Міністерство освіти і науки України

Департамент освіти і науки Дніпропетровської облдержадміністрації

Дніпропетровське територіальне відділення МАН України

Відділення: комп’ютерні науки

Секція: Мультимедійні системи, навчальні та ігрові програми

СТВОРЕННЯ ТА ВИКОРИСТАННЯ НАОЧНОГО МЕТОДУ ПРЕДСТАВЛЕННЯ ГРАФА

Роботу виконав:   
Караваєв Іван Олександрович,

учень 11 класу КЗО «Дніпровський ліцей інформаційних технологій при ДНУ»

Науковий керівник:

Ентін Йосиф Абрамович,

вчитель інформатики КЗО «Дніпровський ліцей інформаційних технологій при ДНУ»

Дніпро-2020

ЗМІСТ

[ВСТУП 3](#_Toc30659426)

[РОЗДІЛ 1 4](#_Toc30659427)

[ПОСТАНОВКА ЗАДАЧІ ТА АНАЛІЗ ПРЕДМЕТНОЇ ТА ОБ’ЄКТНОЇ ЧАСТИН 4](#_Toc30659428)

[1.1 Постановка задачі 4](#_Toc30659429)

[1.2 Теорія графів. Сфери застосування 5](#_Toc30659430)

[1.3 Існуючі методи подання графів 8](#_Toc30659431)

[1.3.1 Матриця суміжності 8](#_Toc30659432)

[1.3.2 Список ребер 9](#_Toc30659433)

[1.3.3 Список суміжності 9](#_Toc30659434)

[РОЗДІЛ 2](#_Toc30659435) [СПИСОК СТЕКІВ. ОПИС ТА РОЗРОБКА 11](#_Toc30659436)

[2.1 Знайомство з пошуком у глибину (DFS) 11](#_Toc30659437)

[2.2 Новий спосіб збереження графа у пам’яті 12](#_Toc30659438)

[2.2.1 Побудова 12](#_Toc30659439)

[2.2.2 Оптимізація 13](#_Toc30659440)

[2.3 Узагальнення методу 15](#_Toc30659441)

[2.3.1 Наявність циклів 15](#_Toc30659442)

[2.3.2 Незв’язний граф 15](#_Toc30659443)

[2.4 Переваги, недоліки та застосування 16](#_Toc30659444)

[2.4.1 Дерево стеків 16](#_Toc30659445)

[РОЗДІЛ 3](#_Toc30659446) [РОЗРОБКА ПРОГРАМИ 19](#_Toc30659447)

[3.1 Вибір мови програмування і технології розробки 19](#_Toc30659448)

[3.2 Інтерфейс програми 20](#_Toc30659449)

[ВИСНОВКИ 22](#_Toc30659450)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 23](#_Toc30659451)

[ДОДАТКИ 24](#_Toc30659452)

[ДОДАТОК А 24](#_Toc30659453)

[ДОДАТОК А-1 24](#_Toc30659454)

[ДОДАТОК А-2 25](#_Toc30659455)

[ДОДАТОК А-3 27](#_Toc30659456)

[ДОДАТОК B 30](#_Toc30659457)

[ДОДАТОК B-1 30](#_Toc30659458)

[ДОДАТОК B-2 35](#_Toc30659459)

# ВСТУП

XXI століття характеризується небувалим зростанням інтенсивності інформаційних потоків, що стосується більшості сфер діяльності людини. Необхідність успішно діяти в цих умовах робить актуальними: 1) вміння аналізувати та фільтрувати інформацію; 2) підвищення наочності її представлення.

Тому **мета роботи** - розробка, опис та дослідження альтернативного способу зберігання інформації про граф у пам’яті, який збільшує наочність цієї інформації для користувача і спрощує доступ до елементів графа. Аби бути послідовним, я вирішив створити навчальну програму, яка ознайомить користувача із сферою мого дослідження, вказуватиме на переваги представленого в роботі методу та його недоліки порівняно з іншими методами. Отже, проект є своєрідним провідником між хаотичним поданням інформації та її послідовним, збалансованим за обсягом представленням.

**Об’єктом дослідження** є граф: сукупність певних об’єктів зі зв’язками між ними.

**Предмет дослідження** – різні способи представлення графа.

Проект можна використати як навчальний посібник із метою ознайомлення користувача з концепцією теорії графів, перевагою і особливістю якого є унікальність розглянутих теорій та максимальне спрощення подання інформації.

# РОЗДІЛ 1

# ПОСТАНОВКА ЗАДАЧІ ТА АНАЛІЗ ПРЕДМЕТНОЇ ТА ОБ’ЄКТНОЇ ЧАСТИН

## Постановка задачі

Було поставлено задачі:

* 1. описати базові поняття теорії графів; дослідити існуючі способи збереження графа у пам’яті;
  2. розробити новий метод (список стеків) та алгоритм його побудови;
  3. виокремити його переваги та недоліки;
  4. розробити алгоритм побудови дерева на основі списку стеків;
  5. створити навчальну програму, яка візуалізує вищеперераховане, дозволяє детально ознайомитись з методом та надає користувачу такий функціонал:

1. можливість будувати граф випадково або власноруч, завантажувати та зберігати дані до текстового файлу;
2. покрокове моделювання побудови списку стеків з можливістю зворотного руху по його фазах;
3. візуалізація побудови дерева за допомогою описаного методу та можливість зберегти побудову у форматі JPG або PNG.

## 1.2 Теорія графів. Сфери застосування

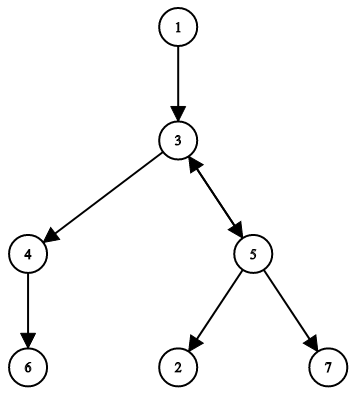
Теорія графів – розділ математики, що вивчає властивості графів. У його термінах можна сформулювати багато задач, пов’язаних із дискретними об’єктами, тобто такими об’єктами, складові яких чітко відділені одна від одної.

Теорія графів являє собою дуже зручну мову для опису програмних та багатьох інших моделей. Її струнка система просторових термінів та означень дозволяє просто і доступно описувати складні та тонкі речі.

Граф – сукупність об’єктів та зв’язків між ними. Об’єктами є вершини (вузли) графу, зв’язками – ребра (в неорієнтованому графі) або дуги (в орієнтованому). На площині вершини є точками, ребра та дуги – відрізками. В неорієнтованому графі дозволено рух по ребрах в обидва боки, в орієнтованом, рух по дузі можливий тільки в один бік.

Графом називається об’єкт, який заданий парою множин , де -множина вершин, – множина ребер.

Іноді, цей математичний об’єкт називають топологічним, тобто його властивості не змінюються при розтягуванні, стисненні та викривленні.

Изображение выглядит как объект, фотография, внутренний, стена

Автоматически созданное описание Рис 1.1. Приклад неорієнтованого графа, що містить цикли.

Изображение выглядит как объект, часы

Автоматически созданное описаниеРис 1.2. Приклад орієнтованого графа

Рис 1.3 Приклад зваженого графу.

Деревом називається неорієнтований зв’язний граф без петель та циклів.

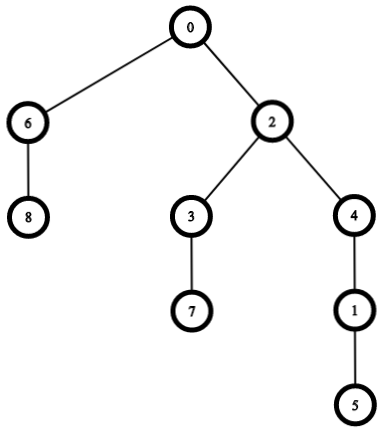
* Граф називається зв’язним тільки тоді і тоді, коли існує така послідовність вершин, що , а також та . Тобто із будь-якої вершини графа можна дістатися іншої.
* Граф містить петлю тільки тоді і тоді, коли . Тобто граф містить ребро, яке сполучає вершину саму із собою.
* Граф містить цикл тільки тоді і тоді, коли існує така послідовність вершин, що , а також та . Тобто існує така вершина, почавши обхід з якої, ми знову потрапимо до неї.

Рис 1.4 Зображення дерева

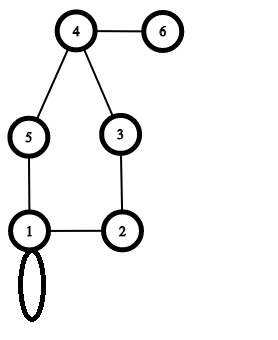
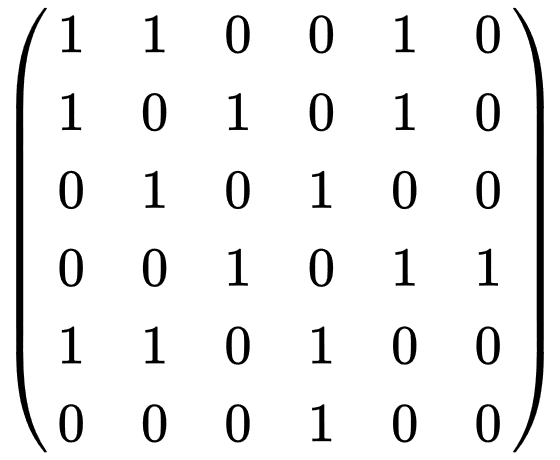
Листом дерева називається вершина, що не є коренем та має рівно одного сусіда. На рис 1.4 такими вершинами є 5, 7, 8. Коренем дерева може бути будь-яка вершина.

## 1.3 Існуючі методи подання графів

### 1.3.1 Матриця суміжності

Матрицею суміжності графа зі скінченною кількістю вершин , які пронумеровані від до , називається така матриця розміру , в якій значення елементу дорівнює одиниці або нулю, в залежності від існування ребра між вершинами .

Використання матриці суміжності є актуальним здебільшого у випадку нерозріджених графів, коли кількість ребер набагато більша за кількість вузлів. Це пов’язано з тим, що на утримання матриці суміжності виділяється пам’яті, що у випадку розріджених графів менш вигідно, аніж розглянуті нижче альтернативи.

**Крім цього, перевагами цього методу є швидкий доступ до інформації, щодо існування ребра між деякою парою вершин графа та легке визначення його типу.

А Б

Рис.1.5. А. Матриця суміжності.

Б. Зображення графу, якому відповідає матриця суміжності А.

Але з іншого боку представлення графа матрицею суміжності є незручним для сприйняття людським оком. Більш асоціативно зрозумілим можна вважати список суміжності*.*

### 1.3.2 Список ребер

Списком ребер називається подання графа у вигляді деякої множини його ребер. Наприклад, для графа на рис. 1.8 список ребер виглядатиме таким чином:

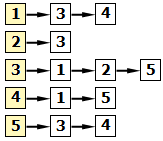
(1, 1); (1, 2); (1, 5); (2; 3); (3; 4); (4; 5); (4; 6);

На його підтримку необхідно виділити пам’яті, де кількість ребер. Подання графа цим методом є зручним, якщо наприклад, необхідно впорядкувати і дослідити ребра.

Зауважимо, що цей спосіб також не дозволяє людині швидко знайти взаємозв’язки між даними. А це в свою чергу ускладнює побудову та роботу алгоритмів графічної візуалізації графа представленого списком ребер.

### 1.3.3 Список суміжності

Изображение выглядит как объект

Автоматически созданное описаниеСписком суміжності графа є масив А, розрахований на элементів, та комірка якого містить деякий список вершин, які є суміжними з вершиною .

A Б

Рис.1.6. А. Список суміжності.

Б. Зображення графу, якому відповідає список суміжності А.

Цей метод можна вважати найзручнішим, якщо вам потрібно якимось чином обходити граф. У найпопулярніших і найефективніших реалізаціях пошуку у ширину та глибину застосовують список суміжності.

Саме список суміжності є базовим методом збереження графа у моїх подальших дослідженнях.

– кількість пам’яті, необхідної задля збереження списку суміжності деякого графа

# РОЗДІЛ 2

# СПИСОК СТЕКІВ. ОПИС ТА РОЗРОБКА

## 2.1 Знайомство з пошуком у глибину (DFS)

Алгоритм пошуку у глибину – з англійської Depth-first search ( DFS ) – виконується для пересування між послідовними вершинами графа, сполученими ребрами. Щоб не повторювати пересування по раніше пройдених вершинах, вони помічаються.

Для його реалізації застосовується структура даних стек, яка працює за принципом LIFO - last in first out, тобто “останнім зайшов – першим вийшов”. У багатьох мовах програмування стек є частиною стандартної бібліотеки.

Розглянемо покрокову реалізацію DFS. Це є важливо, бо саме на основі пошуку у глибину базується мій метод збереження графа у пам’яті.

1. Почати з довільної вершини нехай ця вершина є 0. Додати її до стеку.
2. Розглянути вершину у верхівці стеку та інші вершини у які можна потрапити із неї (Граф представлений списком суміжності). Якщо всі такі вершини розглянуто – перейти до кроку 3), інакше до кроку 4).
3. Перейти до деякої вершини графа. Якщо вона не є поміченою, то кладемо її у стек та позначаємо як розглянуту. Повернутись до кроку 2).
4. Видалити верхню вершину. Якщо стек після цього порожній – зупинитись, інакше перейти до кроку 2).

