МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХЕРСОНСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
Факультет комп’ютерних наук, фізики та математики
Кафедра комп’ютерних наук та програмної інженерії

АЛГОРИТМИ ПОШУКУ НАЙКОРОТШИХ ШЛЯХІВ В ОРІЄНТОВАНИХ ГРАФАХ

Кваліфікаційна робота

на здобуття ступеня вищої освіти «бакалавр»

Виконав: студент 4 курсу 441 групи
Спеціальності: 121 Інженерія програмного забезпечення
Освітньо-професійної програми: Інженерія програмного забезпечення
Пилипенко М.А.
Керівник: проф. Вінник М.О.
Співкерівник: Співаковський О. В.
Рецензент: Котова О.В., доцент кафедри алгебри, геометрії та математичного аналізу ХДУ

Херсон – 2022

ЗМІСТ

ВСТУП3

РОЗДІЛ 1 ТЕОРЕТИЧНІ ВІДОМОСТІ З ТЕОРІЇ ГРАФІВ5

1.1 Основні визначення5

1.2 Основні різновиди графів5

6

1.4 Способи представлення графів в пам'яті комп'ютера6

1.5 Граф як абстрактний тип даних8

РОЗДІЛ 2 АЛГОРИТМИ ПОШУКУ НАЙКОРОТШИХ ШЛЯХІВ В ГРАФІ10

10

2.2 Алгоритм Беллмана-Форда13

2.3 Алгоритм Флойда-Воршелла15

2.4 Алгоритм Левіта17

РОЗДІЛ 3 РОЗРОБКА ПРОГРАМНОГО ЗАСОБУ ВІЗУАЛІЗАЦІЇ АЛГОРИТМІВ ПОШУКУ НАЙКОРОТШОГО ШЛЯХУ19

3.1 Опис вимог19

19

3.3 Короткий огляд використаних технологій21

23

3.5 Редактор графів24

26

ВИСНОВКИ30

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ31

ВСТУП

В епоху великих даних аналіз даних стає все більш важливим для підприємств. Традиційний аналіз OLAP з використанням діаграм широко використовується в діловому світі, щоб допомогти компаніям приймати бізнес-рішення. Це відоме як «BI» (Business Intelligence). Протягом останніх років розвиток технології 5G + IoT сприяє поширенню нового типу даних – графові дані. Наприклад, у будинку холодильник, кондиціонер, пральна машина, мобільний телефон і комп'ютер і навіть акаунти в соціальних мережах поєднані в єдиний граф за допомогою технологій Wi-Fi або 5G.

Обробка графових даних і аналіз топології графів – складна задача. Аналіз графів – це аналіз відношень і зв'язків.

На даний момент аналіз графів широко використовується в сферах запобігання фінансовому шахрайству, громадської безпеки, моніторингу інфраструктури та розумного медичного обслуговування. В цих умовах для вирішення задачі обробки і аналізу даних високої актуальності набуває задача створення потужних інструментів для обробки графів.

Для аналізу графових даних також є необхідним інструмент для візуалізації алгоритмів на графах.

В цьому контексті авторами роботи було прийнято рішення розробити додаток, що буде надавати можливості візуального аналізу графів, а також деяких алгоритмів на графах.

Об'єкт дослідження: технології візуалізації графових даних.

Предмет дослідження: додаток для візуалізації алгоритмів на графах.

Мета дослідження: розробка програмного засобу візуалізації алгоритмів на графах.

Досягнення зазначеної мети здійснюється шляхом вирішення таких **основних завдань**:

1. проаналізувати способи представлення графів в пам'яті комп'ютера та означити абстрактний тип даних графу;
2. проаналізувати основні алгоритми для вирішення задачі пошуку найкоротшого шляху в графі;
3. реалізувати бібліотеку компонентів, що повинна вирішувати задачі візуалізації алгоритмів пошуку найкоротшого шляху в графі;
4. на основі реалізованої бібліотеки створити програмний засіб для візуалізації алгоритмів на графах.

РОЗДІЛ 1

ТЕОРЕТИЧНІ ВІДОМОСТІ З ТЕОРІЇ ГРАФІВ

1.1 Основні визначення

Графом будемо вважати пару $G = (V, E)$, де V – множина вершин графа G, а E – множина елементів $\{x; y\}$, для яких $x, y \in V$, що називаються ребрами графа G[1, 2]. Якщо ребро $e = \{x, y\}$, кажуть, що e інцідентне вершинам x і y.

Нехай v – вершина графа $G = (V, E)$. Множиною сусідів v назвемо таку підмножину V' множини вершин G, що для кожної вершини $v' \in V'$ існує ребро $e = \{v; v'\}$, що належить E.

Підграфом G' графа $G = (V, E)$ називається такий граф $G' = (V', E')$, для якого виконуються наступні умови[1]:

- V' є підмножиною V;
- для кожного ребра $\{x; y\}$: $x, y \in V'$.

Обходом в графі G назвемо такий підграф $P = (V', E')$ графа G, для ребер якого існує така перестановка $e_1 e_2 \dots e_n$, що $e_i = \{v_i; v_{i+1}\}$[1]. Для обходів доцільний як запис $e_1 e_2 \dots e_n$, що відповідає зазначеній перестановці ребер, так і запис $v_1 v_2 \dots v_n$.

Шляхом в G називається такий обхід $P = v_1 v_2 \dots v_n$, в якому всі вершини відрізняються [1].

Цикл в G – це такий шлях $v_1 v_2 \dots v_n$, для якого $v_1 = v_n$[1].

1.2 Основні різновиди графів

В рамках даної роботи корисно визначити деякі різновиди графів, серед яких:

1. зважені графи;

2. орієнтовані графи.

Зваженим графом називаємо таку трійку $G = (V, E, \psi)$, для якої[1, 2]:

- (V, E) – граф;
- $\psi: E \to R$.

Для ребра e зваженого графу величина $\psi(e)$ називається вагою або довжиною.

Орієнтованим графом називається пара $G = (V, E)$, де де V – множина вершин графа G, а E – множина впорядкованих пар (x, y), для яких $x, y \in V$[1, 2].

