

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Інститут інформатики та радіоелектроніки
Факультет комп'ютерних наук і технологій
(повне найменування інституту, факультету)

Кафедра системного аналізу та обчислювальної математики
(повне найменування кафедри)

Пояснювальна записка

до дипломного проекту (роботи)

бакалавра

(ступінь вищої освіти)

на тему Візуалізація решітки формальних понять

Виконав: студент(ка) 4 курсу, групи КНТ-816

Спеціальності 124 «Системний аналіз»
(код і найменування спеціальності)

Освітня програма (спеціалізація)
«Інтелектуальні технології та прийняття рішень»

Литвиненко А.В.

(прізвище та ініціали)

Керівник

Терещенко Е.В.

(прізвище та ініціали)

Рецензент

Шиманова І.І.
(прізвище та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
 (повне найменування вищого навчального закладу)

Інститут, факультет _____ Інститут інформатики та радіоелектроніки,
 Кафедра _____ Факультет комп'ютерних наук і технологій
 Системного аналізу та обчислювальної математики
 Ступінь вищої освіти _____ бакалавр
 Спеціальність _____ 124 Системний аналіз
 (код і найменування)
 Освітня програма (спеціалізація) _____ Інтелектуальні технології та прийняття рішень
 (назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри _____
 проф. Г.В. Корніч
 « 4 » _____ червня 2020 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

Литвиненка Артема Валерійовича
 (прізвище, ім'я, по батькові)

1. Тема проекту (роботи) _____ Візуалізація решітки формальних понять

керівник проекту (роботи) Терещенко Еліна Валентинівна, к.ф.- м.н., доц.
 (прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНТУ від « 19 » травня 2020 року № 136

2. Строк подання студентом проекту (роботи) _____ 04 червня 2020 року

3. Вихідні дані до проекту (роботи) Теоретичні відомості аналізу формальних понять,
 графічна бібліотека SFML

4. Зміст розрахунково-пояснювальної записки (перелік питань, які належить
 розробити) Проведено дослідження публікацій та програмних реалізацій за темою
 роботи. Наведено власну програмну реалізацію для побудови решітки формальних
 понять

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

6. Консультанти розділів проекту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	завдання видав	Підпис, дата		прийняв виконання завдання
1-3	Терещенко Єліна Валентинівна, к.ф.-м.н., доцент				
	Морозова Вікторія Дмитрівна, м. Київ				

7. Дата видачі завдання « 2 » жовтня 20 19 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Приймає
1	Сформулювати мету та основні завдання дипломної роботи	09.10.2019 – 30.10.2019	
2	Опрацювати літературу та наукові розробки за темою роботи	01.11.2019 – 10.11.2019	
3	Розробка логічної частини програми	11.11.2019 – 15.12.2019	
4	Оптимізація логічної частини програм. Розробка графічної частини програм	11.02.2020 – 05.05.2020	
5	Підсумки щодо дипломної роботи. Оформлення пояснювальної записки	21.05.2020 – 28.05.2020	
6	Попередній захист дипломного проекту та отримання рецензій	26.05.2020 – 03.06.2020	
7	Захист дипломного проекту	04.06.2020	

Студент

Керівник проекту (роботи)

(підпис)

Терещенко Е.В. (прізвище та ініціали)

(підпис)

Литвиненко А.В. (прізвище та ініціали)

РЕФЕРАТ

ПЗ: 49с., 19 рис. 16 джерел, 1 додаток.

Об'єкт дослідження: засоби візуального аналізу даних

Мета роботи: створення алгоритму побудови графа – решітки формального поняття.

В дипломній роботі розглядається проблема візуального аналізу даних. Теоретичним підґрунтям обрано методи аналізу формальних понять. Проаналізовано можливості реалізації алгоритму для побудови та зображення графа – решітки за допомогою сучасних мов програмування. У практичній частині розроблено власний алгоритм побудови графа - решітки. Мова програмування – C++. Графічна бібліотека SFML.

ГРАФ – РЕШІТКА, БІНАРНІ ВІДНОСИНИ, ПОБУДУВАННЯ ГРАФІВ,
АНАЛІЗ АЛГОРИТМІВ, РОЗРОБКА АЛГОРИТМУ, ВІЗУАЛІЗАЦІЯ ГРАФІВ,
АНАЛІЗ ФОРМАЛЬНИХ ПОНЯТЬ

ЗМІСТ

Завдання.....	2
Реферат.....	6
Вступ.....	6
1. Візуалізація даних методами аналізу формальних понять.....	7
1.1 Аналіз формальних понять та лінійні діаграми.....	7
1.2 Загальні техніки побудови та зображення графів.....	10
2. Розробка алгоритму та вибір засобів реалізації.....	13
2.1 Постановка задачі.....	13
2.2 Розробка власного алгоритму	14
2.3 Вибір мови програмування	16
2.4 Вибір середовища розробки	18
2.5 Вибір графічної бібліотеки	19
3 Програмна реалізація та тестування роботи алгоритму.....	20
3.1 Визначення входних параметрів алгоритму.....	20
3.2 Знаходження зв'язків між елементами решітки.....	22
3.3 Графічне зображення решітки	26
Висновок.....	34
Перелік посилань.....	35
Додаток А. Програмний код.....	37

ВСТУП

Візуалізація інформації є важливою частиною інтелектуального аналізу даних (ІАД). Більшість людей найкраще сприймають графічну інформацію. Візуальне уявлення дає можливість наочно представити інформацію і часто дозволяє з першого погляду виявити закономірності, які інакше можна знайти лише за допомогою трудомісткого аналізу.

При вирішенні багатьох проблем виникає завдання візуалізації частково впорядкованих множин, а також їх окремого випадку – решіток. Особливо актуальна ця проблема при використанні одного з найбільш потужних методів ІАД - аналізу формальних понять.

Аналіз формальних понять - потужний метод аналізу даних, який неодноразово успішно застосовувався на практиці. Яскраві приклади: пошук закономірностей, онтологічне моделювання, інформаційний пошук та ін. [1]. Основним засобом візуалізації залежностей, що застосовується в методі аналізу даних, є лінійна діаграма решітки формальних понять. У теоретичній частині проводиться огляд методів зображення лінійних діаграм: розкладання решітки на ланцюг і взаємодії сил. У практичній частині було розроблено та протестовано власний алгоритм побудови решітки.

1 ВІЗУАЛІЗАЦІЯ ДАНИХ МЕТОДАМИ АНАЛІЗУ ФОРМАЛЬНИХ ПОНЯТЬ

1.1 Аналіз формальних понять та лінійні діаграми

Аналіз формальних понять (Formal Concept Analysis, FCA) був запропонований Вілле в 1981 році і активно розвивається донині. Математичний формалізм, що є підґрунтям цього методу, є поняття решітка [2].

Визначення 1. Решітка – це у-множина L , в якій будь-які два елемента x і y мають точну нижню грань, яка називається *перетином* ($x \wedge y$) та точну верхню грань, яка називається *об'єднанням* ($x \vee y$) [2, с.74].

Основою методу FCA є факт, що для бінарного відношення завжди можна однозначно побудувати повну решітку. Метод FCA застосовується для аналізу якісної інформації. Розглянемо дані, які представлені у вигляді таблиці, що містить атрибути об'єктів у стовпцях, а ідентифікатори об'єктів – у рядках. Якщо об'єкт має певну властивість, то перетин стовпця та строки позначається одиницею. Таке представлення є базою для формального контексту.

Формальне поняття (концепт) складається з обсягу і змісту. Зміст – це множина властивостей, що описують поняття. В обсяг входять всі об'єкти з контексту, які мають всі властивості зі змісту. При цьому зміст повинен бути максимальним, тобто, включати всі загальні властивості об'єктів з обсягу поняття. Математично формальне поняття задається за допомогою відповідностей Галуа [3].

Визначення 2. Відношення передування: пара $(A1, B1) \leq (A2, B2)$, якщо $A1 \subseteq A2$ і $B1 \supseteq B2$.

Між поняттями встановлюється відношення передування згідно визначенню 2, де $A1, A2$ – обсяги понять 1 і 2, а $B1, B2$ – відповідно їх змісту.

Множина всіх понять контексту утворює повну решітку, яку називають решіткою концептів. Решітка концептів містить всю інформацію про

взаємозалежності, існуючі між атрибутами в контексті, за яким вона була побудована.

Традиційно для візуалізації частково – впорядкованих множин використовуються діаграми Гассе. У методі FCA застосовується їх різновид, в якому використовується скорочена позначка – кожен об'єкт і атрибут зображуються на діаграмі всього один раз. Ім'я об'єкта приписується перетину всіх понять, в обсягах яких міститься цей об'єкт, а ім'я властивості приписується об'єднанням всіх понять, змісту яких включають цю властивість. Таким чином, ім'я об'єкта приписується найменшому з понять, в яких зустрічається даний об'єкт, а ім'я властивості приписується найбільшому з понять, в яких присутня ця властивість. Такі діаграми називаються лінійними діаграмами. На рисунку 1.1 представлено формальний контекст, а на рисунку 1.2 представлена лінійна діаграма решітки понять з скороченою позначкою.

	a	b	c	d	e
1	×				
2	×	×		×	×
3	×	×	×		×
4	×	×	×	×	

Рисунок 1.1 – Формальний контекст

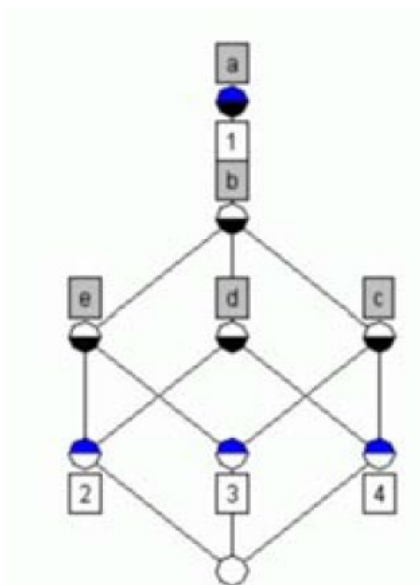


Рисунок 1.2 – Лінійна діаграма решітки понять

Вимоги, що пред'являються до зображень лінійних діаграм: добре представлена лінійна діаграма повинна бути «прозорою, має легко читатися і має полегшувати інтерпретацію представлених даних». Однак способи досягнення цього залежать від цілей інтерпретації і від структури решітки.