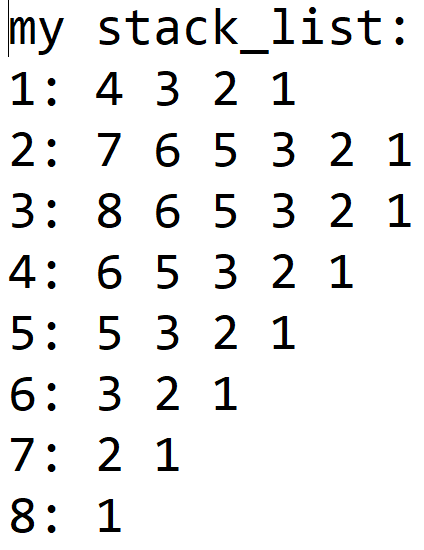
Існує також рекурсивний алгоритм пошуку у глибину, реалізація якого не потребує працювати із стеком за рахунок того, що послідовність виклику процедури dfs реалізує той самий функціонал. У [додатку А-1](#_ДОДАТОК_А-1) наведено і пояснено програмний код мовою програмування С#.

## 2.2 Новий спосіб збереження графа у пам’яті

Розглянемо дещо інший спосіб зберігання неорієнтовного графа. Назвемо його «список стеків».

### 2.2.1 Побудова

Візьмемо за основу рекурсивний алгоритм пошуку у глибину. Розглянемо граф, що за своїм типом є деревом.

Изображение выглядит как объект, часы

Автоматически созданное описаниеОтже, під час деякої ітерації по дереву, нас цікавитимуть вершини та їх порядок, які на той час будуть знаходитись у стеці рекурсивних викликів. Найзручнішим в даному випадку для нас буде використання списку ребер як вхідні дані про граф та списку суміжності, в якості даних для DFS.

Рис.2.1-А. Зображення дерева Рис.2.1-Б. Список стеків дерева А.

Стеки зберігаються у пам’яті у вигляді списків, де найлівіший елемент відповідає верхівці стеку.

**DFS-стеком** будемо називати стек з інформацією про незавершений пошук у глибину.

Кількість стеків завжди дорівнює кількості вершин. Це пов’язано з тим, що DFS-стек зберігається у момент, коли алгоритм дістався листа у дереві рекурсії.

Звідси також випливає, що верхні елементи всіх стеків утворюють перестановку з чисел від 1 до n, де n – кількість вершин дерева. Це можна побачити на рис 2.1-Б.

За списком стеків можна легко визначити кореневу вершину, вона зустрічається у кожному стеці та є найправішою у списку, що характеризує цей стек.

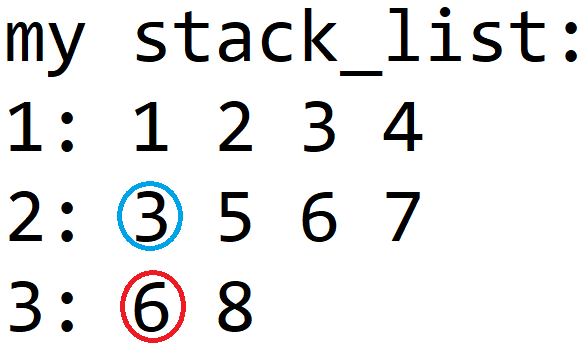
Асимптотика алгоритму у найгіршому випадку – O(), тому є необхідність оптимізувати метод до лінійної складності.

### 2.2.2 Оптимізація

Оптимізаційний метод також представлений у вигляді списку списків, але тепер DFS-стеки записуються справа на ліво, тобто верхівка стеку співвідноситься з найправішим елементом списку. На рис. 2.2-Б верхівкою першого записаного DFS-стеку є 4 - останній (найправіший) елемент списку.

Проаналізувавши (рис. 2.1-Б), можна побачити, що багато чисел та їх послідовностей повторюються і заважають пошуку бажаних закономірностей при співставленні інформації з різних списків.

Изображение выглядит как объект

Автоматически созданное описаниеРозглянемо дерево (рис. 2.2-А) та пояснимо, як працює оптимізаційний метод. (його реалізація мовою програмування С# наведена у [додатку А-2](#_ДОДАТОК_А-2)).

А Б

Рис.2.2. А. Зображення дерева. Б. Оптимізований список стеків дерева А.

Маємо два оптимізаційні моменти.

1. *Аби не зберігати одні і ті ж дані у кожному з n списків, залишимо у списку лише ті DFS-cтеки, де верхній елемент є листом дерева.*

Назвемо вершину вузловою, тоді і тільки тоді, коли під час виконання рекурсивної процедури, що стартує з вершини існує хоча б одне ребро, що сполучає вершину та ще не помічену вершину . На рис 2.2(Б) такими є вершини 3 та 6, вони виділяються на зображенні синім та червоним контурами.

Іншими словами, **вузлова вершина** – елемент списку, що зустрічається у попередньо збережених стеках та слугує зв’язком між ними.

Початкову вершину не вважаємо вузловою, тому кожен список, починаючи з другого, має рівно одну вузлову вершину, а перший список не має таких вершин.

Зазначимо, що кожна вершина може бути вузловою декілька разів. Це пов’язано з тим, що з нею можуть бути сполучені ще кілька вершин.

1. *Після виходу з вузлової вершини видаляємо всі складові DFS-стеку, крім цієї вершини. Таким чином, ми не дублюємо інформацію, утворену вершинами, котрі вже оброблені, при цьому зберігаючи зв’язок між даними окремих списків.*

Можна побачити, що кількість збережених DFS-стеків відповідає кількості листових вершин у дереві, а сумарна кількість елементів у списку списків дорівнює кількість вершин та кількість вузлових вершин дерева. Кожен список, окрім першого, має вузлову вершину, отже асимптотична складність отриманого алгоритму є лінійною.

Маємо, що ключовою ідею оптимізації є збереження вузлових вершин, які позбавляють необхідності запису деякої інформації, за рахунок можливості її відновлення.

Представлення дерева в такому вигляді є зручним інструментом дослідження взаємозв’язків між його вершинами та шляхами.

## 2.3 Узагальнення методу

### 2.3.1 Наявність циклів

Розглянемо побудову списку стеків для графу, що не є ациклічним.

Наявність у графі циклу означає, що деякі ребра сполучають помічені вершини, а отже не впливають на побудову списку стеків, тому такі ребра видаляються з графа і зберігаються окремо, будемо називати їх циклічними.

У випадку нерозріджених графів це призводить до видалення більшої частини ребер, що означає втрату вагомої частини інформації про граф, що є суттєвим недоліком застосування методу до циклічних графів.

Але в залежності від контексту задачі схожий підхід може бути актуальним. Наприклад побудова мінімального каркасу для зваженого циклічного графа, тобто у випадку наявності пріоритетних ребер.

Зручно представляти циклічні ребра списком ребер або списком суміжності вершин, ребра між якими є циклічними.

Отже будь-який циклічний граф можна представити у вигляді ациклічного та деякої множини циклічних ребер. Тому надалі розглядатимемо лише ациклічні графи.

### 2.3.2 Незв’язний граф

Будь-який незв’язний граф можна розбити на компоненти зв’язності та представити їх окремо у вигляді списків стеків.

Так як таке розбиття є єдиним, то такий спосіб узагальнення є доцільним і необхідним.

**!** Узагальнення методу базується на побудові списку стеків для дерева, тому в подальшому будемо працювати з графами, які є деревами.

## 2.4 Переваги, недоліки та застосування

Проведені узагальнення у попередньому пункті дозволяють нам працювати з довільними неорієнтованими графами, але кожне з таких узагальнень зводиться до використання методу на деревах. Це пов’язано з тим, що список стеків будується на основі DFS.

Список стеків є просунутою версією списку ребер. Кожен збережений стек – це шлях від початкової або вузлової вершини до листа дерева.

Фішкою алгоритму є збереження вузлових вершин, за допомогою яких легко прослідкувати зв’язок між записаними DFS-стеками. Отже список стеків можна розглядати у вигляді дерева. Таке дерево будемо називати **деревом стеків**.

### 2.4.1 Дерево стеків

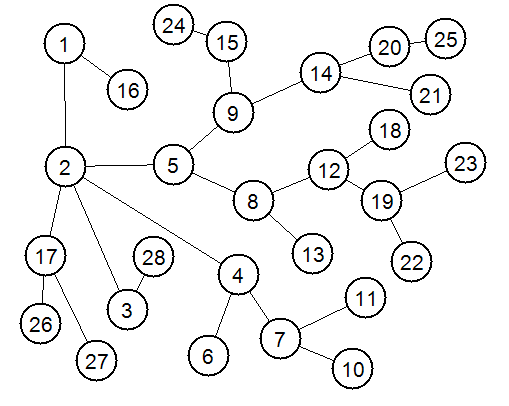
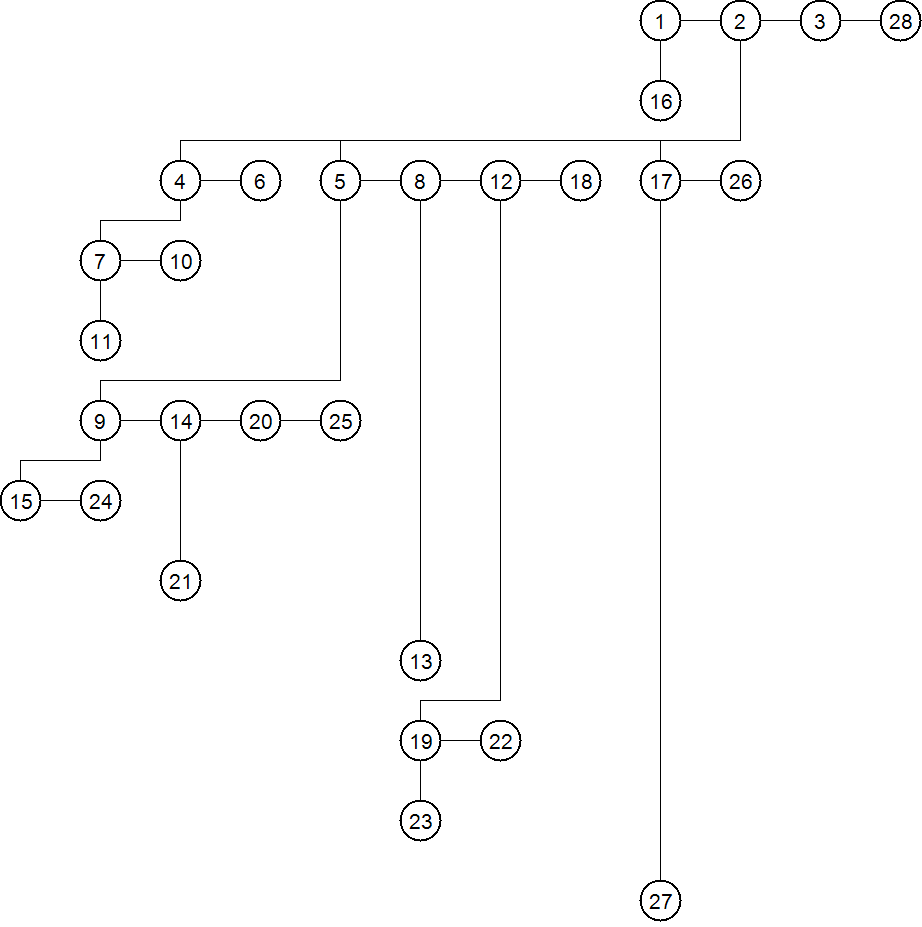
Розглянемо побудову та візуалізацію на прикладі деякого графа (рис 2.3).

Рис 2.3. Побудоване власноруч дерево за допомогою моєї програми

Рис 2.4. Дерево стеків на основі дерева на рис 2.3

Побудуємо список стеків на основі початкового дерева.

(Рис 2.4). Легко помітити, що деякі вершин утворюють горизонтальні ланцюги. Наприклад 1-2-3-28, 5-8-12-18, 9-14-20-25… більш того, кожна вершина належить рівно одному ланцюгу. Кожен з них є одним із збережених стеків у яких відсутня вузлова вершина, яка слугувала лише для побудови зв’язку між ними. При побудові, кожен з ланцюгів представляється у вигляді вершини, а ребрами слугують зв’язки між ними по вузловим вершинам.

Особливості цієї візуалізації є те, що пара ланцюгів знаходиться в одному рядку тільки тоді, коли вузлові вершини збережених стеків, що відповідають цим ланцюгам – співпадають. Внаслідок чого дерево видовжується, і незважаючи на свою інформативність є незручним у майбутньому використанні. Виправимо це!

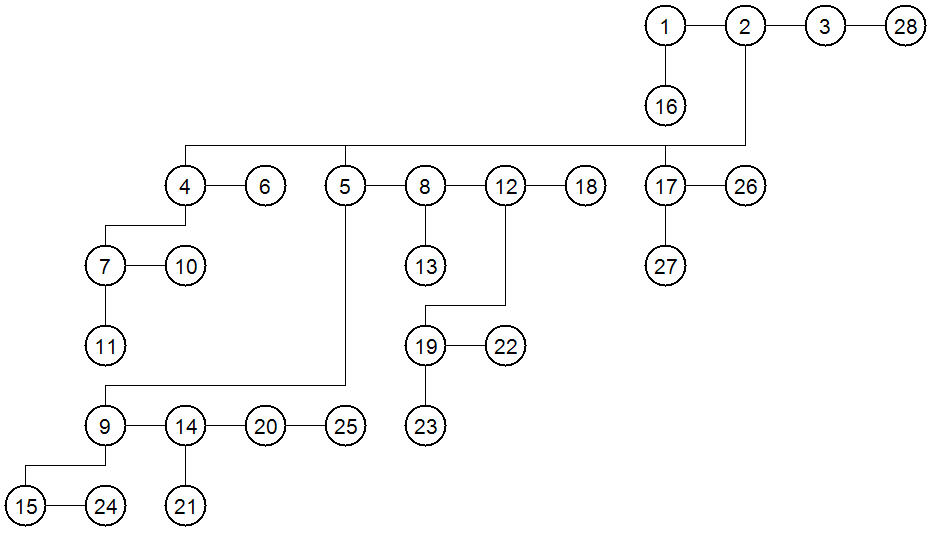


Рис 2.5. Стиснуте дерево стеків(на основі рис. 2.3 – 2.4).