1.3 Визначення задачі пошуку найкоротшого шляху

Нехай дано граф зважений $G = (V, E, \psi)$, а також шлях $P = e_1 e_2 \dots e_n$ в G. Величину $w = \sum \psi(e_i)$ назвемо довжиною шляху P[1, 2].

Набуває сенсу задача пошуку найкоротшого шляху, що формулюється наступним чином. Нехай дано зважений граф $G = (V, E, \psi)$ та дві вершини $x, y \in V$. Необхідно побудувати шлях $P = x v_1 v_2 \dots v_n y$, довжина якого буде мінімальною.

Відрізняють також наступні варіації задачі пошуку найкоротшого шляху[1, 2]:

- задача пошуку найкоротших шляхів з однієї вершини до усіх інших – з єдиним коренем;
- задача пошуку найкоротших шляхів для кожної пари вершин в графі.

1.4 Способи представлення графів в пам’яті комп’ютера

Існує декілька стандартних способів представлення графу $G = (V, E)$ в пам'яті комп'ютера, серед них виділимо наступні:

- в якості набору списків сусідів;
- в якості матриці суміжності.

Представлення графу $G = (V, E)$ у вигляді набору списків суміжних вершин містить масив A, який містить $V \vee$ списків – один на кожну вершину. Для кожної вершини $v \in V$ список суміжних вершин $A[v]$ складається з посилань (вказівників) на усі вершини, суміжні з v(усі u, для яких $\{v; u\} \in E$) у довільному порядку [1].

Для орієнтованого графа сума довжин усіх списків суміжності дорівнює $V \vee$, тоді як для неорієнтованого - $2 \vee V \vee$. В будь-якому разі обсяг необхідної пам'яті дорівнює $O(|V| + |E|)$[1].

Зважені графи зручно представити у вигляді списку сусідів, зберігаючи $\psi(e)$ ребра $e = \{u; v\}$ в списку суміжності $A[u]$ вершин суміжних з u.

Серед переваг представлення списками сусідів треба виділити більш компактну репрезентацію, що доречна у випадку розглядання розріджених графів – таких, для яких $|E|$ набагато менша за $|V|^2$[1].

Для представлення у вигляді матриці суміжності, ми нумеруємо вершини з V натуральними числами, після чого розглядаємо матрицю $A = (a_{ij})$ розміру $|V| \times |V|$, для якої[1]:

$$a_{ij} = \begin{cases} 1, \text{якщо} \{i; j\} \in E; \\ 0 \text{віншомувипадку}. \end{cases}$$

Об'єм необхідної пам'яті для такої репрезентації складає $O(|V|^2)$.

Для зважених графів елемент a_{ij} матриці суміжності повинен містити $\psi(e)$ – вагу ребра $e = \{i; j\}$.

Порівняльна характеристика обох репрезентацій надана в наступній таблиці:

Таблиця 1.1

Порівняльна характеристика способів представлення графів в пам’яті

Операція	Списки сусідів	Матриця суміжності
Зберігання графу (обсяг пам’яті)	O	$O(\|V\|^2)$
Додавання вершини	$O(1)$	$O(\|V\|^2)$
Додавання ребра	$O(1)$	$O(1)$
Видалення вершини	O	$O(\|V\|^2)$
Видалення ребра	O	$O(1)$
Визначення, чи є довільні вершини u і v суміжними	O	$O(1)$

1.5 Граф як абстрактний тип даних

Абстрактний тип даних – це математична модель типів даних, що визначається [4]:

- заданням множини значень певного типу даних; множину значень типу даних також називають його доменом;
- заданням множини функцій або операцій для маніпуляцій над зазначеним типом даних; множина таких функцій називається інтерфейсом типу даних;
- заданням обмежень на визначені функції.

За множину значень, що належать до абстрактного типу даних графа візьмемо одну з визначених вище репрезентацій графу в пам’яті комп’ютера. Для реалізації було обрано представлення на основі списку сусідів, як більш універсальну і компактну.

Серед операцій, необхідних при розгляді графа в рамках даної роботи виділимо наступні [3]:

- $adjacent(G, x, y)$(де G – граф, а xіy – вершини) - визначає, чи існує вершина $e = \{x; y\}$;
- $neighbors(G, x)$ – повертає список усіх вершин, що є сусідами вершини x;
- $getVertexes(G)$ – повертає список усіх вершин графу;
- $getEdges(G)$ – повертає список усіх ребер графу;
- $getEdgeWeight(G, u, v)$ – повертає вагу ребра (u, v);
- $addVertex(G, x)$ – додає вершину x до графу G;
- $removeVertex(G, x)$ – видаляє вершину x з графу G;
- $addEdge(G, x, y, w)$ – додає ребро $e = (x, y)$ до графу G, де w - $\psi(e)$ ребра $e = (x, y)$.
- $removeEdge(G, x, y)$ – видаляє ребро $e = \{x; y\}$ з графу G.

Багато алгоритмів потребують зберігання з вершиною додаткових даних: міток, ключів тощо, тож як опціональну операцію до абстрактного типу даних графа додамо наступні операції:

- $setVertexData(G, x, v)$ – встановлює значення даних, асоційованих з вершиною x в v;
- $getVertexData(G, x, v)$ – повертає значення даних, асоційованих з вершиною x.

РОЗДІЛ 2

АЛГОРИТМИ ПОШУКУ НАЙКОРОТШИХ ШЛЯХІВ В ГРАФІ

Розроблено безліч варіацій алгоритмів розв'язання задачі побудови найкоротшого шляху між вершинами в графі, деякі з яких можуть бути більш чи менш ефективними за певних обмежень на структуру графа: вони відрізняються складністю обчислень та використаними структурами даних, але в рамках даної роботи було вирішено розглянути наступні алгоритми, як найбільш універсальні:

- алгоритм Дейкстри;
- алгоритм Беллмана-Форда;
- алгоритм Флойда-Воршелла;
- алгоритм Левіта.