При зображенні лінійної діаграми обов'язковою є вимога про те, що всі підконцепти будь-якого концепту повинні бути розташовані нижче його.

Бажаним є виконання наступних умов:

- ребра повинні бути зображені у вигляді прямих ліній;
- дві вершини не повинні бути розташовані в одній точці;
- кількість перетинів між ребрами решітки має бути якомога менше;
- ребро не повинне перетинати вершину, яка не є її кінцем;
- повинна бути наочно представлена структура решітки;
- використання якомога меншої кількості різних напрямків;
- максимізація кількості паралельних ліній.

Додатковою вимогою, котру можна пред'явити до алгоритмів зображення лінійних діаграм, є симетричність зображення лінійних діаграм

решіток, побудованих по контексту і відповідного йому транспонованому контексту (так як між цими діаграмами можна встановити ізоморфізм).

Розглянемо тепер існуючі підходи до зображення графів і лінійних діаграм.

1.2 Загальні техніки побудови та зображення графів

Велика група методів зображення графів може бути представлена в узагальненому вигляді наступним чином:

- а) розрахувати початкове положення графа;
- б) поки можливо, оптимізувати викладки графа;
- в) провести кінцеве підстроювання положень окремих вершин.

В рамках даної схеми можуть бути описані як методи для викладки ієрархічних графів, так і методи «взаємодії сил».

Розглянемо традиційні методи зображення ієрархічних графів [4].

До цієї групи методів належать методи, засновані на схемі, вперше запропонованої К.Сугіяма[4]. Схема орієнтована в першу чергу на задоволення наступних критеріїв: мінімізація перетину між ребрами і скорочення довжини ребер.

Алгоритм складається з чотирьох етапів:

- а) розрахунок початкових позицій вершин і перетворення графа в 2-шаровий, у якого є суміжними вершини тільки двох сусідніх рівнів за рахунок додавання фіктивних вершин;
- б) мінімізація перетинів з використанням барицентрична або медіанної евристики;
- в) розрахунок координат вершин графа;
- г) безпосереднє зображення графа.

Результат роботи традиційного методу можна побачити на рисунку 1.3.

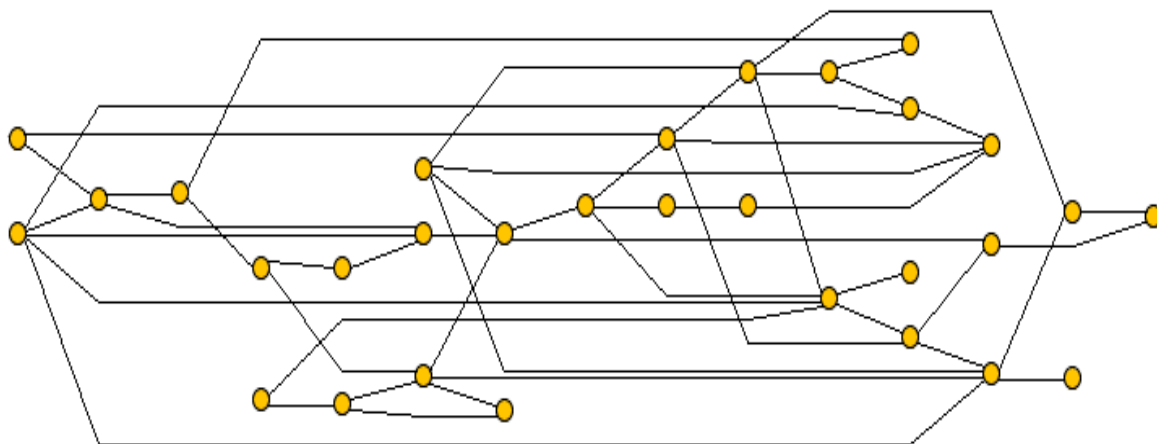


Рисунок 1.3 – Приклад розміщення графа за допомогою методу К.Сугіями (традиційного методу)

Розглянемо методи взаємодії сил.

Широко поширеною групою методів для зображення графів загального вигляду є методи, засновані на моделюванні взаємодії сил (force – directed layouts).

Початкова модель була запропонована П.Ідсом [4].

У моделі зазвичай задаються сили «відштовхування», що діють між усіма вершинами, і сили «тяжіння», що діють між вершинами, сполученими ребрами. Також можуть задаватися сили, що діють між ребрами і вершинами, з метою уникнення перетину ребер і вершин.

Алгоритми «взаємодії сил» задаються моделлю сил, що діє між вершинами і ребрами графа, і процедурою оптимізації, яка використовується для знаходження положення, в якому досягається локальний мінімум енергії. Перевагами даних методів є виявлення симетрій, що існують в графі, і легка переносимість на випадок трьох і більше вимірів.

Результат роботи методу взаємодії сил можна спостерігати на рисунку 1.4.

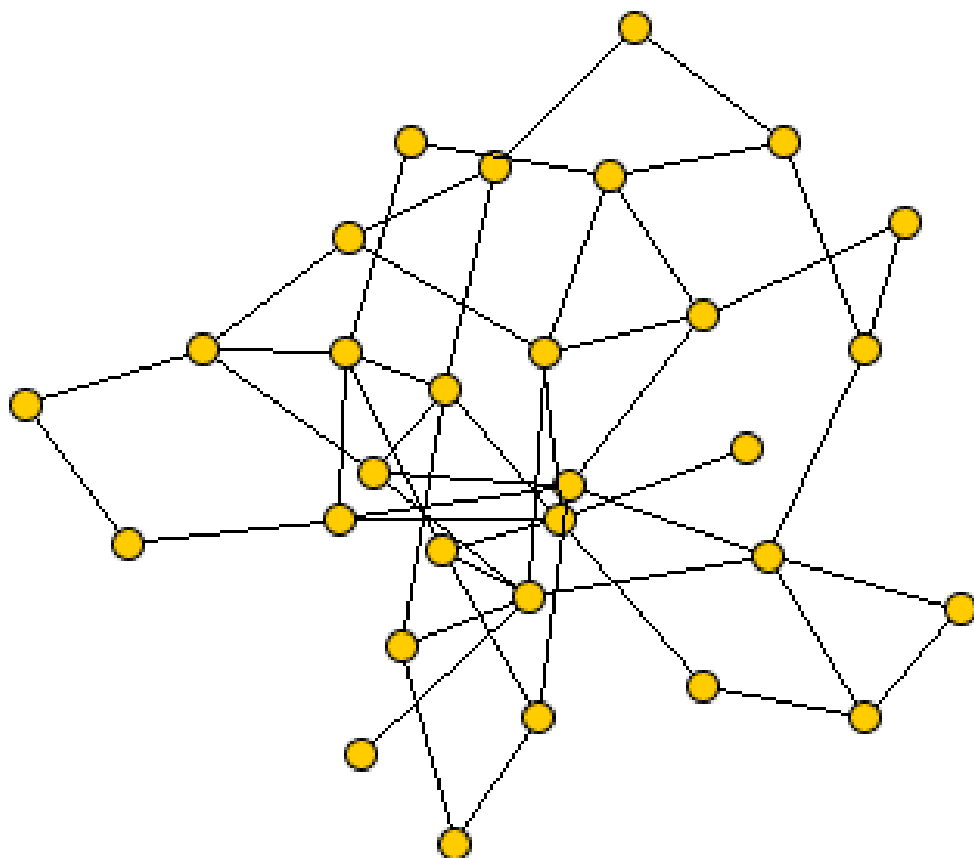


Рисунок 1.4 – Приклад розміщення графа за допомогою методу П.Ідса
(методу взаємодії сил)

2 РОЗРОБКА АЛГОРИТМУ ТА ВИБІР ЗАСОБІВ РЕАЛІЗАЦІЇ

2.1. Постановка задачі

Розібравшись з наведеними у теоретичній частині методами побудови решітки, було звернено увагу, що загальний вигляд побудованих графів – решіток не виконує більшість вимог. Тому, для практичної частини було обрано рішення розробити власний алгоритм, який би виконував умови більшості вимог та міг побудувати граф – решітку таким чином, щоб граф був би зрозумілим кінцевого користувача та щоб була можливість відредагувати або оптимізувати його.

Задача полягає в тому, щоб розробити програму, яка зможе зобразити решітку, дотримуючись таких критеріїв:

- Елементи решітки розміщені не в хаотичному порядку. Розбити елементи по рівням таким чином, щоб у кожного рівня був свій ряд елементів,
- Висота між рядами елементів розрахована відносно розміру вікна програми, щоб жоден елемент не вийшов за межі,
- Після запуску програми на екрані зображено лише елементи решітки та стрілки, які зображають зв'язок між елементами,
- Додаткова інформація повинна бути схована всередині елементів. Розробити програму таким чином, щоб інформація виводилась на екран лише тоді, коли це захоче користувач,
- Доступ до інформації елементів отримувати за допомогою курсору. Це може бути або клік або наведення курсору на елемент,
- Для перевірки правильності зв'язків, інформація повинна мати в собі назву елементів та бінарні строки,
- Зовнішній вигляд результату повинен бути презентабельним, мати приємну кольорову гаму, а кожен елемент програми буде чітко видно,
- Розміри елементів розраховуються автоматично, в залежності від кількості елементів та розміру вікна,

2.2. Розробка власного алгоритму

Розглянемо кожен крок алгоритму та опишемо очікуваний результат кожного етапу програми.

Перший крок – генерація комбінацій бінарних строк. В результаті очікуємо повну комбінацію всіх можливих бінарних строк для будь – якої кількості елементів. Для зручності сформувати матрицю (масив) з усіма комбінаціями.

Другий крок – пошук зв'язків між бінарними строками. Треба розробити таку функцію, яка б перевіряла всі можливі зв'язки та перебирала всі елементи таким чином, щоб для кожного елемента було порівняння з усіма іншими елементами. В результаті отримуємо масив зв'язків. Спосіб збереження зв'язків – зберігати індекси пов'язаних бінарних строк у вигляді «індекс – індекс».