Для того щоб “стиснути” дерево стеків я зберігав та працював з інформацією про найправіші вершини кожного рівня візуалізації.

У [додатку А-3](#_ДОДАТОК_А-3) представлена та детально описана рекурсивна частина побудови мовою програмування С#.

Особливості візуалізації дерева на основі списку стеків.

1. формування горизонтальних ланцюгів;
2. виражені зв’язки між ланцюгами, що дозволяють цілісно розглядати горизонтальні послідовності вершин;
3. ребра під кутом кратним ;
4. компактність.

Все це значно покращує наочність побудови, що дозволяє зробити висновки щодо структури дерева, степені його розгалуженості та загалом його особливостей.

Така побудова буде значно кращою за її альтернативи, особливо у випадку ручного аналізу дерева з великою кількістю вершин.

# РОЗДІЛ 3

# РОЗРОБКА ПРОГРАМИ

## 3.1 Вибір мови програмування і технології розробки

Складно уявити собі дослідження, пов’язане із наочним сприйняттям інформації, котре не передбачає створення графічної візуалізації об’єктів чи процесів.

Для створення програми було обрано технологію Windows Forms – інтерфейс програмування додатків, відповідальний за графічний інтерфейс користувача, який є частиною Microsoft.NET Framework.

Чому саме Windows Forms ?

1. Простота і зручність платформи, можливості якої повністю задовольняють мої потреби.
2. Здатність працювати за відсутності підключення до Інтернету.
3. Windows Forms використовує надійну систему безпеки платформи .NET Framework. Завдяки їй додатки є більш надійними, порівняно з іншими традиційними додатками Windows.

Середовище програмування: Visual Studio 2019.

Мова програмування: C#.

## Інтерфейс програми

Рис. 3.1 Меню програми

1. Тести.

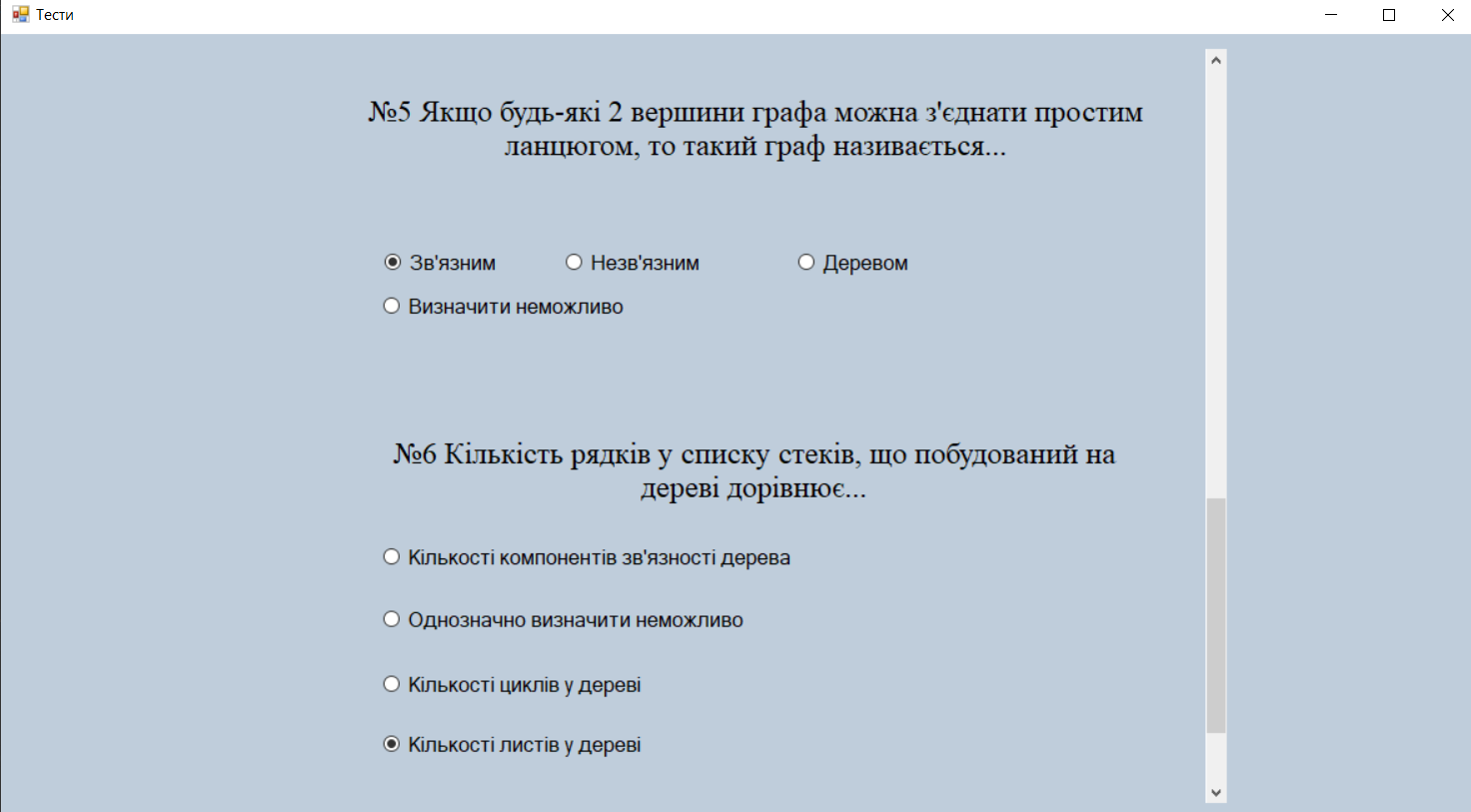
З метою ознайомлення користувача з теоретичним матеріалом або перевірки його знань, було створено швидке тестування. Рис.3.2.

Рис. 3.2. Тести

1. Теорія. Теоретичний розділ – документ Word.
2. Практика.

Основний розділ програми. Складається з двох частин.

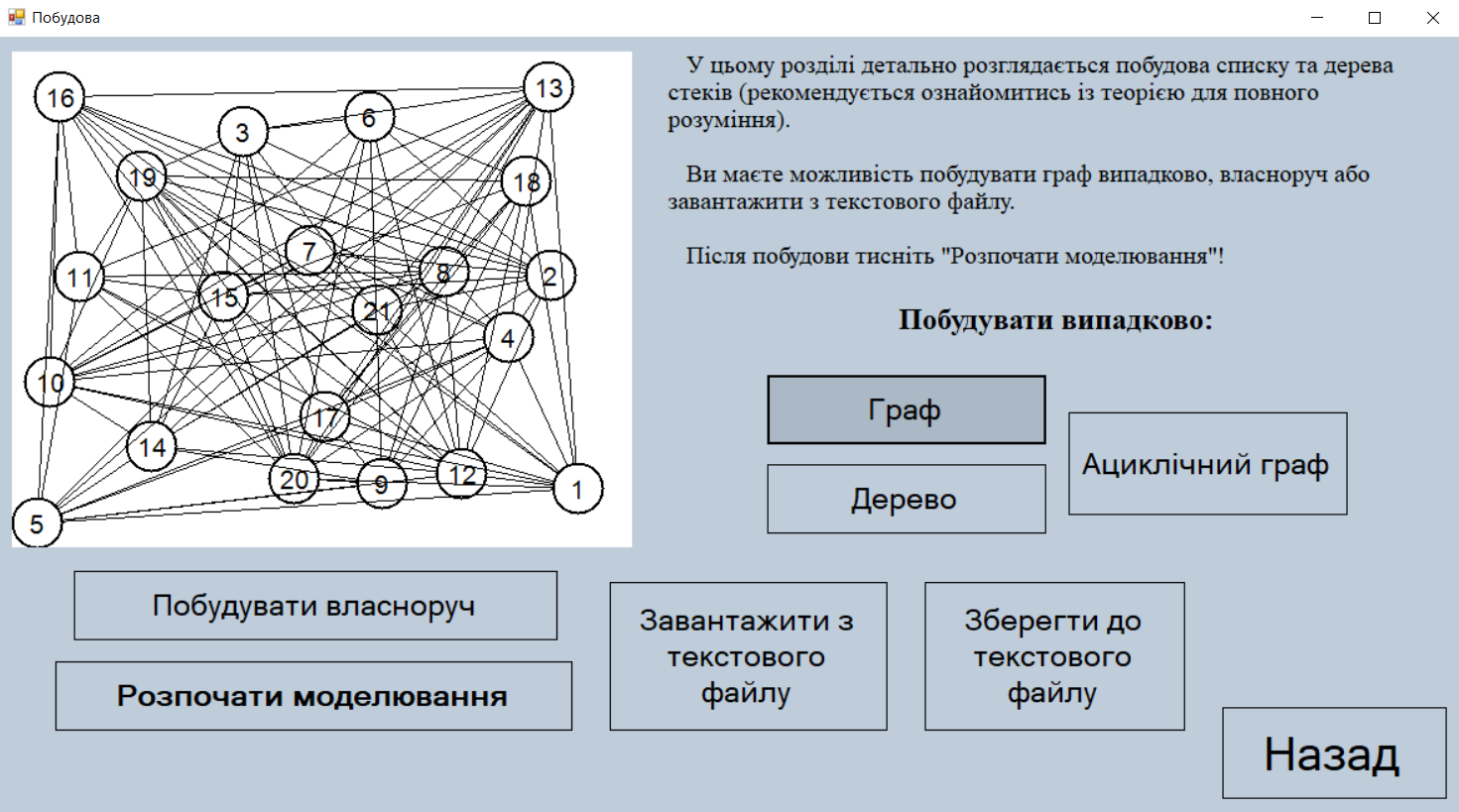
1. Побудова. Користвучач може побудувати граф випадково(рис.3.3), власноруч або завантажити з текстового файлу.
2. Моделювання. Покрокова побудова списку стеків та візуалізація дерева стеків на основі побудованого графа.

Рис. 3.3. Випадково побудований граф

У [додатку B-1](#_ДОДАТОК_B-1) детально розглянуто практичний розділ.

1. Побудова.

Функціонал цього розділу дозволяє швидко побудувати початковий граф, список стеків або дерево стеків, та зберегти побудову у форматі PNG або JPG на комп’ютер користувача. Початковий граф треба завантажувати з текстового файлу.

Цей розділ можна вважати узагальненням роботи. Його вигляд та можливості розгянуто у [додатку B-2](#_ДОДАТОК_B-2).

# ВИСНОВКИ

1. Досліджено існуючі способи збереження графа у пам’яті, а саме: матрицю суміжності, список ребер та список суміжності.
2. Розроблено новий метод, описано та покроково візуалізовано алгоритм його побудови.
3. Вказано, що список стеків доцільно використовувати лише для неорієнтованих графів. Розглянуто побудову списку стеків на дереві та узагальнено її для довільних неорієнтованих графів.
4. Розроблено алгоритм візуалізації дерева, що працює на основі списку стеків, особливістю якого є виражені зв’язки між ланцюгами вершин, що дозволяють розглядати їх як одне ціле. Це значно покращує наочне сприйняття дерева.
5. Розроблено навчальну програму, яка знайомить користувача з пошуком у глибину та його застосуваннями у побудові списку стеків та дерева стеків та має такий програмний функціонал:
   * можливість будувати граф випадково або власноруч, завантажувати та зберігати дані про нього до текстового файлу;
   * покрокове моделювання побудови списку стеків з можливістю зворотного руху по його кроках;
   * візуалізація побудови дерева за допомогою описаного методу та можливість зберегти побудову у форматі JPG або PNG.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Вікіпедія. [Електронний ресурс]. – Режим доступу:

<https://uk.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BE%D1%80%D1%96%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D1%96%D0%B2>

1. Вікіпедія. [Електронний ресурс]. – Режим доступу:

https://uk.wikipedia.org/wiki/%D0%9F%D0%BE%D1%88%D1%83%D0%BA\_%D1%83\_%D0%B3%D0%BB%D0%B8%D0%B1%D0%B8%D0%BD%D1%83

1. Офіційна документація Microsoft. [Електронний ресурс]. – Режим доступу:

https://docs.microsoft.com/ru-ru/dotnet/framework/winforms/windows-forms-overview

1. Керівництво по Windows Forms. [Електронний ресурс]. – Режим доступу:

<https://metanit.com/sharp/windowsforms/>

1. Стаття на задану тему. [Електронний ресурс]. – Режим доступу:

https://habr.com/ru/post/65367/%20...%20%D0%A0%D0%B8%D1%81.1.9

1. Програма для побудови графів. [Електронний ресурс]. – Режим доступу:

<https://csacademy.com/app/graph_editor/>

1. Стаття на задану тему. [Електронний ресурс]. – Режим доступу:

<https://evgavrilenko.ucoz.ru/DS/LEKCIYA_1.pdf>

# ДОДАТКИ

# ДОДАТОК А

Програмний код важливих частин проекту та пояснення до нього.