Приклади реалізацій наведено мовою JavaScript. Причиною вибору цієї мови стала сукупність наступних факторів:

- це найпопулярніша у світі мова програмування за даними сайту Stack Overflow [5];
- функціональна спрямованість;
- простота вивчення;
- найрозвиненіша інфраструктура: найбільш широкий обсяг інструментів розробника, бібліотек, фреймворків тощо.

2.1 Алгоритм Дейкстри

Алгоритм Дейкстри – це алгоритм пошуку дерева найменших шляхів з коренем в заданій вершині. Вперше опублікований у 1959 році Едсгером Дейкстрою [1].

Існує безліч варіацій даного алгоритму: опублікована Дейкстрою шукала найкоротший шлях між двома заданими вершинами графа, але

більш розповсюджена потребує закріплення за певною вершиною графа статусу кореня і будує найкоротші відстані від неї до всіх інших вершин графа.

Приклад реалізації даного алгоритму потребує визначення абстрактного типу даних черги з пріоритетом. Надалі в рамках даної роботи за визначення черги і черги з пріоритетом візьмемо означення дані в джерелі [3].

Нехай s – початкова або коренева вершина графу $G = (V, E, \psi)$, хід алгоритму Дейкстри складається з наступних кроків:

1. створимо масив d, в елементі $d[v]$ якого будемо зберігати довжину поточного найкоротшого шляху від s до d: $d[s]$ буде становити 0, а для всіх інших вершин – нескінченність;

2. створимо чергу з пріоритетом pq в яку покладемо усі вершини, величина пріоритету для вершини v буде становити $d[v]$;

3. створимо, нарешті, масив p, для якого елемент $p[v]$ являє собою вершину, попередню до даної в найкоротшому шляху від s; початкове значення для всіх вершин становить null.

4. за допомогою $pq.extract(\ )$ отримаємо найближчу до s невідвідану вершину; запишемо її в c.

5. для кожної вершини v у $G.neighbors(c)$ встановимо $d[v]$ рівним min;

6. якщо для вершини v $d[c] + \psi((c, v))$ виявилося меншим за $d[v]$ встановимо $p[v]$ в c;

7. якщо pq ще містить вершини, повернемося до кроку (4).

За отриманим масивом p можна відновити шляхи до кожної вершини v, шукаючи $p[v]$, $доки не потрапимо в s$. Зазначимо також, що у випадку, коли $p[v]$ не містить значення, шляху між sі v не існує у графі G.

Далі наведено приклад реалізації алгоритму Дейкстри:

```
function dijkstra(g, s) {
    let d = G.getNodes().map(() => Number.POSITIVE_INFINITY);
    d[s] = 0;

    let pq = new PriorityQueue();
    G.getNodes().forEach(n => pq.insert(n, d[n]));

    let p = G.getNodes().map(() => null);

    while (!pq.isEmpty()) {
        let c = pq.extract();

        for (const n of G.neighbours(c)) {
            let w = G.getEdgeWeight(c, n);

            if (d[c] + w < d[n]) {
                d[n] = d[c] + w;
                pq.changePriority(n, d[c] + w);
                p[n] = c;
            }
        }
    }

    return {
        distances: d,
        parents: p
    };
}
```

Рис. 2.1 Вихідний код реалізації алгоритму Дейкстри.

Далі наведено приклад реалізації алгоритму відновлення шляхів з початкової вершини s до вершини $target$ за допомогою отриманого в ході виконання алгоритму Дейкстри масиву p:

```
function restorePath(parents, target) {
    const reversedPath = [];

    let c = parents[target];

    while (c != null) {
        reversedPath.push(c);
        c = parents[c];
    }

    return
}
```

Рис. 2.2 Вихідний код реалізації алгоритму відновлення шляхів.

Зазначимо, що обчислювальна складність наведеного рішення для орієнтованого зваженого графу $G = (V, E, \psi)$ становить $O\big((|V| + |E|) log|V|\big)$[1,2,6], однак алгоритм може бути легко переписаний з використанням масиву замість черги з пріоритетом: в такому разі обчислювальна складність буде становити $O(|V|^2)$[6].

2.2 Алгоритм Беллмана-Форда

Алгоритм Беллмана-Форда – алгоритм, вперше запропонований Альфосо Шимбелом у 1955 році, але названий на честь Річарда Беллмана і Лестера Форда, що опублікували його у 1958 і 1956 роках відповідно [1, 2]. Як і алгоритм Дейкстри, дозволяє побудувати дерево найкоротших шляхів з деякої фіксованої вершини s графа $G = (V, E, \psi)$ до усіх інших вершин.

Використовуючи принцип динамічного програмування, розрахуємо довжини найкоротших шляхів. Нехай s – початкова вершина, а $d[k][v]$ – довжина найкоротшого шляху від s до v, що містить k ребер, тоді [1]:

$$d[k][v] = \min_{u \in G.neighbours(v)} \left(d[k-1][u] + \psi(u, v) \right)$$

Цим рекурентним співвідношенням можна скористатись для створення алгоритма динамічного програмування. Далі наведемо вихідний код рішення:

Відновлення найкоротших шляхів можна реалізувати наступним чином:

- створимо масив p, елемент $p[v]$ якого є попереднім до v на найкоротшому шляху від s до v;
- під час обчислення $d[k][v]$, відстежимо, для якого саме u (сусіда вершини v) величина $d[k-1][u] + \psi(u, v)$ буде найменшою; встановимо $p[v] \coloneqq u$.

Використовуючи масив попередніх елементів p, можна відновити найкоротші шляхи способом, описаним вище для алгоритму Дейкстри.

Далі наведена повна реалізація алгоритму Беллмана форда засобами мови JavaScript:

```
function bellmanFord(G, s) {
    let d = G.getNodes().map(
        () => G.getNodes().map(
            () => Number.POSITIVE_INFINITY));
    d[s][0] = 0;

    let p = G.getNodes().map(() => null);

    for (let i = 1; i < nodes.length; i++) {
        for (const edge of graph.getEdges()) {
            const { source, target, weight } = edge;

            if (d[source][i - 1] + weight < d[target][i]) {
                d[target][i] = d[source][i - 1] + weight;
                p[target] = source;
            }
        }
    }

    return {
        distances: d.map(d => d[G.getNodes()[G.getNodes().length - 1]]),
        parents: p
    };
}
```

Рис. 2.3 Вихідний код реалізації алгоритму Беллмана-Форда.