Наприклад, елементи 1 та 3 пов'язані, відповідно, зберігаємо зв'язок у вигляді строки «1–3». Можливість зберігати зв'язки таким чином залежить від обраної мови програмування, тому, можливо, що на стадії практичної частини буде вигадано інший спосіб зберігання зв'язків.

Третій крок – розмістити всі елементи решітки у вікні програми. Перед тим, як розміщувати елементи, треба розрахувати ширину та висоту для кожного ряду решітки. Отримавши необхідні дані щодо ширини та висоти, почнемо розміщувати елементи в рядок. Щоб визначити, до якого ряду належить елемент, підрахуємо суму бінарної строк цього елемента. Наприклад, сума строки «0 1 1» дорівнює двом. Відповідно, даний елемент буде розміщено в другому ряді.

Четвертий крок – поєднати пов'язані між собою елементи стрілками. Наприклад, маємо строку зв'язку «1–3». Розіб'ємо її на частини (масив символів) за допомогою символу «←→». Відповідно, отримаємо масив чисел –

індексів у вигляді [«1», «3»]. Дані числа будуть показувати індекси елементів, які треба поєднати.

Для кращого сприйняття розглянемо блок – схему першої версії власного алгоритму на рисунку 2.1.

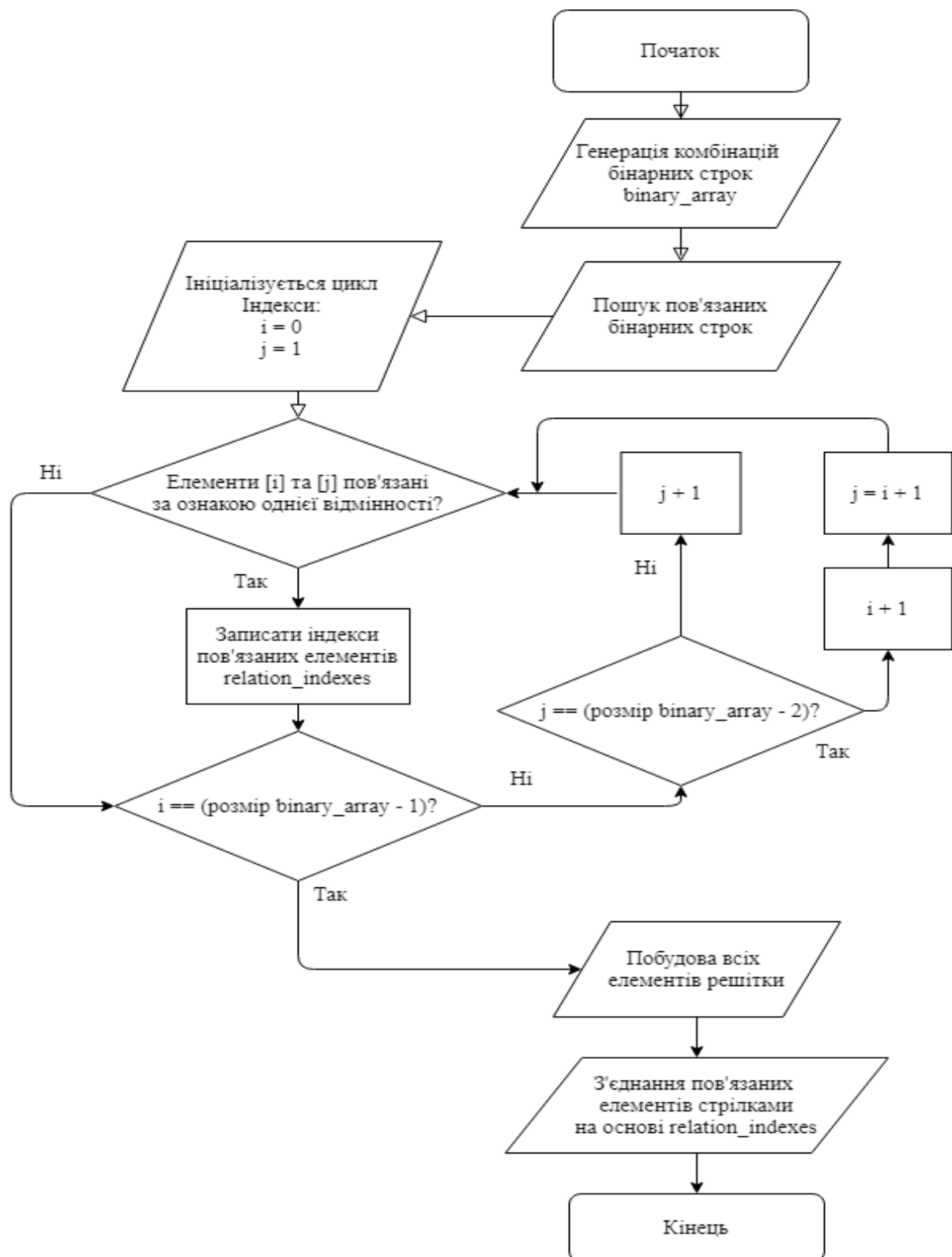


Рисунок 2.1 – Блок – схема алгоритму

2.3. Вибір мови програмування

Для успішної реалізації алгоритму, який будуватиме графічне зображення решітки формальних понять, потрібно обрати гнучку та надійну мову програмування, яка пройшла перевірку часом, має власну широку аудиторію користувачів та має велику кількість додаткових бібліотек. Першою тестовою мовою для реалізації алгоритму була РНР версії 7.2. Дана мова була обрана заради розробки певного прототипу алгоритму та для перевірки, чи можливо взагалі зобразити решітку формальних понять у будь – якому вигляді. В результаті, спираючись на блок – схему, було програмно реалізовано власний алгоритм, який генерує комбінації бінарних строк та будує решітку формальних понять. На рисунку 2.2 розглянемо першу версію побудованої решітки.

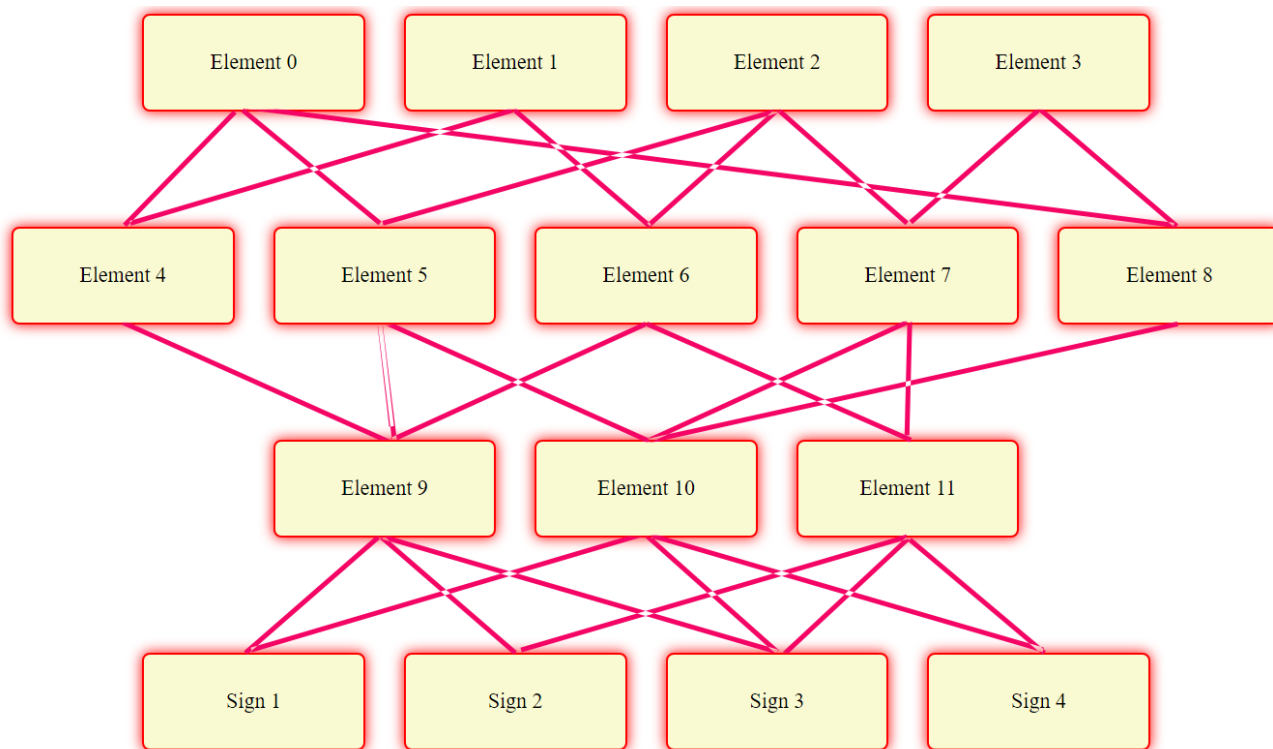


Рисунок 2.2 – Перша версія решітки

Незважаючи на те, що логічна частина програми працювала ідеально та виконувала всі кроки, які були описані в розділі 2.1, була одна важлива

проблема, яка заважала використовувати далі мову РНР, а саме – відсутність графічних бібліотек, які будуть коректно працювати. Були бібліотеки, які можна було підключити, але вони були реалізовані мовою С++, а сам процес підключення витрачав занадто багато часу. Тому було прийнято рішення перенести алгоритм з мови РНР на С++.

Незважаючи на те, що алгоритм було успішно перенесено на С++, сама мова потребує підготовки та не пробачає помилок програміста. Тож, як підсумок, розглянемо переваги та недоліки даної мови.

Переваги:

- Велика аудиторія користувачів, за допомогою якої можна знайти багато рішень проблеми,
- Велика кількість бібліотек (в тому числі й графічних) та зручна документація до них,
- Мова, яка перевірена часом та має за плечима потужні проекти в сфері ігрової індустрії,
- Строга типізація для більшого контролю над змінними,

Недоліки:

- Потребує багато часу на вивчення,
- Має багато нюансів, з якими може не впоратись починаючий програміст,
- Потребує обробки помилок для уникнення припинення роботи всієї програми,
- Складний процес роботи з динамічною пам'яттю,

Незважаючи на складність мови, вибір зупиняється саме на ній через наявність великої кількості графічних бібліотек, одну з яких і буде використано в реалізації алгоритму.