Мова програмування С#.

## ДОДАТОК А-1

Рекурсивний алгоритм пошуку у глибину.

Мова програмування С#.

* usedVertex[v] = true, якщо вершина v помічена;
* graph – граф представлений списком суміжності;

List<int>[] graph = new List<int>[maxN];

bool[] usedVertex = new bool[maxN];

* рекурсивна процедура

void dfs(int vertex)

{

usedVertex[vertex] = true;

for(int i = 0; i < graph[vertex].Count(); ++i)

{

* graph[vertex][i] – вершина, що суміжна до vertex;

if(usedVertex[graph[vertex][i]] == false)

{

dfs(graph[vertex][i], vertex);

}

}

}

## ДОДАТОК А-2

Побудова списку стеків. Оптимізований метод.

Мова програмування С#.

* used[v] = true, якщо вершина v помічена;
* list\_of\_stack – список стеків;
* cyclic\_edge – список ребер, що утворюють цикли;
* graph – граф представлений списком суміжності;

bool[] used = new bool[maxN];

List<List<int>> list\_of\_stack = new List<List<int>>();

List<Point> cyclic\_edge = new List<Point>();

List<int>[] graph = new List<int>[maxN];

* рекурсивна процедура. Мета: заповнити list\_of\_stack;

void dfs(int vertex, int ancestor = -1)

{

used[vertex] = true;

if (list\_of\_stack.Count() == 0) list\_of\_stack.Add(new List<int>());

list\_of\_stack[list\_of\_stack.Count() - 1].Add(vertex);

* updateInTime() – процедура збереження поточного кроку алгоритму;

updateInTime();

* haveEdge – true, якщо vertex суміжна хоча б з одною непоміченою вершиною;

bool haveEdge = false;

for(int i = 0; i < graph[vertex].Count(); ++i)

{

* to = graph[vertex][i] – вершина, яка суміжна з vertex;

int to = graph[vertex][i];

if (used[to] == true)

{

* якщо to помічена, то додамо ребро між вершинами to та vertex до cyclic\_edge;

Point new\_edge = new Point(vertex, to);

Point new\_edge\_reverse = new Point(to, vertex);

if (to != ancestor && !cyclic\_edge.Contains(new\_edge))

{

if (!cyclic\_edge.Contains(new\_edge\_reverse))

{

cyclic\_edge.Add(new\_edge);

}

}

}

if (used[to] == false)

{

if (haveEdge == true)

{

list\_of\_stack[list\_of\_stack.Count() - 1].Add(vertex);

}

haveEdge = true;

dfs(to, vertex);

}

}

if (haveEdge == false)

{

if(list\_of\_stack[list\_of\_stack.Count() - 1].Count > 0)

{

list\_of\_stack.Add(new List<int>());

}

}

updateInTime();

}

public void buildStackList(int startVertex)

{

dfs(startVertex);

}

## ДОДАТОК А-3

Рекурсивна частина візуалізації стиснутого дерева стеків.

Мова програмування С#.

* lastRow – максимальна кількість рівнів візуалізації. Дорівнює максимальній кількості вершин maxN;
* lastXposition[i] – координата найправішої вершини на рівні i;

int lastRow = maxN;

int[] lastXposition = new int[maxN];

* usedVertex[i] = true, якщо вершина i помічена;
* coordinates[v] – відносні координати вершини v. На основі цих даних визначаються реальні координати вершин під час етапу візуалізації;
* stackList – список стеків, що завчасно побудований на початковому дереві;
* graph – список суміжності, де вершинами є збережені стеки після побудови stackList.

bool[] usedVertex = new bool[maxN];

public Point[] coordinates = new Point[maxN];

public List<List<int>> stackList = new List<List<int>>();

public List<int>[] graph = new List<int>[maxN];

* findYposition – функція пошуку вільного місця, тобто найнижчий рівень, де можна розташувати один або декілька ланцюгів.

int findYposition(int value)

{

for (int i = 0; i < lastRow; ++i)

{

if (lastXposition[i] < value)

{

return i;

}

}

return -1;

}

* dfsCompressed – процедура пошуку у глибину. Результатом її роботи є знайдені відносні координати вершин початкового дерева coordinates[] ;

void dfsCompressed(int vertex, int ancestor = -1)

{

int nowYposition = new int();

usedVertex[vertex] = true;

* заповнення першого горизонталі вершин;

if (ancestor == -1)

{

List<int> way = stackList[vertex];

nowYposition = findYposition(0);

coordinates[way[0]] = new Point(0, nowYposition);

for (int i = 1; i < way.Count(); ++i)

{

Point last = coordinates[way[i - 1]];

coordinates[way[i]] = new Point(last.X + 1, last.Y);

}

* update lastXposition[] після заповнення coordinates[];

for(int i = 0; i <= nowYposition; ++i)

{

if(coordinates[way[way.Count() - 1]].X > lastXposition[nowYposition])

{

lastXposition[nowYposition] = coordinates[way[way.Count() - 1]].X;

}

}

}

* adjacent – список суміжних вершин graph із вершиною vertex;

List<int> adjacent = new List<int>();

for (int i = 0; i < graph[vertex].Count(); ++i)

{

if (usedVertex[graph[vertex][i]] == true) continue;

adjacent.Add(graph[vertex][i]);

}

* wayWertex – список індексів вершини початкового дерева, що входять до горизонтальної послідовності вершин, яка має номер vertex;

List<int> wayWertex = stackList[vertex];

int st = 1; if (ancestor == -1) st = 0;

* Для кожної вершини X поточної горизонталі знайдемо горизонтальні послідовності вершин, що суміжні з поточною по вершині X. На кожній ітерації запишемо їх до списку toAdjacent;
* Вершини поточної горизонталі будемо проходити зліва направо;

for (int i = st; i < wayWertex.Count(); ++i)

{

int size = 0;

List<int> toAdjacent = new List<int>();

for (int j = 0; j < adjacent.Count(); ++j)

{

if (stackList[adjacent[j]][0] != wayWertex[i]) continue;

size += stackList[adjacent[j]].Count() - 1;

toAdjacent.Add(adjacent[j]);

}

* пошук nowYposition та оновлення lastXposition[];

if (toAdjacent.Count() == 0) continue;

nowYposition = findYposition(coordinates[wayWertex[i]].X - size + 1);

for (int j = 0; j <= nowYposition; ++j)

{

if (coordinates[wayWertex[i]].X > lastXposition[j])

{

lastXposition[j] = coordinates[wayWertex[i]].X;

}

}

* визначаємо координати вершин знайдених суміжних горизонталей;

int x = coordinates[wayWertex[i]].X;

for (int j = toAdjacent.Count() - 1; j >= 0; --j)

{

List<int> way = stackList[toAdjacent[j]];

for (int k = way.Count() - 1; k >= 1; --k)

{

coordinates[way[k]] = new Point(x, nowYposition);

--x;

}

}

* запускаємо dfsCompressed від суміжних (по вузловій вершині ) до vertex горизонтальних послідовностей вершин.

for (int j = 0; j < toAdjacent.Count(); ++j)

{

dfsCompressed(toAdjacent[j], vertex);

}

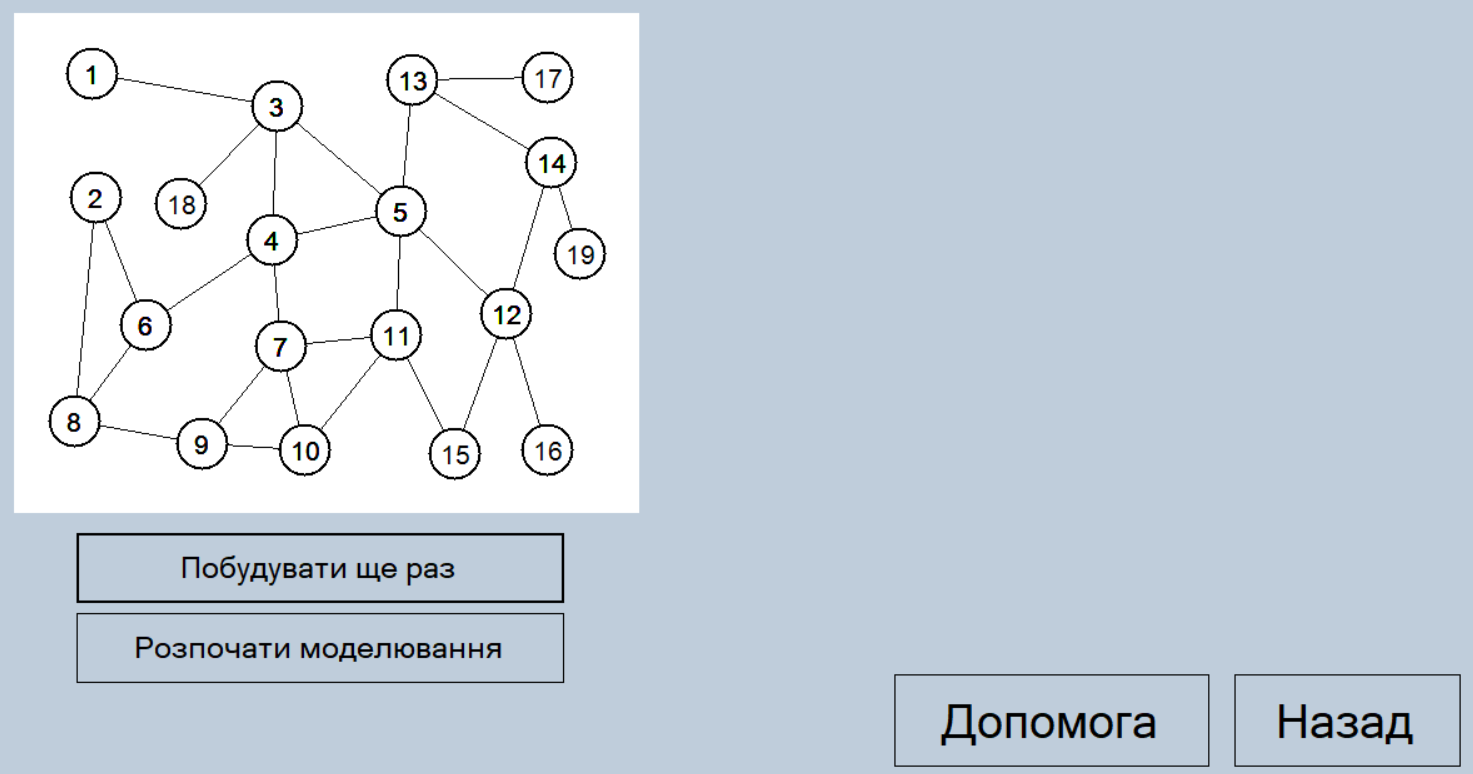
}

}

# ДОДАТОК B

Детальне ознайомлення зі створеним навчальним додатком

## ДОДАТОК B-1

Детальний розгляд практичного розділу.