Обчислювальна складність даного алгоритму складає $O\left((|V||E|)\right)$, а об’єм необхідної пам’яті становить $O\left((|V|^2)\right)$[1, 2, 7].

2.3 Алгоритм Флойда-Воршелла

Алгоритм Флойда-Воршелла – алгоритм пошуку найкоротших шляхів між усіма парами вершин в орієнтованому зваженому графі, названій на честь Роберта Флойда і Стівена Воршелла – досліджувачів, що незалежно один від одного опублікували цей алгоритм у 1962 році [1, 2].

На відміну від алгоритму Дейкстри і алгоритму Беллмана-Форда, алгоритм Флойда лише розраховує довжини найкоротших шляхів між вершинами: відновлення шляхів можна виконати за допомогою відомої модифікації алгоритму.

Як і попередній алгоритм, алгоритм Флойда-Воршелла заснований на принципі динамічного програмування і використовує рекурентні співвідношення спеціальної структури.

Нехай дано орієнтований зважений граф $G = (V, E, \psi)$, аd_{vu}^{k} – довжина найкоротшого шляху з вершини vдовершиниu, що проходить лише через вершини $v_1 \ldots v_k$[1, 2, 8].

$$d_{vu}^{k} = min\left(d_{uv}^{k-1}, d\middle|vi^{k-1} + d_{iu}^{k-1}\right) \quad (2.1)$$

Користуючись результатом, наведеним в [8], наведемо Приклад реалізації алгоритму Флойда-Воршелла:

```
function floydWarshall(G) {
    const nodes = G.getNodes();

    const d = nodes.map(() => nodes.map(() => Number.POSITIVE_INFINITY));
    const next = nodes.map(() => nodes.map(() => null));

    for (const edge of G.getEdges()) {
        const { source, target, weight } = edge;

        d[source][target] = weight;
        next[source][target] = target;
    }
    for (const v of G.getNodes()) {
        d[v][v] = 0;
        next[v][v] = v;
    }

    for (let k = 0; k < nodes.length; k++) {
        for (let i = 0; i < nodes.length; i++) {
            for (let j = 0; j < nodes.length; j++) {
                if (d[i][j] > d[i][k] + d[k][j]) {
                    d[i][j] = d[i][k] + d[k][j];
                    next[i][j] = next[i][k];
                }
            }
        }
    }

    return {
        distances: d,
        next
    }
}
```

Рис. 2.4. Вихідний код реалізації алгоритму Флойда-Воршелла.

Модифікація алгоритму задля додання можливості відновлення найкоротших шляхів потребує створення додаткового масиву next, елемент $next[u][v]$ містить номер вершини i з формули 2.1 – точки об єднання шляхів меншої довжини [8].

Наведемо приклад реалізації алгоритму відновлення шляху між вершинами $source$ та $target$ для алгоритму Флойда-Воршелла засобами мови JavaScript:

```
function restorePath(next, source, target) {
    if (next[source][target] == null) {
        return [];
    }

    path = [source];
    while (source != target) {
        source = next[source, target];
        path.push(source);
    }
    return path;
}
```

Рис 2.5 Вихідний код реалізації алгоритму відновлення шляхів.

2.4 Алгоритм Левіта

Алгоритм Левіта, як і алгоритм Дейкстри використовується для побудови дерева найкоротших шляхів до усіх вершин графа з деякої фіксованої вершини s.

Приклад реалізації алгоритма Левіта потребує визначення абстрактного типу даних множини, отже тепер і надалі в рамках даної роботи будемо використовувати визначення АДТ множини, надане в роботі [3].

Для наведення неформального опису алгоритму, задамо граф $G = (V, E, \psi)$ і вершину s, що уде слугувати початковою, або кореневою.

Ініціалізація алгоритму Левіта полягає у [9]:

- створенні трьох множин $S1, S2$і$S3$, що повинні виконувати наступні функції:
  1. S1 – множина вершин, відстань до яких вже обчислено (але, можливо, остаточно);
  2. S2 – черга з вершин, відстань до яких обчислюється, на початку містить s;
  3. S3 – множина вершин, відстань яких ще не обчислено;
- створення масиву d, в елементі $d[v]$ якого будемо зберігати довжину поточного найкоротшого шляху від s до d: $d[s]$ буде становити 0, а для всіх інших вершин – нескінченність;
- створення масиву p, для якого елемент $p[v]$ являє собою вершину, попередню до даної в найкоротшому шляху від s; початкове значення для всіх вершин становить null;

Хід алгоритму Левіта буде складатися з наступних кроків [9]:

1. за допомогою $S2.extract(\quad)$ отримаємо поточну вершину c.
2. для кожної вершини v у $G.neighbors(c)$:

- якщо v належить$S3$ – додаємо v в кінець $S2$ і видаляємо з $S3$; встановлюємо $d[v]$ рівним min; якщо для вершини v $d[c] + \psi((c, v))$ виявилося меншим за $d[v]$ встановимо $p[v]$ в c;
- інакше, якщо v належить$S2$, встановлюємо $d[v]$ рівним min; якщо для вершини v $d[c] + \psi((c, v))$ виявилося меншим за $d[v]$ встановимо $p[v]$ в c;
- інакше, якщо v належить$S1$ додаємо v на початок $S2$ і видаляємо з $S3$; встановлюємо $d[v]$ рівним $d[c] + \psi((c, v))$; встановлюємо $p[v]$ в c;

3. у разі, якщо $S2$ ще містить елементи, повертаємося до кроку 1.

Відновлення шляхів виконується аналогічно до алгоритму Дейкстри, наведеного вище.

Далі наведений приклад реалізації алгоритму Левіта засобами мови JavaScript:

Обчислювальна складність наведеного алгоритму для орієнтованого зваженого графу $G = (V, E, \psi)$ становить O[9].