2.4. Вибір середовища розробки

Для розробки програми важливо мати таке середовище розробки, яке дозволяє налаштувати робочий процес під користувача, а також надає можливість легко запустити та відладити код. Було обрано найпопулярніше середовище від компанії Microsoft під назвою Visual Studio.

Visual Studio – це стартовий майданчик для написання, налагодження і складання коду, а також подальшої публікації додатків. Інтегроване середовище розробки являє собою багатофункціональну програму, яку можна використовувати для різних аспектів розробки програмного забезпечення. Крім стандартного редактора і відладчика, які існують в більшості середовищ, Visual Studio включає в себе компілятори, засоби автозавершення коду, графічні конструктори і багато інших функцій для спрощення процесу розробки. На сьогоднішній день Visual Studio є одним з кращих засобів розробки додатків. З кожною новою версією це середовище набуває все більше і більше корисних функцій, але, при цьому, стає все складніше і складніше, тим самим відлякуючи початкових програмістів. Навіть багато професіоналів не використовують усіх можливостей, що значно ускладнює роботу.

Хоч і Visual Studio має гнучкі налаштування та широкий набір інструментів, для реалізації алгоритму буде достатньо базового набору у вигляді компілятора коду та функції відладки. Тому, вибір зупиняється саме на ньому.

2.5 Вибір графічної бібліотеки

Мова C++ має великий вибір гнучких графічних бібліотек, тому вибір відбувався спираючись на зручність документації та час, який потрібно витратити на її вивчення. Однією з таких бібліотек є SFML.

SFML (англ. Simple and Fast Multimedia Library – проста і швидка мультимедійна бібліотека) – вільна кросплатформна мультимедійна бібліотека. SFML містить ряд модулів для простого програмування ігор і мультимедіа додатків. В даний час доступні наступні модулі:

System – управління часом і потоками, він є обов'язковим, так як всі модулі залежать від нього.

Window – управління вікнами і взаємодією з користувачем.

Graphics – робить простим відображення графічних примітивів і зображень.

Audio – надає інтерфейс для управління звуком.

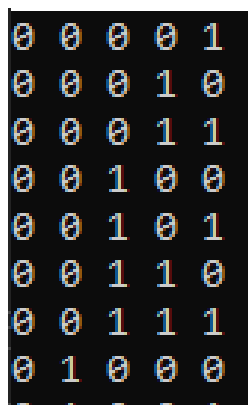
Network – для мережевих додатків.

Незважаючи на те, що дана бібліотека розрахована на розробку ігор, її ресурси можна використати для побудови решітки. SFML часто використовується в невеликих стартапах або програмістами, які самі займаються розробкою ігор. Інструмент є досить популярним через те, що не потребує написання об'ємного коду. Приємним бонусом була наявність підтримки популярних платформ. Також дана бібліотека легко підключалась та налаштовувалась в обраному середовищі розробки Visual Studio. А найбільшим плюсом є низька потреба до рівня володіння мовою програмування. Тому, дана бібліотека була ідеальним кандидатом для реалізації алгоритму зображення решітки.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ РОБОТИ АЛГОРИТМУ

3.1 Визначення вхідних параметрів

Основою для перевірки наявності зв'язків між елементами решітки являються бінарні відносини. Тому, спочатку треба згенерувати масив з усіма можливими комбінаціями 0 та 1 для обраної кількості елементів решітки. Для прикладу виберемо 5 елементів. Запустимо компіляцію коду для генерування масиву комбінацій бінарного коду та виведемо кінцевий результат вхідних даних за допомогою консолі (рисунок 3.1).



0	0	0	0	1
0	0	0	1	0
0	0	0	1	1
0	0	1	0	0
0	0	1	0	1
0	0	1	1	0
0	0	1	1	1
0	1	0	0	0

Рисунок 3.1 – Згенерований масив комбінацій бінарного коду для 5 елементів

Так, як вхідні дані були успішно згенеровані, можна переходити до наступного кроку – рівномірної побудови елементів решітки.

Основним завданням дипломної роботи для етапу побудови решітки було: побудувати граф таким чином, щоб його елементи були розташовані не в хаотичному порядку, а сама решітка мала приємний та презентабельний вигляд. Цю проблему вдалося розв'язати за допомогою розташування елементів решітки по рівням. Було встановлено умову, що рівень – це сума одиниць у бінарному коді одного елемента. Відповідно, якщо для прикладу було обрано число 5, то і максимальний рівень елемента буде дорівнювати 5. Перед тим, як

побудувати решітку, треба розрахувати відповідні рівні для кожного елемента. Тому, далі виконується функція *calculateCountOfElementsOnEachLevel()*, яка виконує підрахунок елементів на кожному рівні. Дана функція має такий вигляд:

```
void Algorithm::calculateCountOfElementsOnEachLevel()
{
    int binaryArraySize = binaryArray.size();

    for (int j = 0; j < binaryArraySize; j++) {
        int rowSum = this->calculateRowSum(binaryArray[j]);

        if (rowSum == 0) {
            continue;
        }

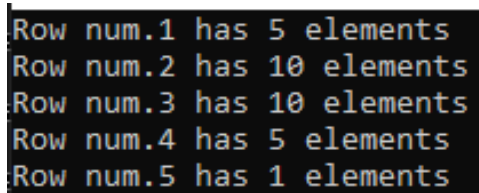
        this->countOfElementsOnEachLevel[rowSum - 1]++;
    }
}
```

Варто звернути увагу на строку функції:

this->countOfElementsOnEachLevel[rowSum - 1],

де від суми бінарної строки віднімається одиниця, а результат даної різниці є індексом масиву. Дана різниця була реалізована для виконання правила індексації масивів. В C++ індекси масивів починаються з 0, а сума бінарних строк починається з 1. Тому, виходить, що масив *countOfElementsOnEachLevel* з індексом 0 поверне значення кількості елементів на першому рівні.

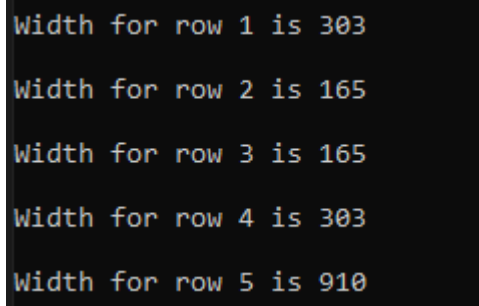
Розглянемо результат роботи методу за допомогою консолі (рисунок 3.2).



```
Row num.1 has 5 elements
Row num.2 has 10 elements
Row num.3 has 10 elements
Row num.4 has 5 elements
Row num.5 has 1 elements
```

Рисунок 3.2 – Результат роботи функції підрахунку кількості елементів на кожному рівні

Отримавши результат, наступним кроком запускається алгоритм для розрахунку ширини та висоти між елементами решітки. Розглянемо приклад на основі вікна програми з роздільною здатністю 1920x1080. Розрахуємо ширину між елементами решітки. Для розв'язання даної задачі візьмемо ширину вікна 1080, та поділимо її на кількість елементів кожного рівня. У результаті буде отримано такий масив (рисунок 3.3).



```
Width for row 1 is 303
Width for row 2 is 165
Width for row 3 is 165
Width for row 4 is 303
Width for row 5 is 910
```

Рисунок 3.3 – Розрахована ширина між елементами решітки для кожного рівня

Для розрахунку висоти між рівнями елементів спочатку розраховується загальна висота. Для цього виберемо значення висот вікна, яке дорівнює 1920 та поділимо на кількість елементів, яка дорівнює 5. Отримаємо число 384, яке і буде висотою між рядками елементів. Розрахунки ширини та висоти для елементів завершені, тому наступний етап – знайти зв'язки між бінарними строками та зберегти розраховані зв'язки для побудови ліній, які будуть поєднувати пов'язані між собою елементи.

3.2 Знаходження зв'язків між елементами решітки

Сам алгоритм працює таким чином: бінарний код кожного елементу попарно порівнюється з іншими бінарними кодами елементів решітки. Зв'язок між елементами решітки буде перевірятися за декількома умовами. Спочатку перевіряється сума одиниць в бінарних кодах в елементах, які порівнюються між собою. Якщо сума одиниць першого елемента більша суми одиниць другого елемента, то ми повинні перевірити, що елементи мають зв'язок

завдяки одній відмінності у бінарних кодах. Щоб перевірити наявність однієї відмінності, треба використати функцію *hasOneDifferenceInRows()*. Ця функція перевіряє порядок розміщення нулів та одиниць в обох бінарних масивах. Для того, щоб розібратися в функціях, які перевіряють наявність зв'язку між елементами, ми розберемо один крок роботи алгоритму.

Обираємо такі елементи для порівняння:

[елемент 1]:[1 0 0 0 0]

[елемент 3]:[1 1 0 0 0]

Сума одиниць бінарного коду першого елемента дорівнює 1. Сума одиниць другого елемента дорівнює 2. Так, як суми відрізняються лише на одиницю, то перевірка продовжується і наступним кроком обрані два елемента перевіряються на наявність однієї відмінності. Тому, далі викликається функція *hasOneDifferenceInRows()*. Ця функція має вигляд:

```
bool Algorithm::hasOneDifferenceInRows(vector<int> mainRow, vector<int>
rowToCompare)
{
    int countOfEqualities = 0;
    int sizeOfRow = mainRow.size();

    for (int i = 0; i < sizeOfRow; i++) {
        if (mainRow[i] == rowToCompare[i]) {
            countOfEqualities++;
        }
    }

    return countOfEqualities == (sizeOfRow - 1);
}
```

Масив *mainRow* – елемент з індексом [1], масив *rowToCompare* – елемент з індексом [3]. Спочатку ініціалізується лічильник зі значенням 0. Далі виконується цикл, котрий порівнює значення елементів на однаковій позиції. Якщо елементи першого та другого масивів на однаковій позиції однакові, то лічильник збільшує своє значення на 1.