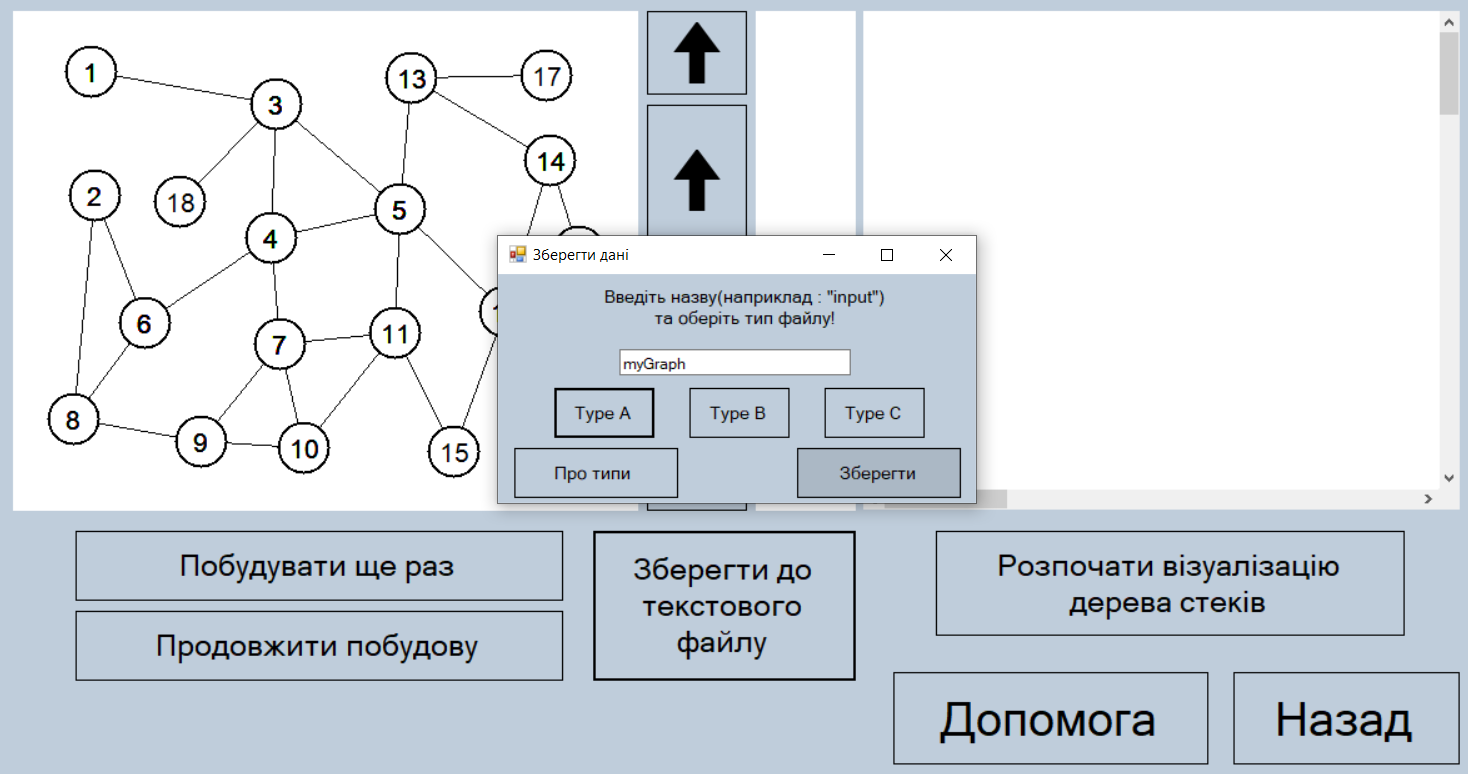
Рис 1. Режим побудови графу власноруч

Рис. 2. Можливість зберігати побудований граф

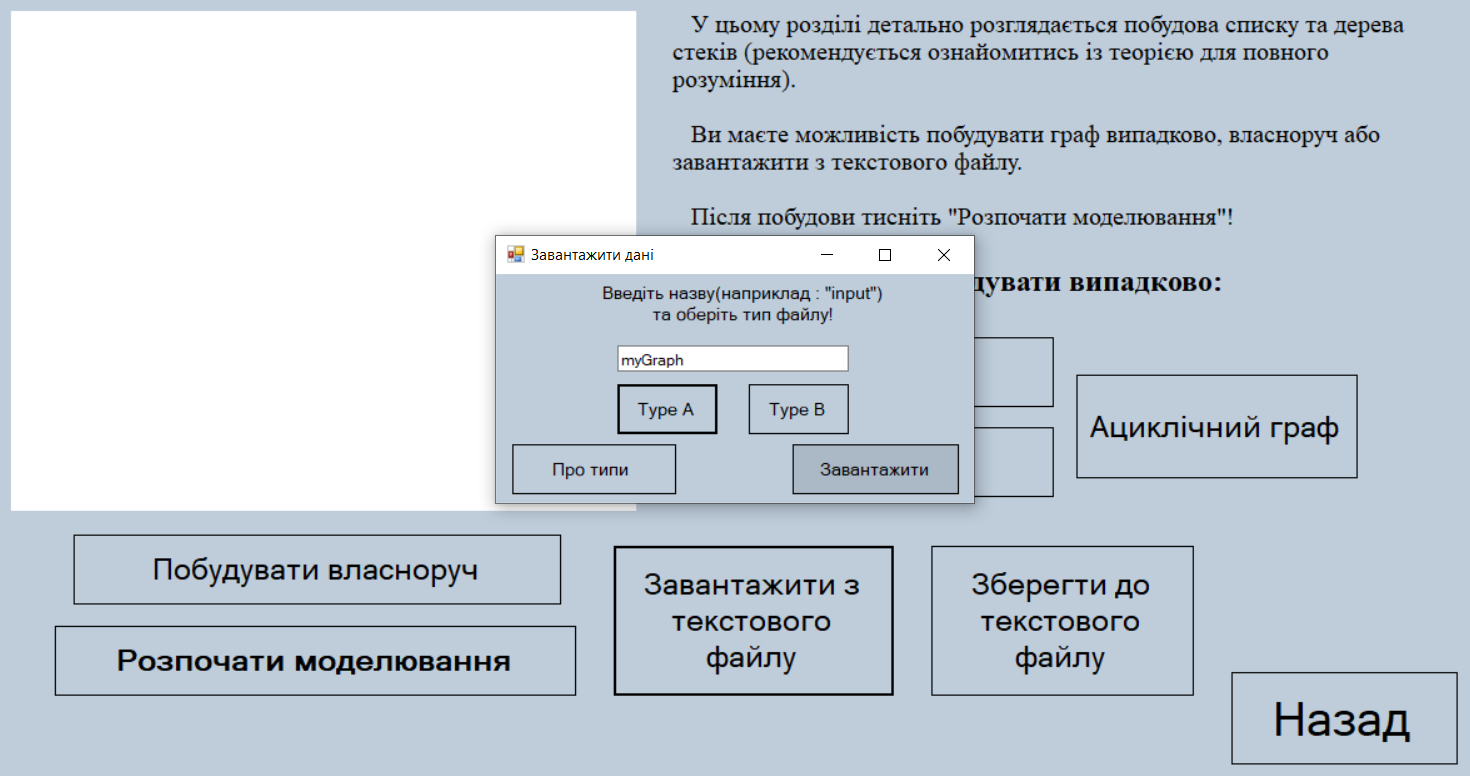
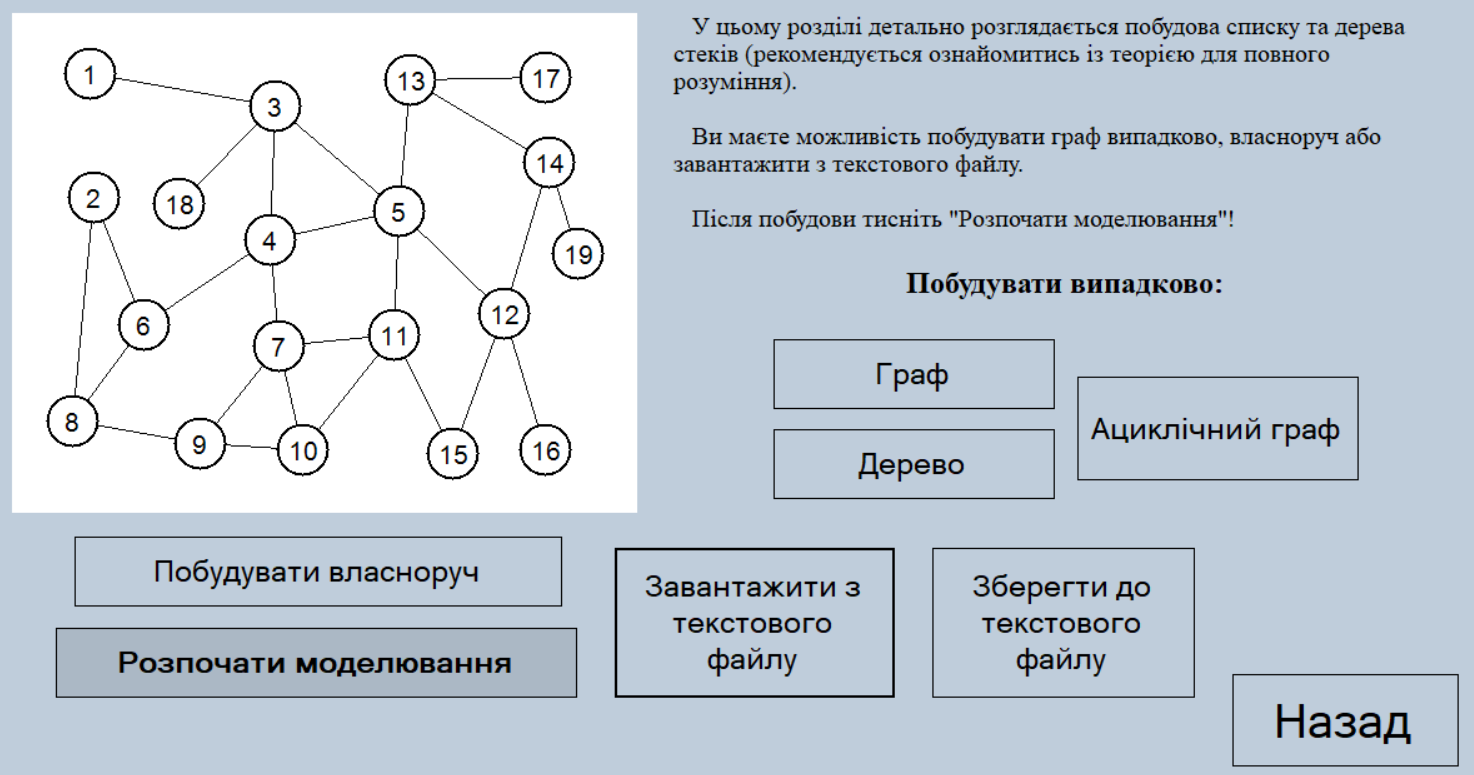
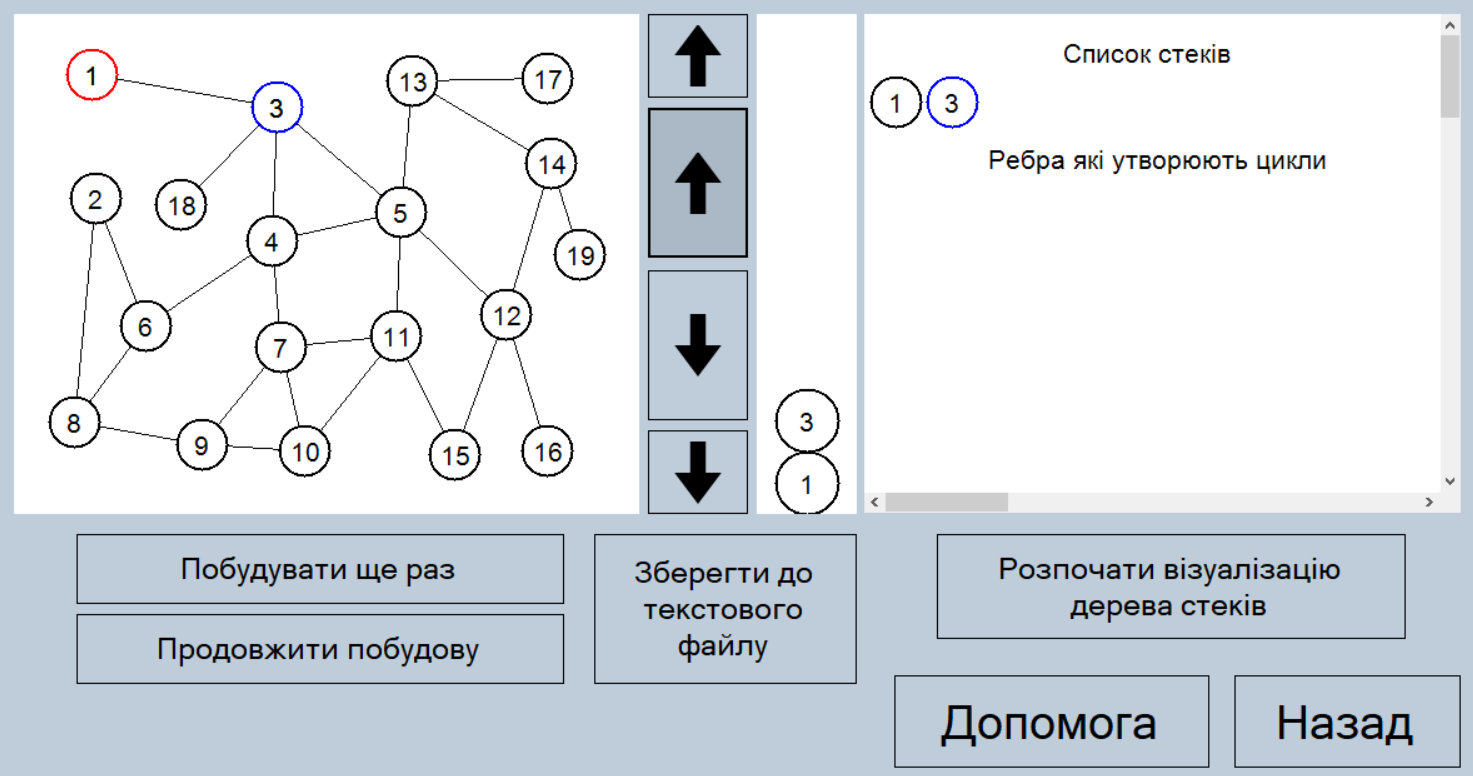
Рис. 3. Можливість завантажити дані на етапі побудови

Рис. 4. Успішно завантажено побудований раніше граф

Також граф можна будувати випадково. Функціонал дозволяє побудувати довільний граф, дерево або ациклічний граф.

Завершивши побудову тиснемо “Розпочати моделювання”.