РОЗДІЛ 3

РОЗРОБКА ПРОГРАМНОГО ЗАСОБУ ВІЗУАЛІЗАЦІЇ АЛГОРИТМІВ ПОШУКУ НАЙКОРОТШОГО ШЛЯХУ

3.1 Опис вимог

В межах даної роботи поставлено мету розробити бібліотеку програмних компонентів для редагування, візуалізації і аналізу графів і візуалізації алгоритмів на графах.

Бібліотека повинна надавати наступні можливості:

1. здійснення основних операцій над графами, наведених у визначенні абстрактного типу даних графу як програмно, так і за допомогою графічного інтерфейсу користувача;
2. відображення візуального представлення графу і здійснення маніпуляцій над ним;
3. візуалізація основних алгоритмів пошуку найкоротшого шляху у графі;
4. розширення бібліотеки новими алгоритмами за допомогою реалізації певного протоколу.

Також необхідно реалізувати програмний засіб редагування графів та візуалізації алгоритмів пошуку найкоротшого шляху у графі.

3.2 Архітектура додатку

Серед дизайнерських рішень найвищого рівня, що були прийняті задля досягнення поставленої мети можна виділити перш за все рішення розроблювати програмний комплекс візуалізації алгоритмів а графах як односторінковий веб додаток, або SPA (Single Page Application).

Особливість архітектури SPA полягає в тому, що всі елементи, необхідні для роботи ПЗ: скрипти, стилі та ін. знаходяться одній

сторінці. Вони завантажуються при ініціалізації [10]. Також даний вид додатків завантажує додаткові модулі після запиту від користувача.

Обробка даних в додатку SPA відбувається на стороні сервера: за допомогою сукупності технологій AJAX усі дані, необхідні для функціонування програми після ініціалізації отримуються додатком асинхронно без перезавантаження сторінки [12].

При завантаженні нових модулів в SPA контент на них оновлюється тільки частково, так як елементам, що не потребує зміні, немає необхідності завантажуватися повторно, сповільнюючи тим самим швидкість відповіді і переданий обсяг даних між браузером і сервером.

Даний вид доданків за способом взаємодії з користувачем найбільш схожий на роботу десктопних додатків. Але, як ми вже говорили, різниця в тому, що пристрій користувача: ПК, планшет або навіть телефон, є тільки засобом введення і виведення інформації, а саме додаток розташований на сервері і використовує його обчислювальні потужності [12].

Серед переваг односторінкових доданків виділимо [11]:

- доступність: можна отримати моментальний доступ до функціонала з будь-якого типу пристрою без проблем з сумісністю, достатнім обсягом пам'яті, необхідними обчислювальними потужностями або з витратою часу на установку;
- можливість задіяти великі обсяги даних: розмір програми і використовуються ним даних не обмежений пам'яттю пристрою;
- швидкість. Одна сторінка, що містить весь необхідний інтерфейс не тільки економить час на повторну завантаження даних, а й підвищує продуктивність роботи з веб-додатком;
- чимало можливостей розробки: розробникам доступні фреймворки, які спрощують створення архітектури проекту і надають чимало готових елементів для роботи.

Недоліки [11]:

- труднощі з SEO: особливості SPA ускладнюють або унеможливлюють процес індексації пошуковими системами всіх модулів програми. Це може викликати труднощі з оптимізацією;
- SPA не працюють у користувачів з вимкненою підтримкою JS: деякі користувачі вимикають підтримку JS-елементів у себе в браузерах, через що Single Page Application, що використовує їх у роботі, не функціонує.

Т.я. розроблюваний додаток не є інформаційним ресурсом та не містить великої кількості тексту, що потребує індексації пошуковими сервісами, недоліки такої архітектури не є для нас критичними.

3.3 Короткий огляд використаних технологій

В рамках виконання даної роботи буди використані наступні технології, фреймворки і бібліотеки:

1. мова програмування JavaScript версії ES8;
2. JavaScript-фреймворк React [13];
3. бібліотека для програмної навігації React Router;
4. JavaScript-бібліотека для візуалізації графів G6[15];
5. бібліотека для візуалізації графів засобами фреймворку React Graphin [14] та інші.

Далі надамо більш детальний опис технологіям, що заслуговують окремої уваги.

Згідно з офіційною документацією ReactJS, React – це бібліотека для створення компонованих інтерфейсів користувача, які представляють дані, що змінюються з часом. Серед переваг React виділимо наступні [13]:

1. React заохочує до використання компонентної архітектури в проектуванні інтерфейсів користувача, тобто фокусується на розбитті графічного інтерфейсу на окремі функціональні або логічні компоненти, що мають чітко визначений набір методів, подій і властивостей;

2. React використовує для опису компонентів інтерфейсу JSX замість окремих мов шаблонізації; JSX – це надмножина ECMAScript, що дозволяє використання коду, подібного до XML/HTML в програмах, написаних мовою JavaScript;

3. React реалізує однонапрямлений потік даних, за якого існує лише один шлях, яким дані програми можуть передаватися між компонентами – від батьківського компоненту до дочірнього; використання цієї техніки програмування полегшує розуміння програми, т.я. кількість позицій в коді, в яких дані програми можуть бути змінені значно обмежено.

G6 – це JavaScript-бібліотека для створення інструментів візуалізації, аналізу і моделювання графів. Серед переваг використання даної бібліотеки виділимо наступні [15]:

1. надає реалізацію структури даних графу, що відповідає наведеному вище визначенню графу як абстрактного типу даних;

2. надає високопродуктивний двигун візуалізації графів;

3. надає широкі можливості розширення бібліотеки власним кодом: дозволяється як кастомізація зовнішнього вигляду елементів графу, так і їх поведінки: з використанням вбудованого механізму реєстрації поведінок (Behaviors).

Graphin – це бібліотека, що надає набір компонентів для використання бібліотеки G6 в React-додатках [14]. На додаток Graphin розширює визначену в G6 концепцію поведінки: поведінки в Graphin – це спеціальні React-компоненти, що найчастіше не відмальовують ніяких JSX-елементів, а лише додають обробники подій графу, використовуючи вбудовані можливості фреймворку React.