Ітерація 0:

Масив 1 -> перший елемент масиву = 1

Масив 2 -> перший елемент масиву = 1

Елементи масиву однакові, значення лічильника збільшується на 1.

Ітерація 1:

Масив 1 -> другий елемент масиву = 0

Масив 2 -> другий елемент масиву = 1

Елементи відрізняються, значення лічильника не змінюється

Ітерація 2:

Масив 1 -> третій елемент масиву = 0

Масив 2 -> третій елемент масиву = 0

Елементи масиву однакові, значення лічильника збільшується на 1.

Ітерація 3:

Масив 1 -> четвертий елемент масиву = 0

Масив 2 -> четвертий елемент масиву = 0

Елементи масиву однакові, значення лічильника збільшується на 1.

Ітерація 4:

Масив 1 -> n'ятий елемент масиву = 0

Масив 2 -> n'ятий елемент масиву = 0

Елементи масиву однакові, значення лічильника збільшується на 1.

Робота циклу завершується, значення лічильника – 4.

Наступний крок перевірки – порівняти значення лічильника та розміри бінарного масиву. Якщо два елементи мають одну відмінність, то від кількості елементів масиву віднімається одиниця і якщо ця різниця дорівнює значенню лічильника, то елементи мають одну відмінність. В нашому випадку:

Кількість елементів в масиві – 5. Для порівняння треба від кількості елементів відняти 1 та порівняти зі значенням лічильника.

Розмір масиву $5 - 1 =$ знач. лічильника 4.

Умова виконується, а це означає, що обрані елементи мають зв'язок. Алгоритм не будує ці зв'язки одразу, а лише зберігає ці дані в результуючому масиві зв'язків для можливості побудування одразу всіх зв'язків в кінці роботи алгоритму. Виведемо результуючий масив зв'язків за допомогою консолі (рисунок 3.4).

```
1-3
1-5
1-9
1-17
2-3
2-6
2-10
2-18
3-7
3-11
```

Рисунок 3.4 – Результуючий масив зв'язків

Ці зв'язки виводяться на екран для того, щоб була можливість перевірити, чи правильно з'єднані між собою елементи на решітці. Так, як було згенеровано всі можливі комбінації бінарного коду для п'яти елементів, то на решітці будуть з'єднані всі елементи.

Наступний етап - графічне зображення решітки за допомогою графічної бібліотеки. Але, перед тим, як перейти до етапу зображення, перевіримо всі вхідні дані. В результаті, маємо такі дані:

vector <int> widthBetweenGridElements; - масив ширин для кожного рівня
vector <int> heightBetweenGridElements; - масив висот для кожного рівня
vector <string> relationStrings; - масив строк – зв'язків між елементами решітки
vector <VertexArray> connectionLines; - масив для ліній, які поєднують пов'язані елементи (об'єкти класів графічної бібліотеки)
vector <GridElement> gridElements; - масив для фігур елементів решітки (об'єкти класів графічної бібліотеки)

Всі необхідні дані було згенеровано, тому можна переходити до наступного етапу – графічного зображення решітки.

3.3 Графічне зображення решітки

Для коректної роботи бібліотеки, спочатку виконується цикл, в якому відбувається перевірка за допомогою функції з графічної бібліотеки, чи відкрите зараз вікно програми. Цикл має такий код:

```
while (window->isOpen()) {
    Event event;

    while (window->pollEvent(event)) {
        if (event.type == sf::Event::Closed ||
            Keyboard::isKeyPressed(Keyboard::Escape)) { // Перевірка, чи натиснуто клавішу Esc
            window->close();
        }
    }

    window->clear();
    window->draw(bg); { //Виклик функції для зображення фону

    for (VertexArray connectionLine : connectionLines) {
        window->draw(connectionLine); // Зображення ліній для
        поєднання пов'язаних елементів
    }
}
```

```

    }
    for (CircleShape arrow : arrows) {
        window->draw(arrow); // Зображення напрямків для ліній
    }
    for (GridElement gridElement : gridElements) {
        window->draw(gridElement.getElementShape()); //
Зображення елементів решітки
    }

    for (GridElement gridElement : gridElements) { // Зображення
інформації елементів
        if
(checkIsMouseInShapeArea(gridElement.getElementShape())){
            window->draw(gridElement.getHoverInfoBox());
            window->draw(gridElement.getElementLabel());
            window->draw(gridElement.getBinaryRowString());
        }
    }
    window->display();

```

Спочатку виконується функція, яка виводить робоче вікно на екран. Графічна бібліотека дозволяє налаштовувати вікно та встановлювати будь-яку картинку в якості фонового зображення. В результаті отримаємо вікно з фоновим зображенням (рисунок 3.5).



Рисунок 3.5 – Вікно програми з фоновим зображенням

Наступним кроком будуються елементи решітки. Так, як потрібна висота та ширина були розраховані заздалегідь, то процес побудови буде простим, а всі елементи будуть розміщені рівномірно та не в хаотичному порядку. Якщо алгоритми розрахунку ширини та висоти були реалізовані правильно, то ми отримаємо зображення елементів відповідно до вимог. Переконаємось, чи вірні розміри були отримані за допомогою алгоритму та розглянемо, як саме були побудовані елементи решітки (Рисунок 3.6).



Рисунок 3.6 – Побудовані елементи решітки

Наступний крок – побудова ліній, які поєднують пов’язані між собою елементи. Самі лінії прив’язуються до координати середини кожного елемента. Трохи складнішою задачею було зобразити напрямок, тим самим перетворити лінію на стрілку. Для реалізації напрямку стрілки було взято просту фігуру – рівнобічний трикутник. Так, як відомо, що всі стрілки прямують зверху вниз, то вершину трикутника одразу було направлено вниз за допомогою коду. Головна

проблема – дізнатись, під яким кутом повернути трикутник, щоб він правильно вказував напрямок лінії. Для цього було використано такі формули:

$$\arctg\left(\frac{len1}{len2}\right) \cdot \frac{180}{\pi}, \quad \text{– для розрахунку кута напрямку лінії.}$$

Як саме було використано цю формулу, продемонстровано на рисунку 3.7 та на рисунку 3.8.

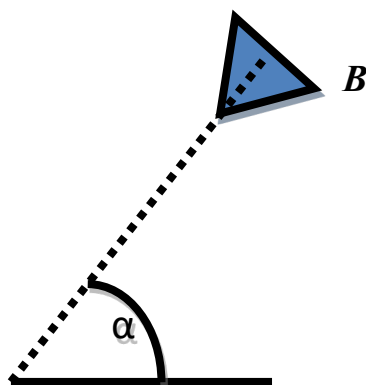


Рисунок 3.7 – Кут, під яким нахилено лінію зв'язку

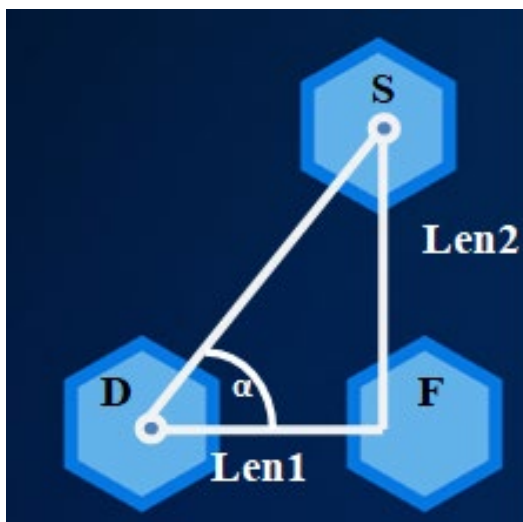


Рисунок 3.8 – Сторони прямокутного трикутника для розрахунку кута α

Фігура В є додатковою фігурою, щоб у комбінації з лінією утворити стрілку. Дані зображення демонструють, що фігуру В треба повернути під кутом α навколо свого центру для того, щоб вершина фігури В лежала на лінії SD.

Формули для розрахунку позиції трикутника:

$$X_a - \frac{radius}{\sqrt{(X_a - X_b)^2 + (Y_a - Y_b)^2}} \cdot (X_a - X_b),$$

$$Y_a - \frac{radius}{\sqrt{(X_a - X_b)^2 + (Y_a - Y_b)^2}} \cdot (Y_a - Y_b),$$

Основою формули для розрахунку позиції трикутника було взято формулу для розрахунку відстані між двома точками:

$$C = \sqrt{(X_a - X_b)^2 + (Y_a - Y_b)^2},$$

Формулу було перероблено таким чином, щоб прокласти відрізок від центру елемента по лінії, яка поєднує елементи, та знайти точку, де потрібно розмістити трикутник. В даному випадку, цим відрізком слугує радіус елемента. На рисунку 3.9 можна детально розглянути, яку саме точку знаходить формула.

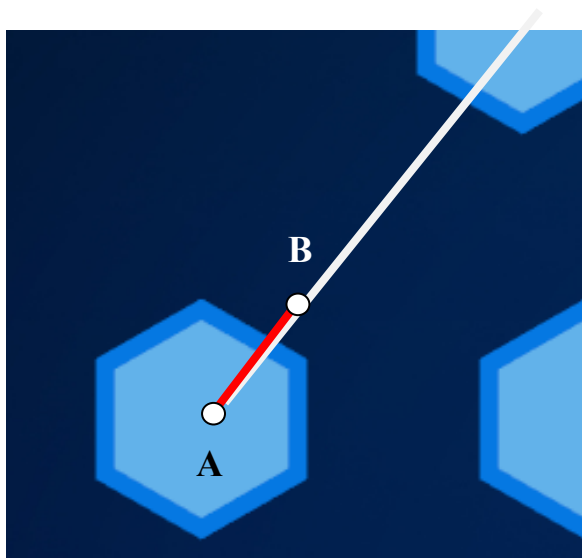


Рисунок 3.9 – Точка для розміщення трикутника

Відрізок AB є радіусом шестикутника (графічна бібліотека дозволяє зберігати радіус для фігур). А сам трикутник потрібно розмістити в розрахованій за допомогою формули точці B . Розглянемо, чи правильно була розрахована точка для розміщення трикутника. Демонстрація роботи побудови лінії та трикутника, які утворюють стрілку, розглянемо рисунок 3.10.