Рис. 5. Розпочато моделювання

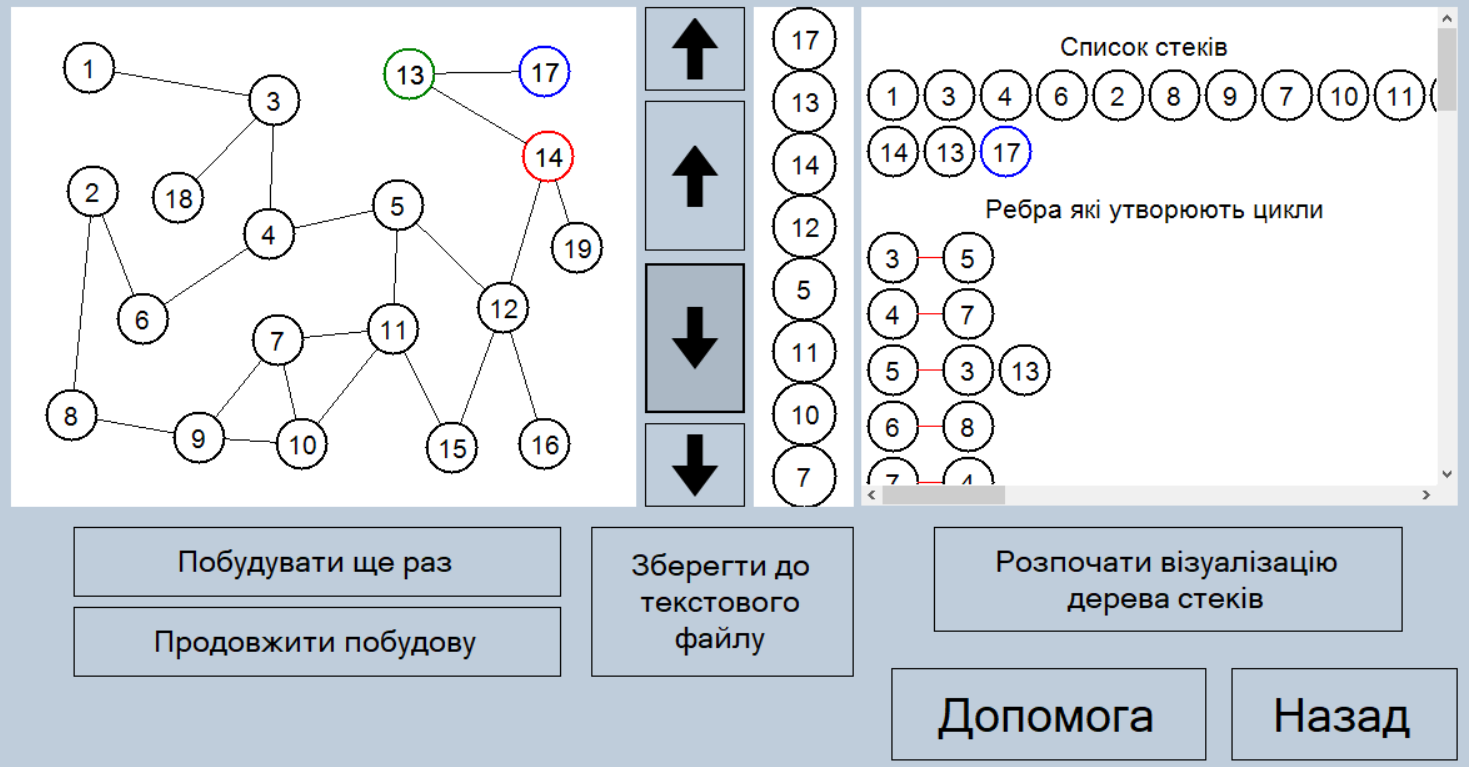
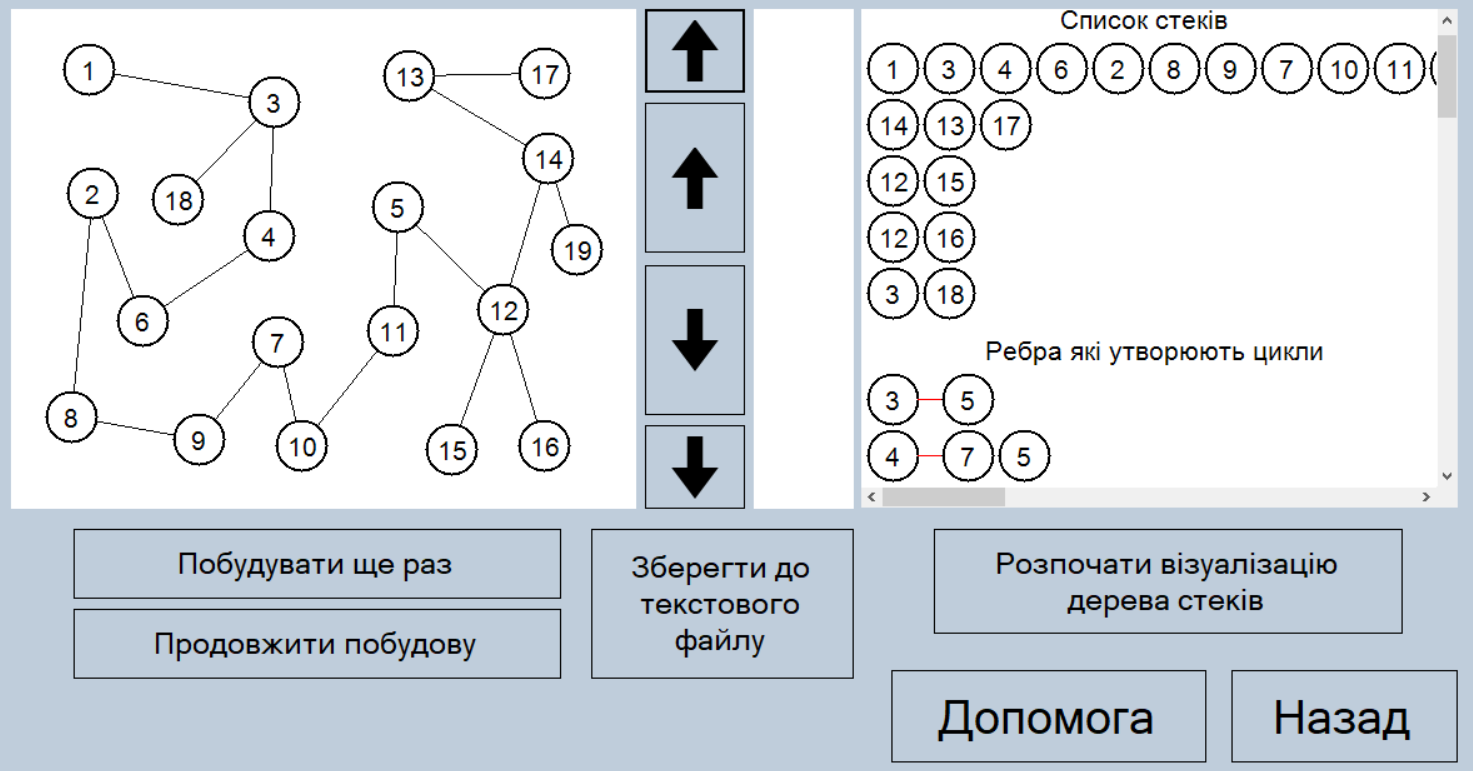
Два рази натиснувши на велику стрілку вгору, ми пройшли дві фази алгоритму. Якщо бажаєте на останню фазу – тисніть малу стрілку вгору. Аналогічно працюють стрілки вниз.

Рис 6. Середня фаза алгоритму

Рис. 7. Кінцева фаза моделювання

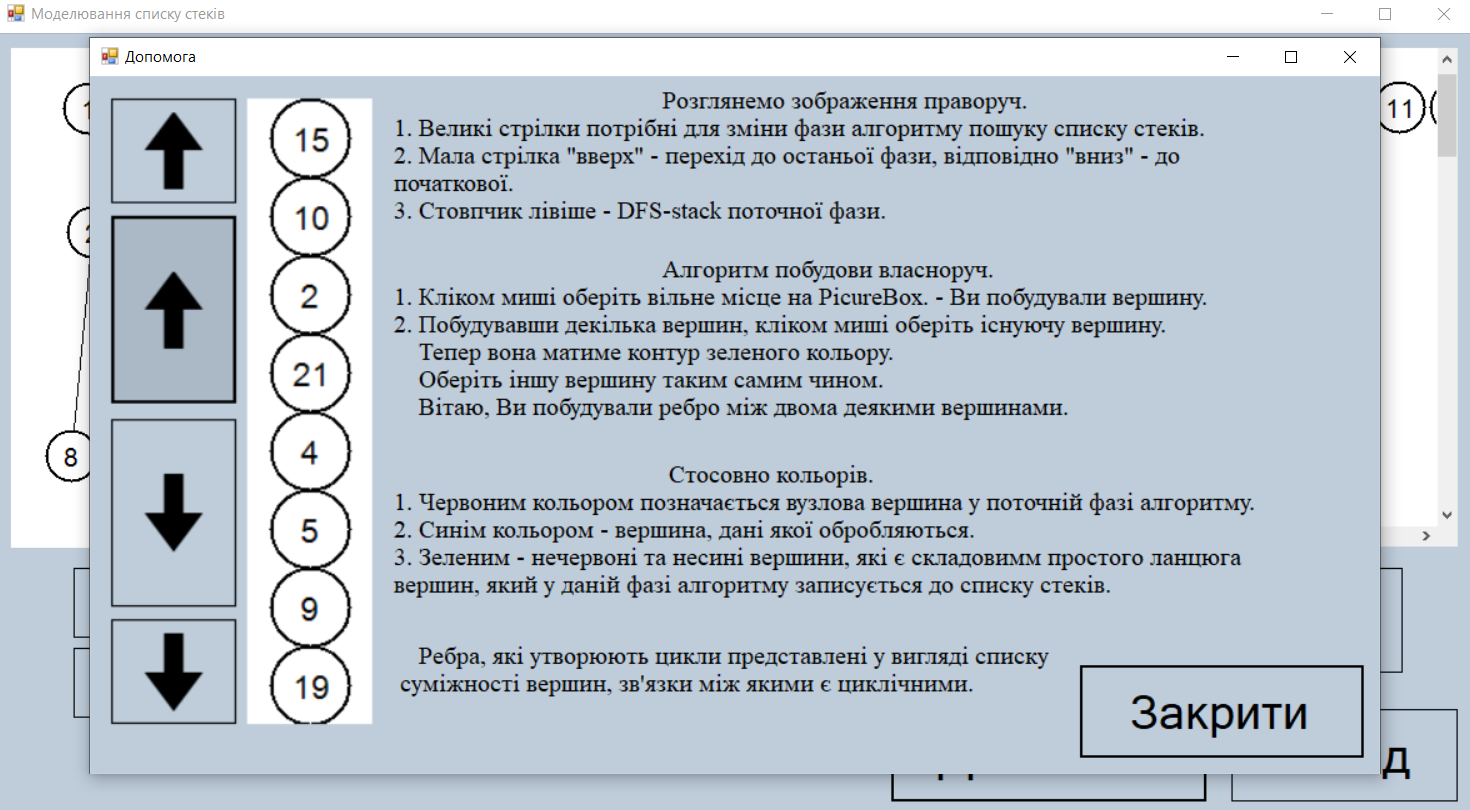
Також реалізована можливість збереження побудованого списку стеків.

Рис.8 Допомога

Після завершення моделювання тиснем “Розпочати візуалізацію дерева стеків”.