3.4 Бібліотека компонентів

В ході роботи над програмним засобом візуалізації було прийнято рішення розбити додаток на 2 модулі – редактор графів і відображувач алгоритмів, які будуть реалізовані у вигляді окремих React-компонентів. Ці два модулі поєднуються в додаток за допомогою компонента верхнього рівня App, що реалізує програмну навігацію між редактором і відображувачем, а також відтворює панель з посиланнями на відображувач і редактор.

Програмну навігацію створено за допомогою бібліотеки React Router.

На малюнку наведена схема компонента App:

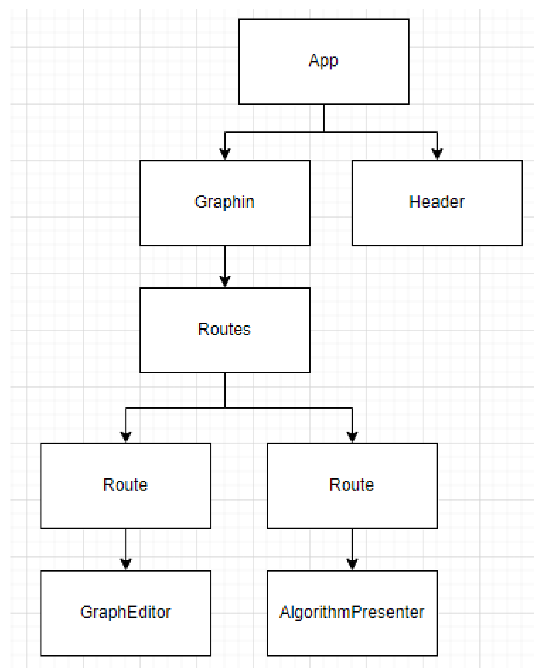

Рис. 3.1 Схема компонента App

Зазначимо, що компоненти GraphEditor і AlgorithmPresenter обгорнуті в єдиний компонент Graphin, тобто відповідні операції

виконуються на єдиному об’єкті G6-графу, а GraphEditor і AlgorithmPresenter є лише наборами Graphin-поведінок.

При проектуванні компонентів додатку широко використовується патерн проектування «State Machine», що для певного програмного компоненту вимагає явного зазначення множини станів, в яких той може перебувати, поведінки компонента у кожному станів і множини можливих переходів між станами.

3.5 Редактор графів

Функції редактору графів реалізовано за допомогою компоненту GraphEditor. Схема розроблюваного компонента представлена на малюнку:

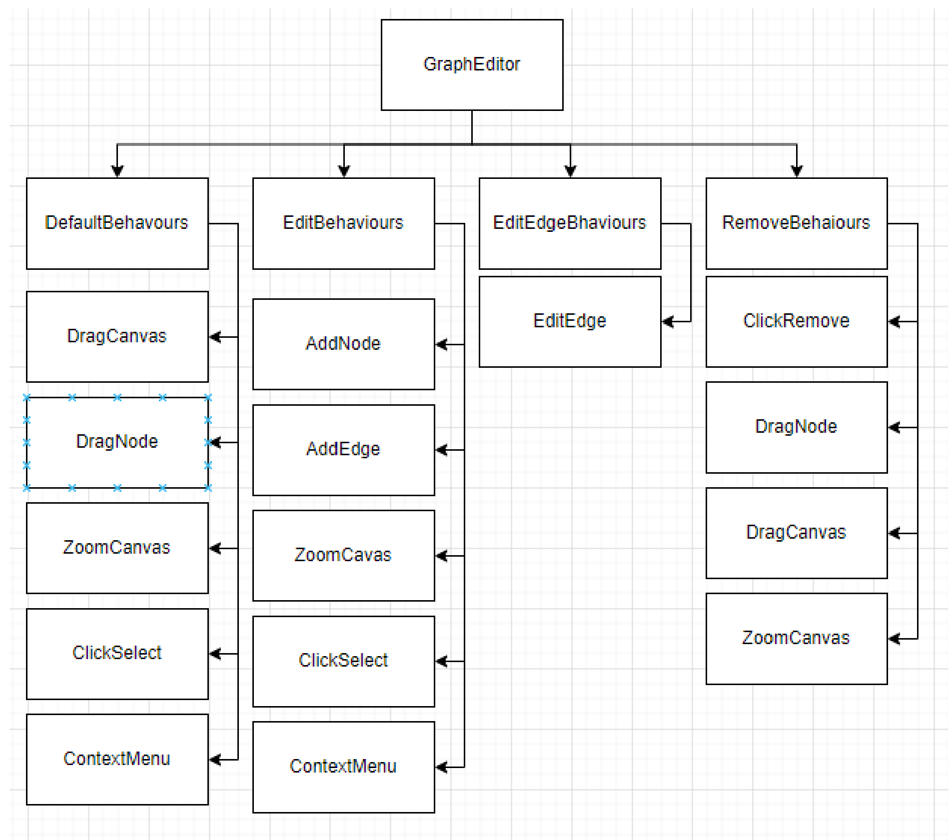

Рис. 3.2 Схема компонента GraphEditor.

Для компоненту GraphEditor визначено наступну множину станів:

- Default – стан редактора за замовчуванням: містить поведінки, що відповідні за навігацію по зображенню графа за допомогою миші, збільшення та зменшення зображення графу, візуальне виділення елемента графу при натисканні на нього лівою миші, а також компонент ContextMenu, що відповідальний за відмалювання контекстного меню з опціями редагування і видалення елемента графу при натисканні на нього правою кнопкою миші;
- Edit – стан редактора, що використовується для редагування графу: містить поведінки для додавання вершин і ребер графу за допомогою кліку на порожнє місце на зображенні графу і виборі двох вершин графу відповідно;
- EditEdge – стан редактора, що відповідальний за редагування певного ребра графу: відмальовує форму для редагування ваги ребра;
- Remove – стан редактора, який використовується для видалення елементів графу: містить поведінки, що відповідальні за видалення ребра або вершини за допомогою натискання лівої кнопки миші.