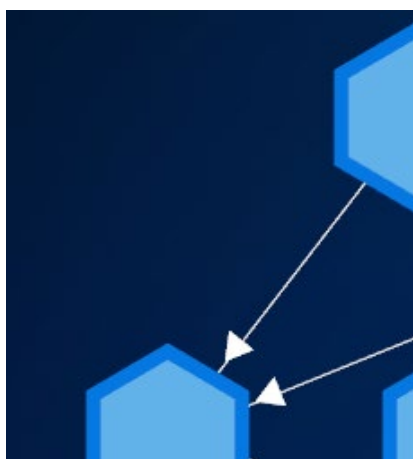


Рисунок 3.10 – Побудована стрілка для поєднання елементів

Так, як були побудовані всі необхідні елементи решітки, то на рисунку 3.11 можна розглянути фінальний вигляд побудованої решітки формальних понять.

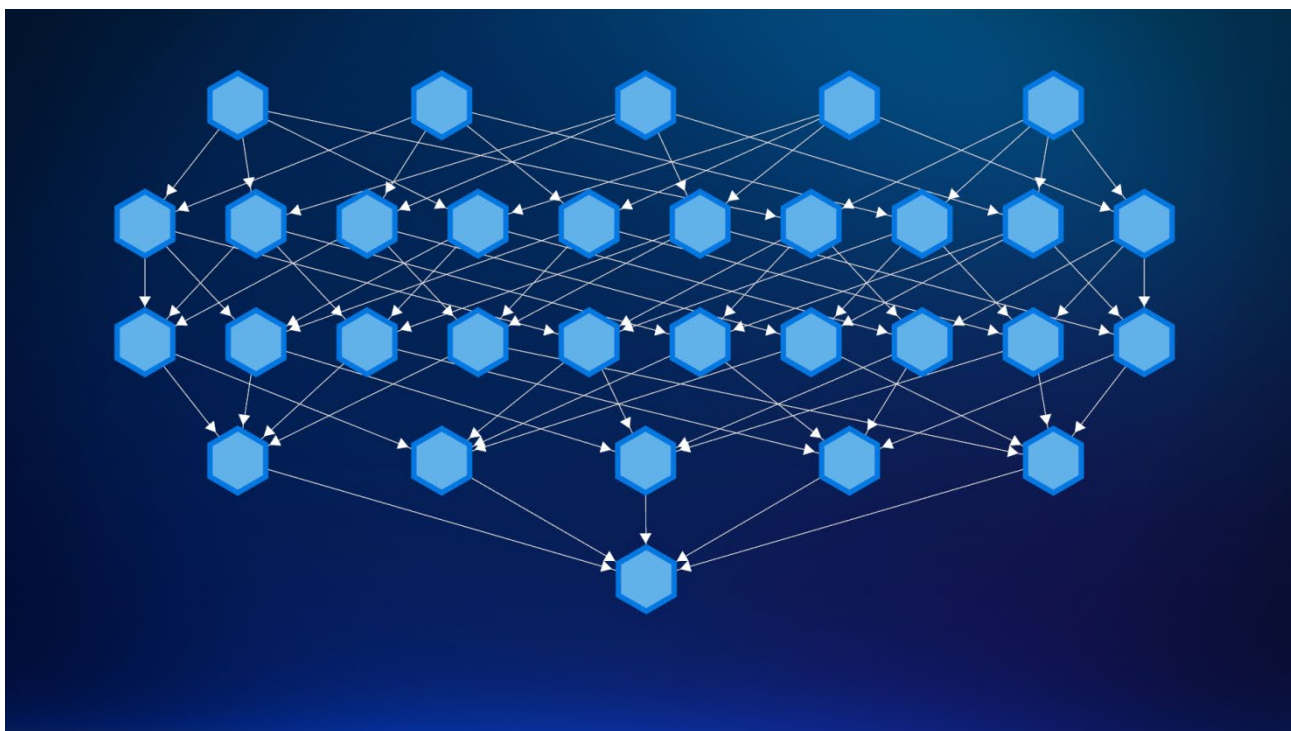


Рисунок 3.11 – Фінальний вигляд побудованої решітки

Відповідно до постановки задачі, фінальний вигляд решітки має презентабельний вигляд. Всі елементи побудовані рівно в ряд, а відстань між ними рівномірна. Але, суть решітки полягає в тому, яку інформацію несе в собі кожен елемент. Щоб дізнатися, яку інформацію несе в собі кожен елемент, достатньо навести курсор на один з елементів. На екран буде виведено спеціальну додаткову фігуру в вигляді прямокутника, на якому буде зображено назву та бінарну строку елемента (рисунок 3.12).



Рисунок 3.12 – Додаткова інформація елемента решітки

Щоб перевірити, чи правильно були побудовані стрілки, скористаємось додатковою інформацією елементів. В розділі 3.3 було відомо, що елементи з індексами 1 та 3 мають зв'язок. Тому, вони повинні бути поєднані стрілкою. На рис.16 дізнаємось, чи поєднані елементи з індексами 1 та 3.

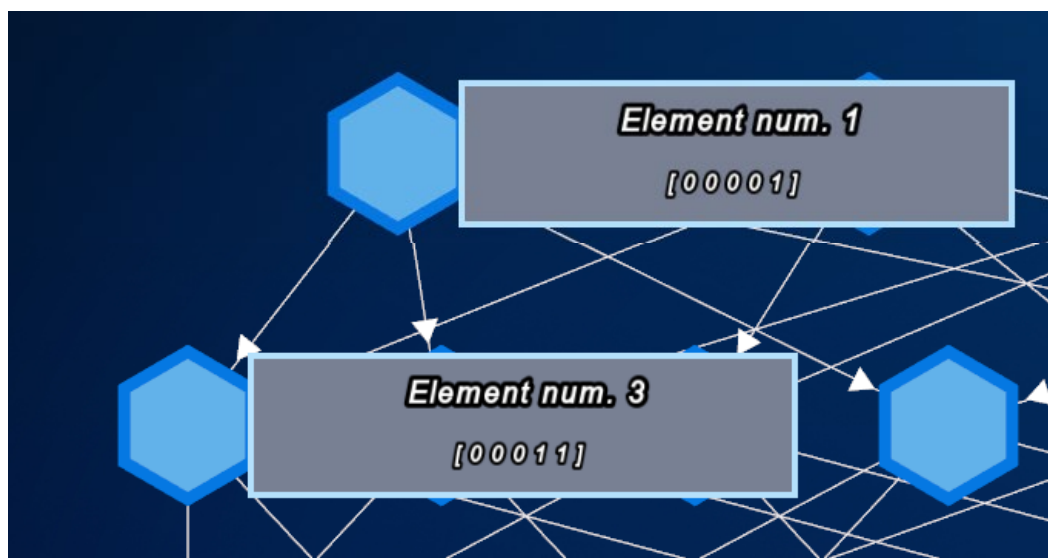


Рисунок 3.13 – Демонстрація зв'язку між елементами 1 та 3

Отже, стрілки поєднують лише пов'язані елементи, а алгоритм пошуку зв'язків впорався з завданням. Перевірка за допомогою додаткової інформації елементів підтвердила правильність роботи програми. Тому, можна ствердити, що програма правильно будує решітку.

ВИСНОВОК

В дипломній роботі було розглянуто проблеми побудування решітки формальних понять. В теоретичній частині було розглянуто декілька методів та як виглядає решітка в результаті роботи цих методів. Для практичної частини було розроблено алгоритм, який задовольняє вимоги щодо побудування решіток:

- Ребра зображені у вигляді прямих ліній,
- Елементи не розташовані в одній точці,
- Елементи решітки розміщені не в хаотичному порядку,
- Після запуску програми на екрані побудовано лише елементи решітки та ребра графа,
- Інформацію кожного елемента решітки сховано всередині,
- Доступ до інформації отримується за допомогою курсору,
- Розмір елементів, висота та ширина розраховуються автоматично та не потребує втручання користувача.

Також, для демонстрації можливостей програми, було розроблено генератор комбінацій бінарних строк, щоб отримати повний граф – решітку, та розглянути, як програма впорається з повним графом - решіткою. Як свідчить результат - алгоритм виконав поставлене завдання, а отримане зображення відповідає до поставлених вимог.

ПЕРЕЛІК ПОСИЛАНЬ

1. **Ігнатов, Д.І.** Аналіз формальних понять: від теорії до практики [Електронний ресурс] / Д.І.Ігнатов, – Відкриті системи, 2012. Режим доступу: <https://publications.hse.ru/mirror/pubs/share/folder/8yy8txxwr7/direct/72561425>
2. **Таран, Т.А.** Основи дискретної математики [Текст] / Т.А. Таран. – Київ: Просвіта, 2003. – 74 с.
3. **Ganter B.** Formal Concept Analysis: Mathematical Foundations/ B. Ganter, R. Wille. – New York: Springer-Verlag. – Secaucus. Inc. – 1999. – pp. 284
4. **Сігуюма, К.** Методи для візуального розуміння ієрархічних систем [Текст] / К. Сігуюма, – 1981. – с.109 – 125.
5. **ДСТУ 3008-95.** Документація. Звіти у сфері науки і техніки. Структура і правила оформлення [Текст] . – К. –1995.
6. **Вілліс, Р.** Набуття знань методами формального аналізу понять [Текст] / Р. Вілліс. – 1989. – с.445 – 470.
7. **Кедров, С. А.** Дослідження груп користувачів інтернет – ресурсами методами аналізу формальних понять та розробки даних, (Data Mining) [Текст] / С. О. Кузнецов, С. А. Кедров. – Москва: Бізнес – інформатика, – 2007. – с.45 – 51.
8. **Біркгоф, Г.** Теорія решіток. [Текст] / Г. Біркгоф. – М.: Наука, – 1984. – 568 с.
9. **Vychodil V.** A new algorithm for computing formal concepts / V. Vychodil / In Proceedings of the 19th European Meeting on Cybernetics and Systems Research, Vienna. – 2008. – P. 15-21.
10. **Скорняков Л. А.** Елементи теорії структур. [Текст] – Москва: Наука, 1970. – 148 с.
11. **Житомирський Г. И.** Впорядковані множини і решітки. [Текст] – Саратов: СГУ, 1981. – 101 с.