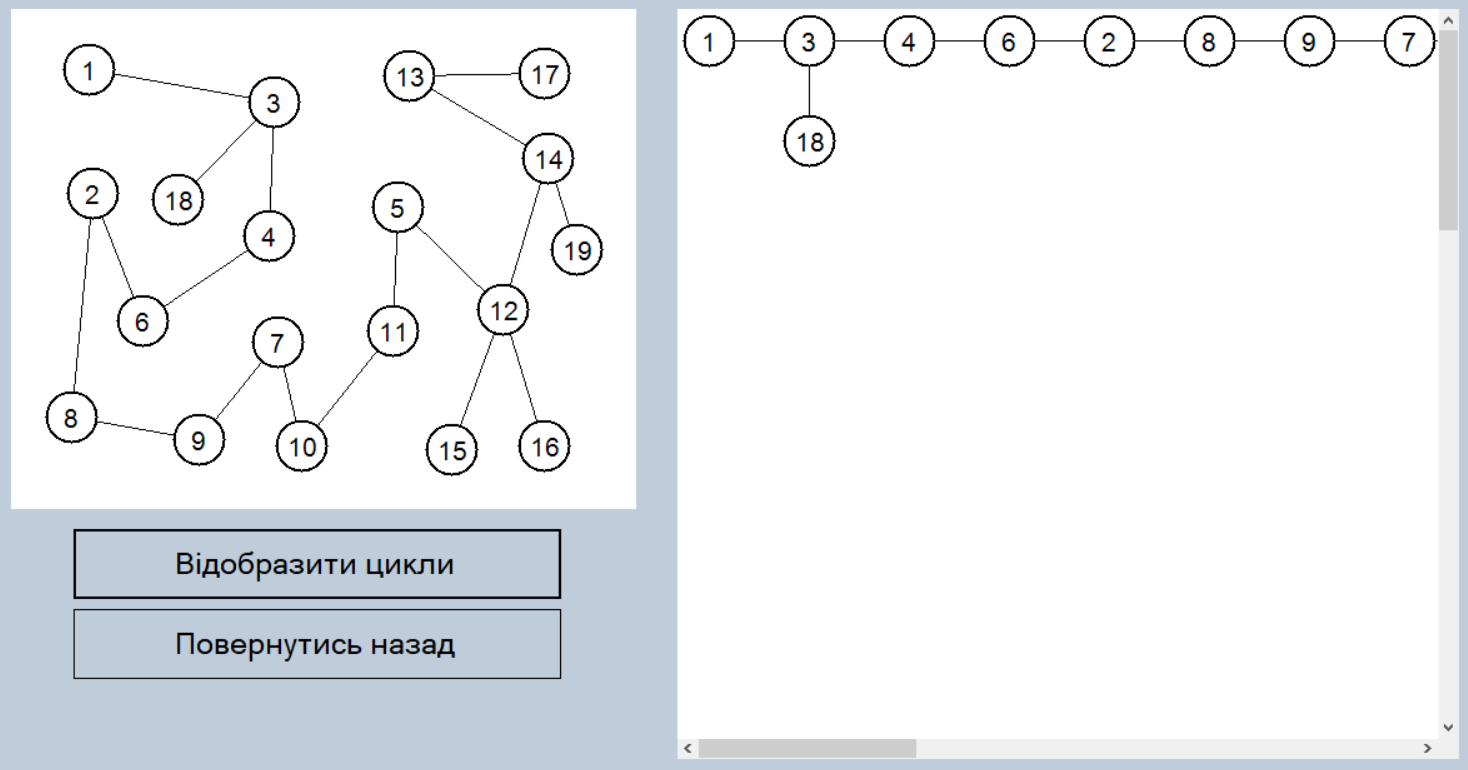
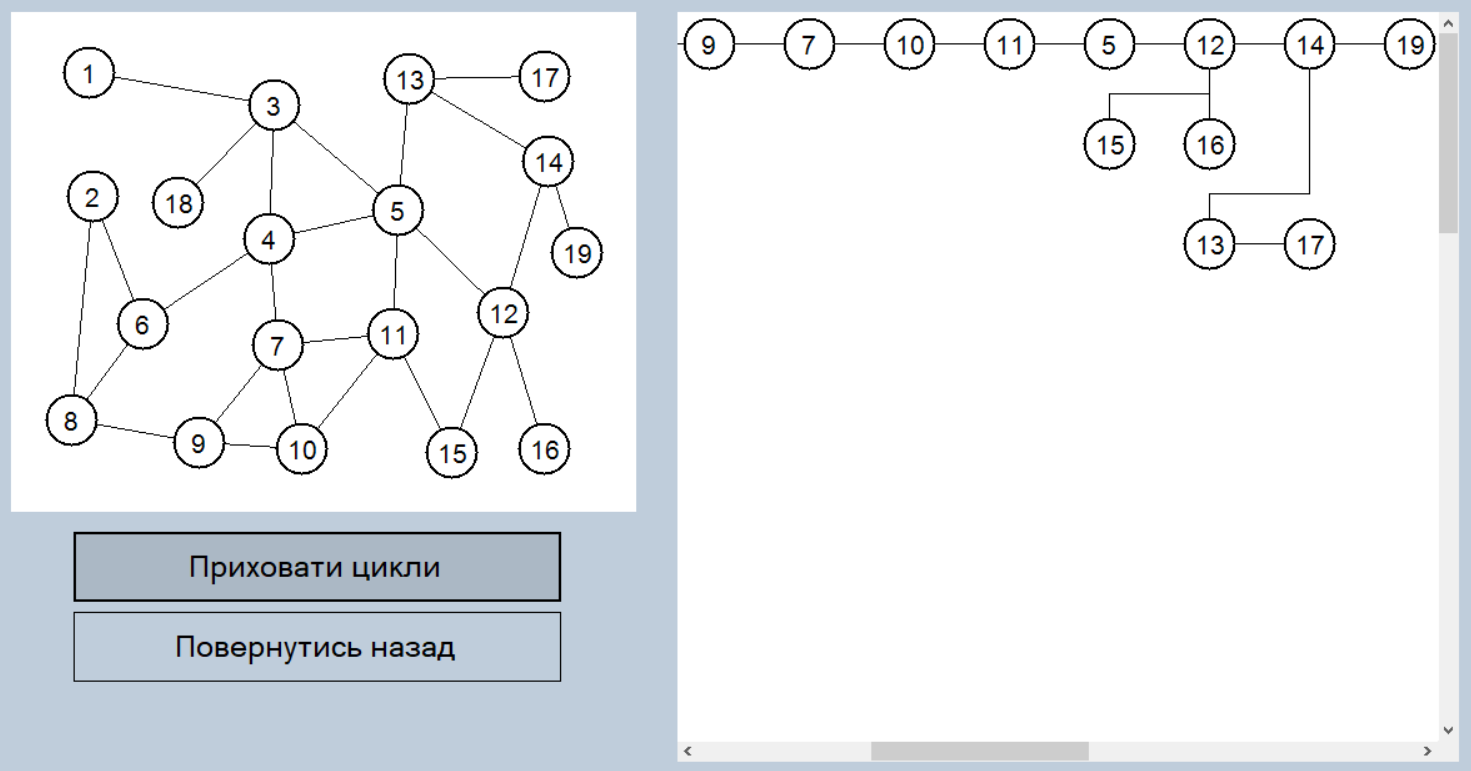
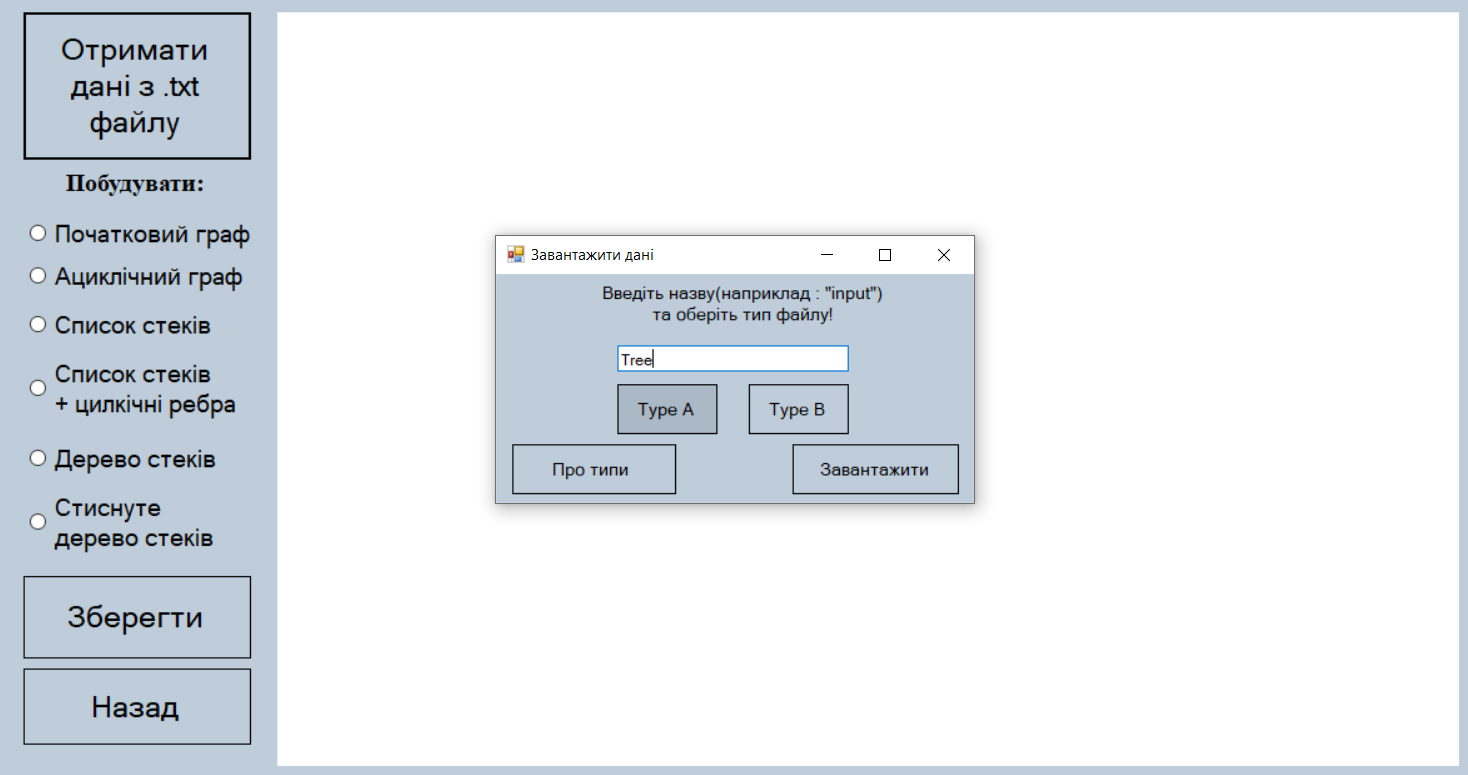
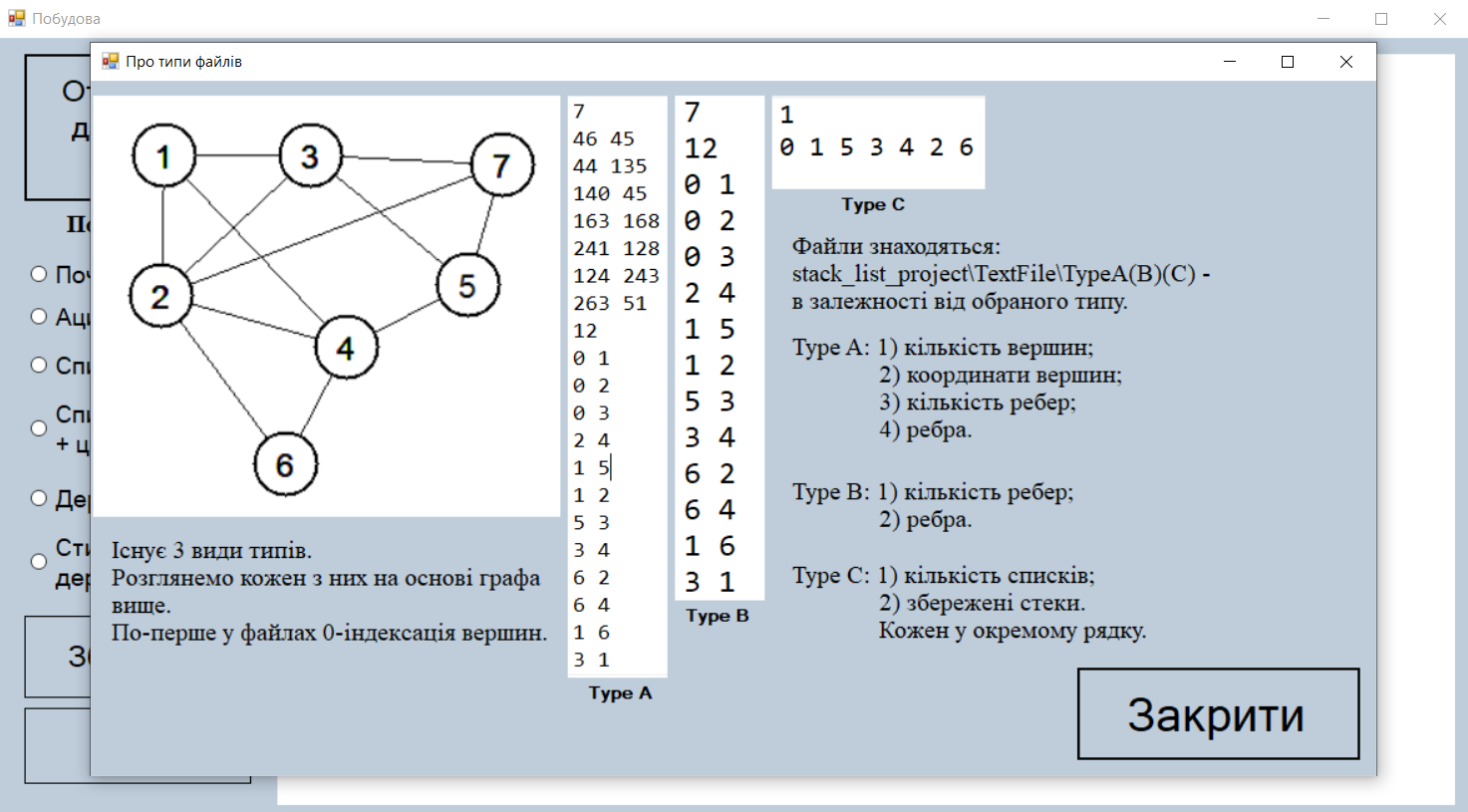
Рис.9. Візуалізація дерева стеків