На наступному малюнку зображена діаграма станів для компонента GraphEditor:

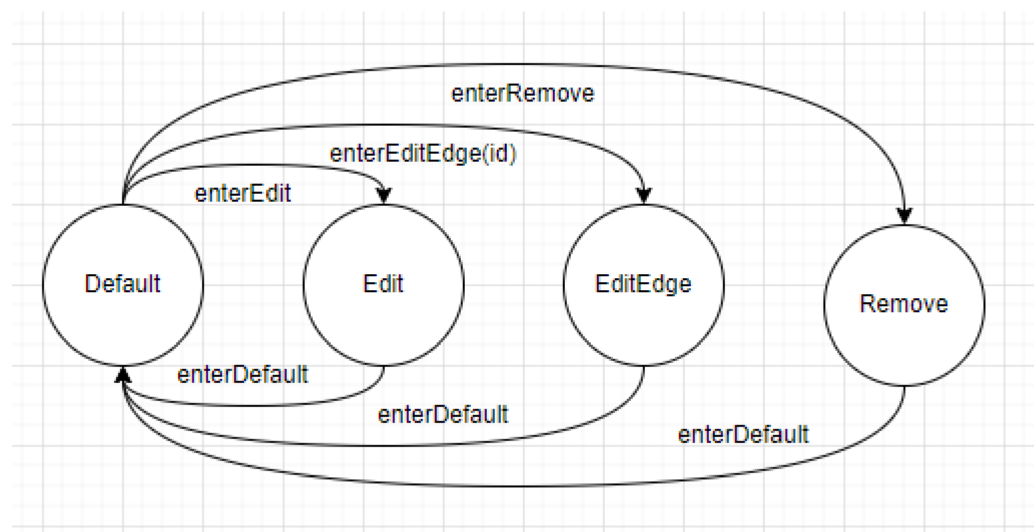

Рис 3.3 Діаграма станів для компонента GraphEditor

Зазначимо також, що, активуючи перехід enterEditEdge, компонент зобов’язаний передати ідентифікатор ребра, що потребує редагування.

Приклад роботи компонента GraphEditor зображено на рисунку.

3.6 Відображувач алгоритмів

Функції візуалізації алгоритмів реалізовано за допомогою компонента AlgorithmPresenter. Схема розроблюваного компонента представлена на рисунку:

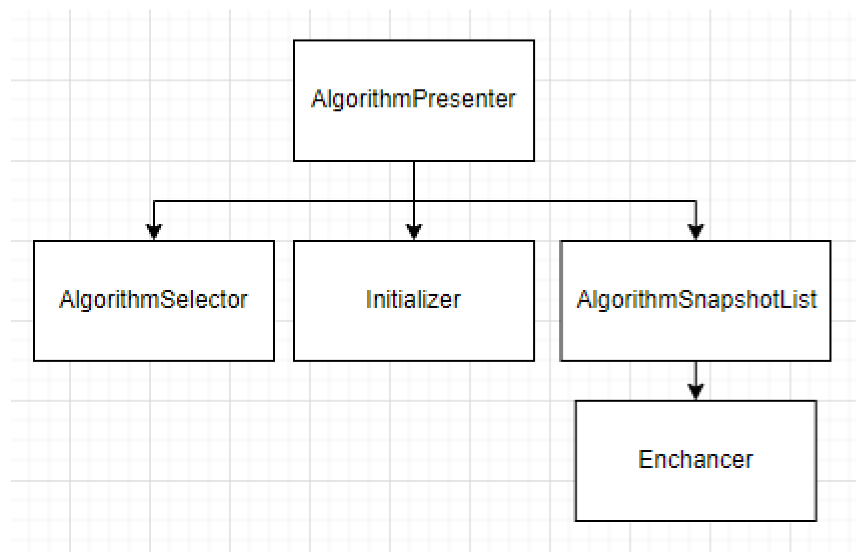

Рис. 3.4 Схема компонента AlgorithmPresenter

Відображувач алгоритмів завжди проходить наступною послідовністю станів:

- Selecting – стан, що використовується для вибору алгоритму, що потребує візуалізації; у цьому стані компонент AlgorithmPresenter відмальовує список реалізованих алгоритмів і пропонує користувачу вибрати один з них натисканням лівої кнопки миші; за умови вибору алгоритму, компонент активує перехід onSelected, в який передає відповідний екземпляр ініціалізатору;
- Initializing – стан компонента, що реалізує функції ініціалізації алгоритму: наприклад, алгоритм Дейкстри потребує

вказання початкової вершини, з якої буде здійснюватися пошук найкоротших шляхів; знаходячись у цьому стані, AlgorithmPresenter відмальовує здобутий на попередньому етапі ініціалізатор; за умови успішної інінціалізації, компонент активує перехід onInitialized, передаючи в нього екземпляр алгоритму – функції, що на вхід приймає екземпляр графу і повертає множину знімків стану графу, що відповідають станам виконання алгоритму, а також компонент Enchancer, що модифікує візуальне відображення стану графу відповідно до алгоритму: наприклад, реалізований SingleSourceEnchancer, що використовується алгоритмами пошуку найкоротших шляхів з однієї вершини до всіх інших, візуально виділяє всі вершини, що на заданому етапі належать до побудованого дерева найкоротших шляхів;

- Presenting – стан компонента, в якому здійснюється відображення знімків стану графу протягом виконання алгоритму; в цьому стані AlgorithmPresenter відмальовує набір кнопок для переходу до наступного, попереднього, першого і останнього знімків;

Діаграма станів компонента AlgorithmPresenter наведена на наступному малюнку:

