257). – Access mode : http://ceur-ws.org/Vol-2257/paper15.pdf
4. Euler L. Solutio problematis ad geometriam situs pertinentis / Euler L. // Commentarii Acad. Sci. Imp. Petrop., 8, 1736. – P. 128-140.
5. Иринёв А. Алгоритм плоской укладки графов [Электронный ресурс] / Иринёв Антон, Каширин Виктор. – [2006]. – 10 с. – Режим доступа : http://rain.ifmo.ru/cat/data/theory/graph-coloring-layout/layout-2006/article.pdf.
6. Гребенюк Б. В. gamma [Электронный ресурс] / [Б. В. Гребенюк]. – 2018. – Режим доступа : https://github.com/BogdanGrebenuk/gamma.

References (translated and transliterated)

1. Semerikov, S.O.: Zastosuvannia metodiv mashynnoho navchannia u navchanni modeliuvannia maibutnikh uchyteliv khimii (Application of machine learning methods in teaching simulation of future chemistry teachers). In: Starova, T.V. (ed.) Technologies of teaching chemistry in school and university, All-Ukrainian scientific and practical Internet conference, Kryvyi Rih, November 30, 2018, pp. 10–19. KDPU, Kryvyi Rih (2018)
2. Komarova, O.V., Azaryan, A.A.: Computer Simulation of Biological Processes at the High School. In: Kiv, A.E., Soloviev, V.N. (eds.) Proceedings of the 1st International Workshop on Augmented Reality in Education (AREdu 2018), Kryvyi Rih, Ukraine, October 2, 2018. CEUR Workshop Proceedings, vol. 2257, pp. 24–32. http://ceur-ws.org/Vol-2257/paper03.pdf (2018). Accessed 30 Nov 2018
3. Modlo, Ye.O., Semerikov, S.O., Shmeltzer, E.O.: Modernization of Professional Training of Electromechanics Bachelors: ICT-based Competence Approach. . In: Kiv, A.E., Soloviev, V.N. (eds.) Proceedings of the 1st International Workshop on Augmented Reality in Education (AREdu 2018), Kryvyi Rih, Ukraine, October 2, 2018. CEUR Workshop Proceedings, vol. 2257, pp. 148–172. http://ceur-ws.org/Vol-2257/paper15.pdf (2018). Accessed 30 Nov 2018
4. Euler, L.: Solutio problematis ad geometriam situs pertinentis. Commentarii Acad. Sci. Imp. Petrop., **8**, 128–140 (1736)
5. Irinev, A., Kashirin, V.: Algoritm ploskoi ukladki grafov (Algorithm for flat styling graphs). http://rain.ifmo.ru/cat/data/theory/graph-coloring-layout/layout-2006/article.pdf (2006). Accessed 1 Nov 2018
6. Hrebeniuk, B.V.: gamma. https://github.com/BogdanGrebenuk/gamma (2018). Accessed 17 Nov 2018