12. Гретцер Г. Общая теория решёток.[Текст] – Москва: Мир, 1982. – 452 с.
13. **B. Ganter** and **S.O. Kuznetsov**, Pattern Structures and Their Projections, Proc. 9th Int. Conf. on Conceptual Structures, ICCS'01, G. Stumme and H. Delugach, Eds., Lecture Notes in Artificial Intelligence, 2120 2001, pp. 129-142..
14. **B. Ganter** and **K. Reuter**, Finding all closed sets: a general approach. Order, 8, 283-290, 1991.
15. **B. Ganter** and **S.O. Kuznetsov**, Formalizing Hypotheses with Concepts, Proc. 8th Int. Conf. on Conceptual Structures, ICCS'98, G. Mineau and B. Ganter, Eds., Lecture Notes in Artificial Intelligence, 1867, 2000, pp. 342-356.
16. **J. Hereth**, **G. Stumme**, **R. Wille**, and **U. Wille**, Conceptual Knowledge Discovery and Data Analysis, Proc. 8th Int. Conf. on Conceptual Structures, ICCS'98, G. Mineau and B. Ganter, Eds., Lecture Notes in Artificial Intelligence, 1867, 2000, pp. 421-437.

ДОДАТОК А

Програмний код

```

#include <SFML/Graphics.hpp>
#include <cstring>
#include <array>
#include <iostream>
#include <vector>
#include <string>
#include <sstream>
#include <cmath>
using namespace sf;
using namespace std;

const Color WINDOW_COLOR = Color(239, 237, 242);
const char* APP_MAIN_FONT = "ArialItalic.ttf";
const string IMAGE_BACKGROUND_PATH = "Textures/background.jpg";
const string IMAGE_GRID_ELEMENT_PATH = "Textures/grid_element.jpg";
const string IMAGE_GRID_ELEMENT_INFO_PATH = "Textures/element_info.jpg";
const int SIZE_FOR_BINARY_ARRAY = 5;

vector<int> widthBetweenGridElements;
vector<int> heightBetweenGridElements;
vector<int> currentWidthForElements;
static int indexOfElement = 0;
int elementRadius = 80;
Font font;
Texture textureBackground;
Sprite bg;

class Algorithm
{
private:
vector<vector<int> > binaryArray;
vector<string> relationStrings;
vector<int> countOfElementsOnEachLevel;
int size;

template <class type>
void freeVectorMemory(vector<type>& originalVector);
void free2dVectorMemory(vector<vector<int> >& originalVector);
int calculateRowSum(vector<int> currentRow);
bool hasOneDifferenceInRows(vector<int> mainRow, vector<int> rowToCompare);
bool checkIsSumAbleToCompare(int sum1, int sum2);
bool checkHasRelationBetweenArrayRows(vector<int> mainRow, vector<int> rowToCompare);
void generateBinaryArray();
void calculateCountOfElementsOnEachLevel();

```

```

public:
Algorithm(int size);
vector<string> launchAlgorithm();
vector<vector<int> > getBinaryArray();
vector<int > getCountOfElementsOnEachLevel();
~Algorithm();
};

template <class type>
void Algorithm::freeVectorMemory(vector<type>& originalVector)
{
vector<type>().swap(originalVector);
}

vector <int > Algorithm::getCountOfElementsOnEachLevel()
{
return this->countOfElementsOnEachLevel;
}

void Algorithm::free2dVectorMemory(vector<vector<int> >& originalVector)
{
vector<vector<int> >().swap(originalVector);
}

int Algorithm::calculateRowSum(vector <int> currentRow)
{
int sum = 0;

for (int value : currentRow) {
sum += value;
}

return sum;
}

bool Algorithm::hasOneDifferenceInRows(vector<int> mainRow, vector<int> rowToCompare)
{
int countOfEqualities = 0;
int sizeOfRow = mainRow.size();

for (int i = 0; i < sizeOfRow; i++) {
if (mainRow[i] == rowToCompare[i]) {
countOfEqualities++;
}
}

return countOfEqualities == (sizeOfRow - 1);
}

bool Algorithm::checkIsSumAbleToCompare(int sum1, int sum2)
{
bool boolValue;

```

```
(sum1 > sum2) ? boolValue = ((sum1 - 1) == sum2) : boolValue = ((sum2 - 1) == sum1);
```

```
return boolValue;
}
```

```
bool Algorithm::checkHasRelationBetweenArrayRows(vector<int> mainRow, vector<int>
rowToCompare)
{
if (checkIsSumAbleToCompare(
calculateRowSum(mainRow), calculateRowSum(rowToCompare))
&& hasOneDifferenceInRows(mainRow, rowToCompare))
{
return true;
}
}
```

```
return false;
}
```

```
void Algorithm::generateBinaryArray()
{
vector<vector<int>> > temp(size, vector<int>(1 << size));
unsigned numberToFill = 1U << (size - 1);

for (unsigned col = 0; col < size; ++col, numberToFill >>= 1U) {
for (unsigned row = numberToFill; row < (1U << size); row += (numberToFill * 2)) {
fill_n(&temp[col][row], numberToFill, 1);
}
}
}
```

```
vector<int> binaryRow;
```

```
for (unsigned x = 0; x < (1 << size); ++x)
{
for (unsigned y = 0; y < size; ++y)
{
//int val = temp[y][x];
binaryRow.push_back(temp[y][x]);
}
}
```

```
binaryArray.push_back(binaryRow);
freeVectorMemory(binaryRow);
}
}
```

```
Algorithm::Algorithm(int size)
{
this->size = size;
vector<int> temp(SIZE_FOR_BINARY_ARRAY, 0);
this->countOfElementsOnEachLevel = temp;
}
```

```
Algorithm::~~Algorithm()
```

```

{
free2dVectorMemory(this->binaryArray);
}

vector<string> Algorithm::launchAlgorithm()
{
vector<string> relationStrings;
generateBinaryArray();
int binaryArraySize = binaryArray.size();

for (int i = 1; i < (binaryArraySize - 1); i++) {
for (int j = 0; j < binaryArraySize; j++) {
if (checkHasRelationBetweenArrayRows(binaryArray[i], binaryArray[j]) && (i < j)) {
relationStrings.push_back(to_string(i) + "-" + to_string(j));
}
}
}

calculateCountOfElementsOnEachLevel();

for (int i = 1; i < (binaryArraySize); i++) {
for (int j = 0; j < this->size; j++) {
cout << binaryArray[i][j] << " ";
}

cout << endl;
}

return relationStrings;
}

vector<vector<int> > Algorithm::getBinaryArray()
{
return this->binaryArray;
}

void Algorithm::calculateCountOfElementsOnEachLevel()
{
int binaryArraySize = binaryArray.size();

for (int j = 0; j < binaryArraySize; j++) {
int rowSum = this->calculateRowSum(binaryArray[j]);

if (rowSum == 0) {
continue;
}

this->countOfElementsOnEachLevel[rowSum - 1]++;
}
int i = 1;

for (auto value : this->countOfElementsOnEachLevel) {

```

```

cout << "Row num." << i++ << " has " << value << " elements" << endl;
}
}

```

```

class GridElement
{
public:
    Text sfmlElementLabel;
    Text sfmlElementBinaryRow;
    RectangleShape hoverInformationPlace;
    CircleShape pentaShape;
    vector <int> binaryRow;
    int indexOfGridElement;

    GridElement(vector <int> binaryRow)
    {
        this->binaryRow = binaryRow;
        this->indexOfGridElement = indexOfElement++;
        createPentaShape();
        createElementInformationOnHoverEvent();
    }

    int calculateRowSum(vector <int> currentRow);
    int getIndexOfElement();

    void createPentaShape()
    {
        int indexOfRow = this->calculateRowSum(this->binaryRow);

        if (indexOfRow == 0) {
            return;
        }

        int currentWidthPx = currentWidthForElements[indexOfRow - 1];
        int heightPx = heightBetweenGridElements[indexOfRow - 1];
        int addWidthPxAfterCreationElement = widthBetweenGridElements[indexOfRow - 1];

        pentaShape.setRadius(elementRadius);
        pentaShape.setPointCount(6);
        pentaShape.setOutlineColor(Color(5, 120, 226));
        pentaShape.setFillColor(Color(98, 178, 234));
        pentaShape.setOutlineThickness(8);
        pentaShape.setPosition(Vector2f(currentWidthPx, heightPx));

        currentWidthForElements[indexOfRow - 1] += addWidthPxAfterCreationElement;
    }

    void createElementInformationOnHoverEvent()
    {
        int width = 200 + 30 * this->binaryRow.size();
    }
}

```

```

hoverInformationPlace.setPosition(this->pentaShape.getPosition().x + elementRadius * 2, this->pentaShape.getPosition().y);
hoverInformationPlace.setSize(Vector2f(width, elementRadius * 2));
hoverInformationPlace.setOutlineColor(Color(180, 224, 251));
hoverInformationPlace.setOutlineThickness(4);
hoverInformationPlace.setFillColor(Color(120, 128, 147));

createBinaryRowString();
createElementLabel();
}

void createBinaryRowString()
{
    string strRow;
    stringstream stream;

    for (int element : binaryRow) {
        stream << element << " ";
    }

    strRow = "[" + stream.str() + "]";
    sfmlElementBinaryRow.setFont(font);
    sfmlElementBinaryRow.setString(strRow);
    sfmlElementBinaryRow.setCharacterSize(15);
    sfmlElementBinaryRow.setFillColor(Color::White);
    sfmlElementBinaryRow.setOutlineColor(Color::Black);
    sfmlElementBinaryRow.setOutlineThickness(2);
    sfmlElementBinaryRow.setStyle(Text::Bold);
    sfmlElementBinaryRow.setPosition(Vector2f(this->hoverInformationPlace.getPosition().x + this->hoverInformationPlace.getSize().x / 2.5 - 10, this->hoverInformationPlace.getPosition().y + this->hoverInformationPlace.getSize().y * 0.6));
}

void createElementLabel()
{
    stringstream stream;
    stream << this->indexOfGridElement;
    sfmlElementLabel.setFont(font);
    sfmlElementLabel.setString("Element num. " + stream.str());
    sfmlElementLabel.setCharacterSize(20);
    sfmlElementLabel.setFillColor(Color::White);
    sfmlElementLabel.setOutlineColor(Color::Black);
    sfmlElementLabel.setOutlineThickness(3);
    sfmlElementLabel.setStyle(Text::Bold);
    sfmlElementLabel.setPosition(Vector2f(this->hoverInformationPlace.getPosition().x + this->hoverInformationPlace.getSize().x / 3.5, this->hoverInformationPlace.getPosition().y + this->hoverInformationPlace.getSize().y * 0.1));
}

CircleShape getElementShape()
{
    return pentaShape;
}