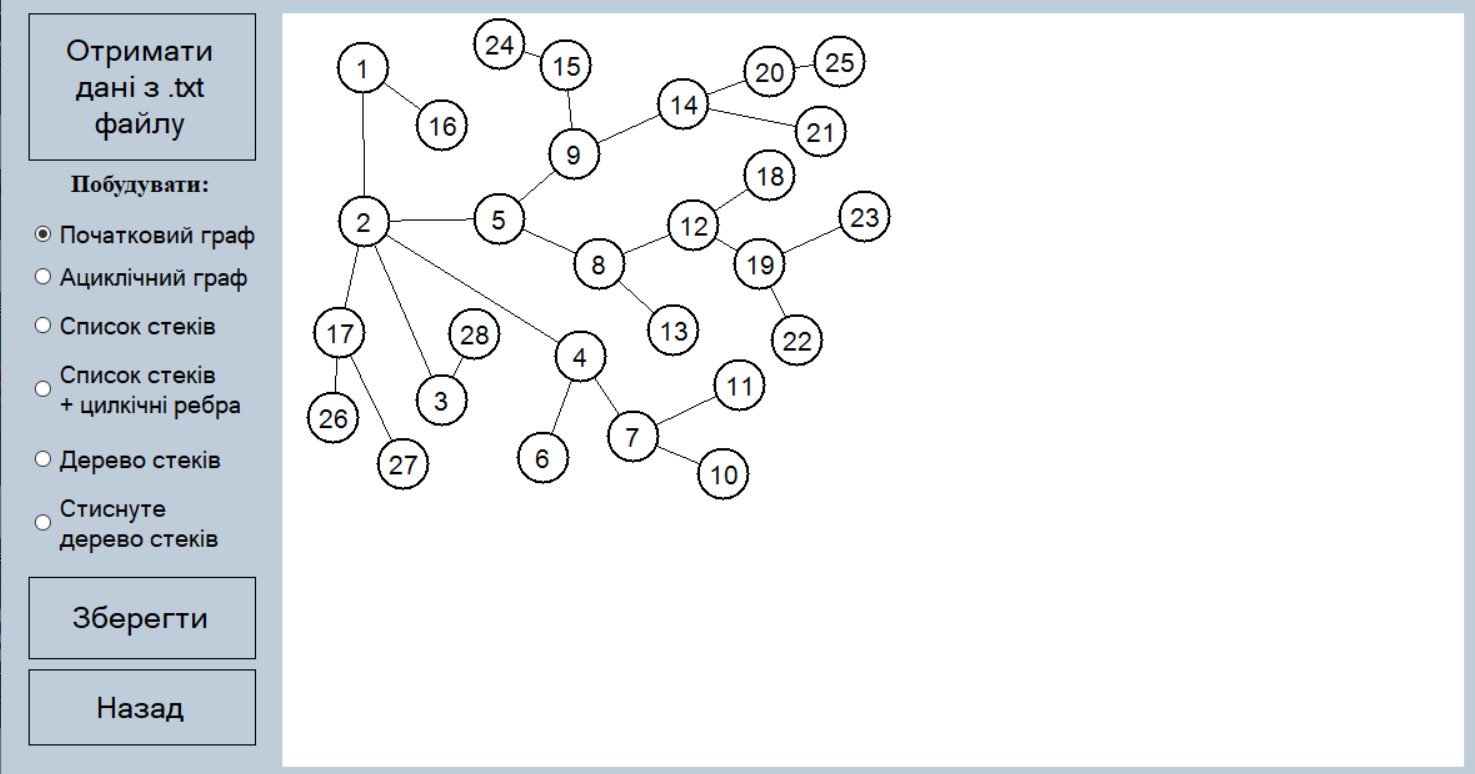
Рис.10. Візуалізація дерева стеків(2)

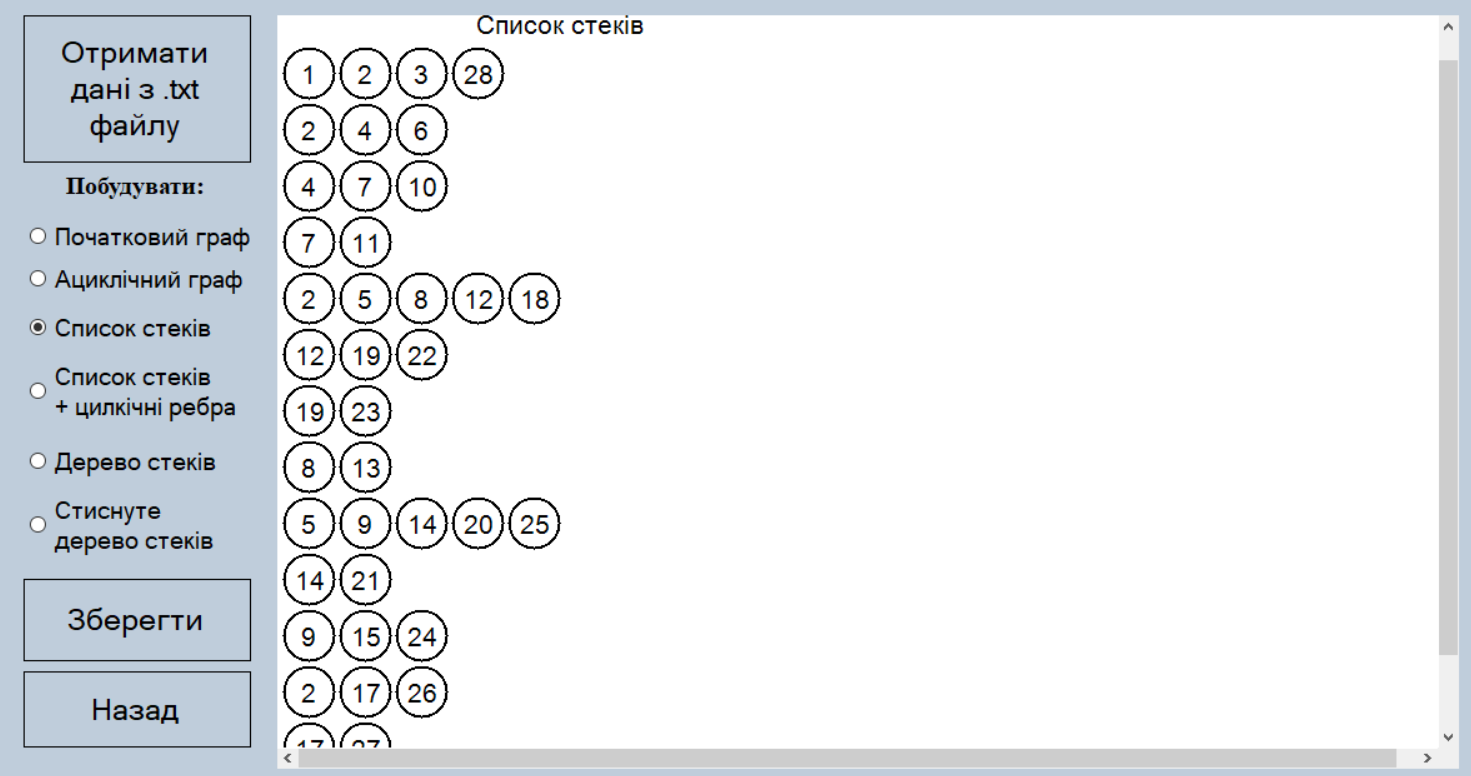
## ДОДАТОК B-2

Детальний розгляд розділу “Побудова”.

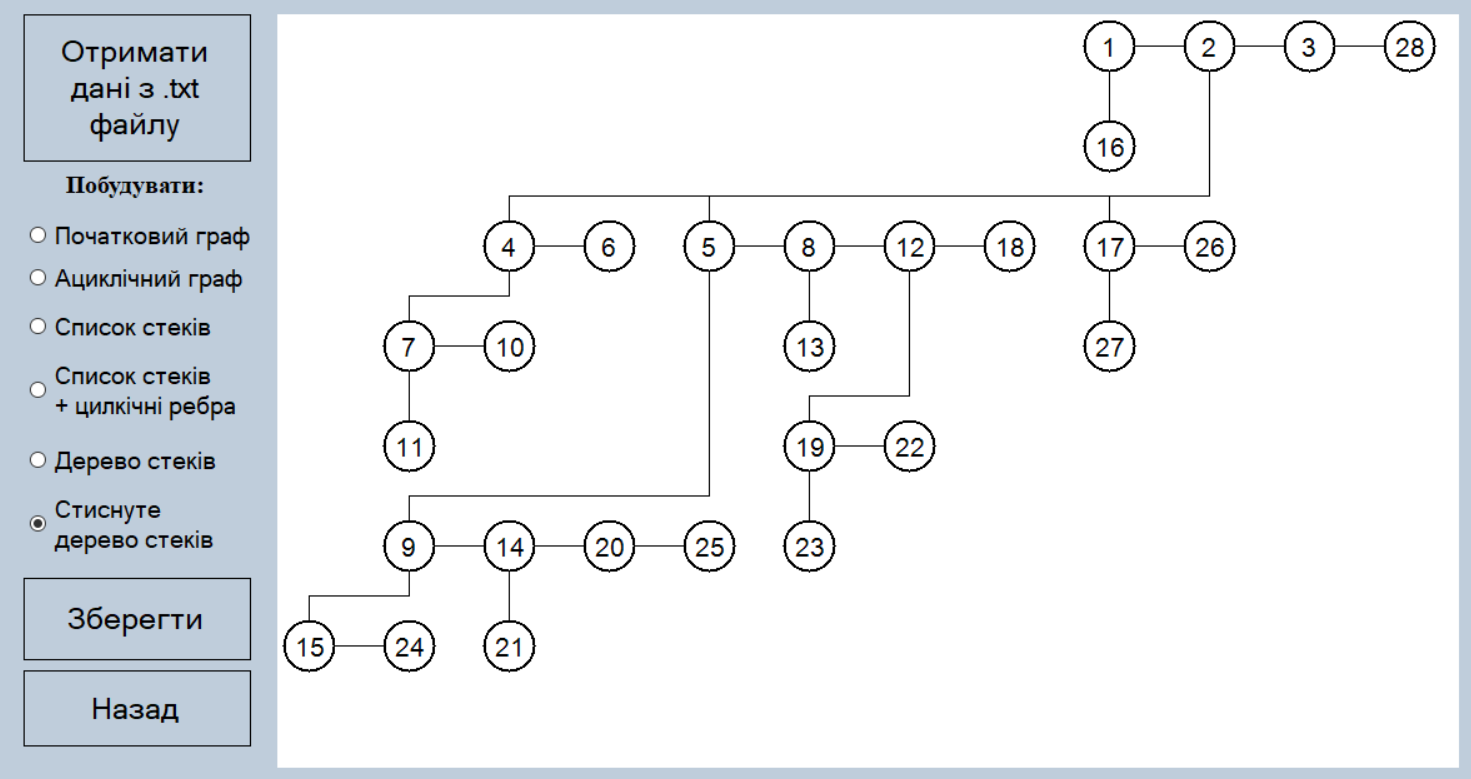
Рис.11. Завантажуємо дані

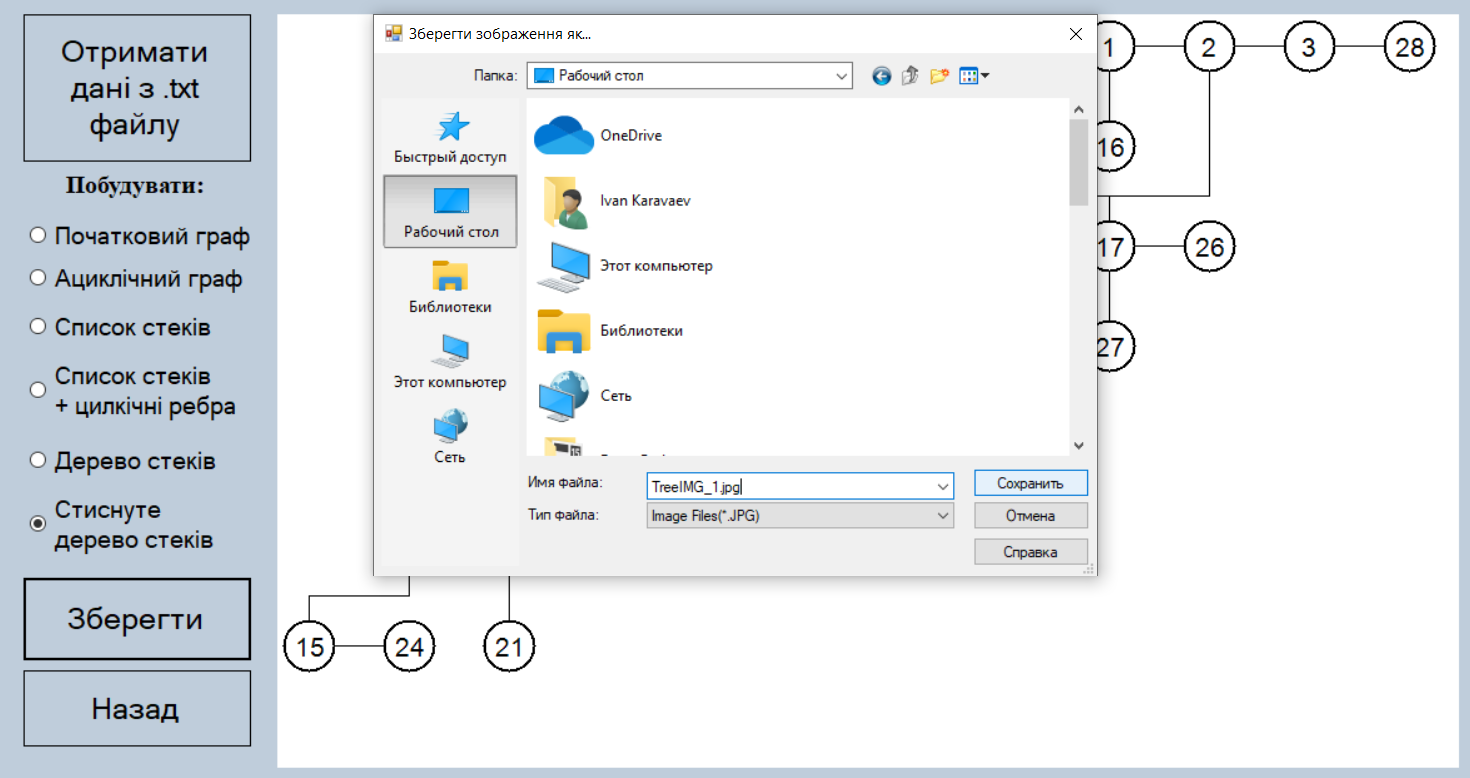
Рис.12. Про види файлів.

Рис.13. Початковий граф

Рис.14. Список стеків

Якщо порівнювати рис.13 та рис.14, можна побачити, що розміри зображення різняться. Справа в тому, що вони динамічно змінюються в залежності від розмірів контенту.

Рис.15. Стиснуте дерево стеків

Рис.16. Зберегти зображення як…

Будь-який з пунктів побудови можна зберегти у форматі JPG або PNG.