```

```

}

Text getElementLabel()
{
return this->sfmlElementLabel;
}

Text getBinaryRowString()
{
return this->sfmlElementBinaryRow;
}

RectangleShape getHoverInfoBox()
{
return this->hoverInformationPlace;
}
};

int GridElement::calculateRowSum(vector <int> currentRow)
{
int sum = 0;

for (int value : currentRow) {
sum += value;
}

return sum;
}

int GridElement::getIndexOfElement()
{
return indexOfGridElement;
}

class RenderApp
{
private:
RenderWindow* window;
Sprite background;
public:
RenderApp(ContextSettings &settings)
{
this->window = new RenderWindow(VideoMode::getDesktopMode(), "Grid", Style::Fullscreen,
settings);
loadMainBackground();
}

RenderWindow* getRenderWindow()

```

```

{
return this->window;
}
void loadMainBackground();
void calculateWidthBetweenGridElements(vector <int> countOfElementsOnEachRow);
void calculateHeightBetweenGridElements(int sizeOfBinaryArray);
void calculateElementRadius(int sizeOfBinaryArray);
};

void RenderApp::calculateElementRadius(int sizeOfBinaryArray)
{
int height = this->window->getSize().y;
int multiplier, calculatedRadius;

if (sizeOfBinaryArray > 1 && sizeOfBinaryArray <= 4) {
return;
}
else if (sizeOfBinaryArray > 4 && sizeOfBinaryArray <= 8) {
multiplier = sizeOfBinaryArray;
}
else {
multiplier = sizeOfBinaryArray * sizeOfBinaryArray * sizeOfBinaryArray;
}

calculatedRadius = height / (multiplier * sizeOfBinaryArray);

if (calculatedRadius < 10) {
elementRadius = 10;

return;
}

if (calculatedRadius < elementRadius) {
elementRadius = calculatedRadius;
}
}

void RenderApp::loadMainBackground()
{
Vector2u textureSize;
Vector2u windowSize;

if (!textureBackground.loadFromFile(IMAGE_BACKGROUND_PATH)) {
return;
}

textureSize = textureBackground.getSize();
windowSize = this->window->getSize();

float ScaleX = (float>windowSize.x / textureSize.x;
float ScaleY = (float>windowSize.y / textureSize.y;

```

```
bg.setTexture(textureBackground);
bg.setScale(ScaleX, ScaleY);
}
```

```
void RenderApp::calculateWidthBetweenGridElements(vector<int>
countOfElementsOnEachRow)
```

```
{
int width = this->window->getSize().x - 100;

for (int rowElementCount : countOfElementsOnEachRow) {
widthBetweenGridElements.push_back((int)width / (rowElementCount + 1));
}
```

```
currentWidthForElements = widthBetweenGridElements;
}
```

```
void RenderApp::calculateHeightBetweenGridElements(int sizeOfBinaryArray)
{
int stepOfHeight = (int)((this->window->getSize().y - 200) / (sizeOfBinaryArray));
int reduceStep = 0;
```

```
if (stepOfHeight >= 100) {
reduceStep = (int)(stepOfHeight - 100);
}
```

```
for (int i = 1; i <= sizeOfBinaryArray; i++) {
heightBetweenGridElements.push_back((stepOfHeight * i) - reduceStep);
}
}
```

```
vector<int> explodeStringBySymbol(string relationString) {
string temp;
vector<int> numbers;
stringstream stream(relationString);
```

```
while (getline(stream, temp, '-'))
{
numbers.push_back(stoi(temp));
}
```

```
return numbers;
}
```

```
VertexArray createConnectionBetweenTwoGridElements(CircleShape element1, CircleShape
element2)
```

```
{
Vector2f firstElemPosition = element1.getPosition();
Vector2f secondElemPosition = element2.getPosition();
int offsetWidth = element1.getRadius() - 3;
VertexArray lines(Lines, 2);
```

```

lines[0].position = Vector2f(firstElemPosition.x + element1.getRadius(), firstElemPosition.y +
element1.getRadius());
lines[1].position = Vector2f(secondElemPosition.x + element2.getRadius(), secondElemPosition.y
+ element2.getRadius());
lines[0].color = WINDOW_COLOR;
lines[1].color = WINDOW_COLOR;

return lines;
}

vector <VertexArray> resolveRelationStringsAndReturnConnectionLines(vector <string>
relationStrings, vector <GridElement> gridElements)
{
vector <VertexArray> connectionLines;
vector <int> numbersOfElements;

for (string relationString : relationStrings) {
numbersOfElements = explodeStringBySymbol(relationString);
connectionLines.push_back(createConnectionBetweenTwoGridElements(gridElements[numbersOf
Elements[0]].getElementShape(), gridElements[numbersOfElements[1]].getElementShape()));
}

return connectionLines;
}

int calculateAngleForArrowRotation(VertexArray connectionLine)
{
float len1, len2, angle;
int x1, x2;
len1 = abs(connectionLine[1].position.x - connectionLine[0].position.x);
len2 = abs(connectionLine[1].position.y - connectionLine[0].position.y);
x1 = connectionLine[0].position.x;
x2 = connectionLine[1].position.x;
angle = atan(len1 / len2) * 180 / 3.14;

if (x1 < x2) {
angle = 360 - angle;
}

return angle;
}

Vector2f calculateCoordinatesForArrow(VertexArray connectionLine)
{
int x1, x2, x3, y1, y2, y3;
x1 = connectionLine[0].position.x;
y1 = connectionLine[0].position.y;
x2 = connectionLine[1].position.x;
y2 = connectionLine[1].position.y;
int wrapperX = 13;
int wrapperY = 15;

```

```

if (x1 > x2) {
    wrapperX = -wrapperX;
    wrapperY = -wrapperY - 10;
}

x3 = x2 - (elementRadius + 20) / sqrt(pow((x2 - x1), 2) + pow((y2 - y1), 2)) * (x2 - x1);
y3 = y2 - (elementRadius + 20) / sqrt(pow((x2 - x1), 2) + pow((y2 - y1), 2)) * (y2 - y1);

return Vector2f(x3, y3);
}

vector<CircleShape> createArrowsForConnectionLines(vector<VertexArray> connectionLines)
{
    vector<CircleShape> arrows;
    Vector2f coords;
    CircleShape arrow(10, 3);
    arrow.setFillColor(Color::White);
    float len1, len2;
    int iter = 0;

    for (auto connectionLine : connectionLines) {
        coords = calculateCoordinatesForArrow(connectionLine);
        arrow.setPosition(calculateCoordinatesForArrow(connectionLine));
        arrow.setOrigin(10, 10);
        arrow.setRotation(180 + calculateAngleForArrowRotation(connectionLine));
        arrows.push_back(arrow);
        iter++;
    }

    return arrows;
}

void loadFont()
{
    font.loadFromFile(APP_MAIN_FONT);
}

bool checkIsMouseInShapeArea(CircleShape gridElement)
{
    int barrierX, barrierY;
    Vector2f elementPositions = gridElement.getPosition();
    barrierX = elementPositions.x + gridElement.getRadius() * 2;
    barrierY = elementPositions.y + gridElement.getRadius() * 2;

    if (Mouse::getPosition().x >= elementPositions.x && Mouse::getPosition().x <= barrierX &&
        Mouse::getPosition().y >= elementPositions.y && Mouse::getPosition().y <= barrierY) {
        return true;
    }

    return false;
}

```

```

int main()
{
    Algorithm binaryRelation(SIZE_FOR_BINARY_ARRAY);
    vector <string> relationStrings = binaryRelation.launchAlgorithm();

    ContextSettings settings;
    settings.antialiasingLevel = 8;
    RenderApp app(settings);
    RenderWindow *window = app.getRenderWindow();
    vector <VertexArray> connectionLines;

    loadFont();
    app.calculateWidthBetweenGridElements(binaryRelation.getCountOfElementsOnEachLevel());
    app.calculateHeightBetweenGridElements(SIZE_FOR_BINARY_ARRAY);
    app.calculateElementRadius(SIZE_FOR_BINARY_ARRAY);
    vector <GridElement> gridElements;

    for (int i = 0; i < binaryRelation.getBinaryArray().size(); i++) {
        GridElement element(binaryRelation.getBinaryArray()[i]);
        gridElements.push_back(element);
    }

    int iter = 1;

    for (auto value : relationStrings) {
        cout << endl << value << endl;
    }

    connectionLines = resolveRelationStringsAndReturnConnectionLines(relationStrings,
        gridElements);
    vector <CircleShape> arrows = createArrowsForConnectionLines(connectionLines);
    window->clear();
    app.loadMainBackground();

    while (window->isOpen()) {
        Event event;

        while (window->pollEvent(event)) {
            if (event.type == sf::Event::Closed || Keyboard::isKeyPressed(Keyboard::Escape)) {
                window->close();
            }
        }

        window->clear();
        window->draw(bg);

        for (VertexArray connectionLine : connectionLines) {
            window->draw(connectionLine);
        }

        for (CircleShape arrow : arrows) {

```

```
window->draw(arrow);
}

for (GridElement gridElement : gridElements) {
    window->draw(gridElement.getElementShape());
}

for (GridElement gridElement : gridElements) {
    if (checkIsMouseInShapeArea(gridElement.getElementShape())) {
        window->draw(gridElement.getHoverInfoBox());
        window->draw(gridElement.getElementLabel());
        window->draw(gridElement.getBinaryRowString());
    }
}

window->display();
}

return 0;
}
```