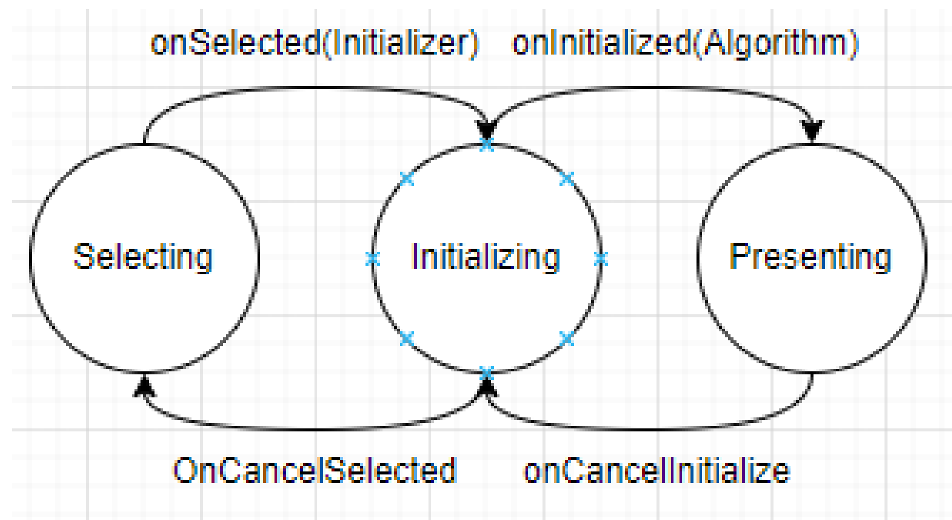

Рис. 3.5 Діаграма станів компонента AlgorithmPresenter.

Таким чином розширити розроблений програмний продукт новими алгоритмами можна за виконання наступної послідовності дій:

- необхідно розробити ініціалізатор – компонент, що довільним чином ініціалізує алгоритм і активує перехід onInitialized, передавши екземпляр алгоритму;
- необхідно додати алгоритм до списку запропонованих компонентом AlgorithmSelector, який відмальовується у стані Selecting і організувати активацію переходу onSelected з екземпляром відповідного ініціалізатору;
- необхідно розробити алгоритм – функцію, що приймає на вхід екземпляр графу і повертає список знімків стану графу, що відповідають етапам виконання алгоритму та компонент Enchancer;
- необхідно розробити Enchancer, що буде візуально виділяти подробиці стану виконання алгоритму, або скористатися одним з запропонованих найбільш універсальних реалізацій: SingleSourceEnchancer або EachPairEnchancer.

Приклад роботи компонента AlgorithmPresenter при візуалізації алгоритму Дейкстри представлено на рисунку:

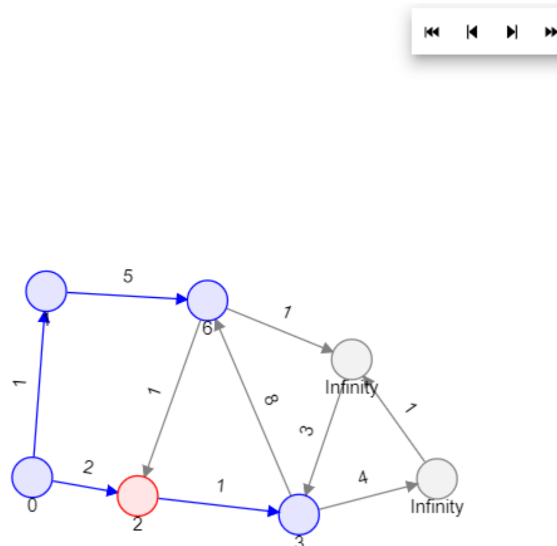

Рис. 3.6 Приклад роботи компонента AlgorithmPresenter при візуалізації алгоритму Дейкстри.

Приклад роботи компонента AlgorithmPresenter при візуалізації алгоритму Беллмана-Форда представлено на рисунку:

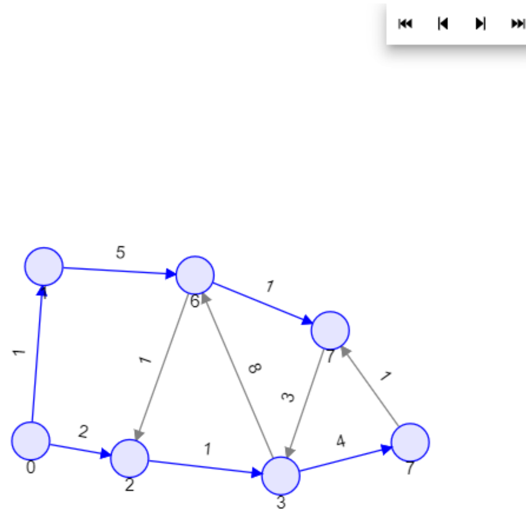

Рис. 3.7 Приклад роботи компонента AlgorithmPresenter при візуалізації алгоритму Беллмана-Форда.

ВИСНОВКИ

В ході роботи поставлена мета розробки програмного засобу для візуалізації алгоритмів для вирішення задачі пошуку найкоротшого шляху в графі.

В процесі було виконано наступні завдання;

- досліджено існуючі інструментів і бібліотек для створення інструментів візуалізації і аналізу графів;
- розроблено бібліотеку компонентів, що реалізує функції візуалізації, редагування і аналізу графів, а також функції візуалізації алгоритмів для вирішення задачі пошуку найкоротшого шляху в графі;
- на основі розробленої бібліотеки створено додаток, що надає функції редагування, візуалізації і аналізу графів, а також візуалізації алгоритмів пошуку найкоротшого шляху в графі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Кормен, Томас Х.; Лейзерсон, Чарльз Е.; Рівест, Рональд Л.; Стайн, Кліффорд. Introduction to Algorithms, - MIT Press, 2001. – 1312 с.

2. Такер, Аллен Б. Computer Science Handbook. Second Edition, - Chapman & Hall/CRC, 2004. – 2624 с.

3. Міллер, Бредлі Н.; Ранум, Девід Л. Problem Solving with Algorithms and Data Structures Using Python. Second Edition, - Franklin, Beedle & Associates, 2001. – 438 с.

4. Abstract data type: [он-лайн ресурс]. URL: https://en.wikipedia.org/wiki/Abstract_data_type(дата останнього звернення: 23.03.2022).

5. Stack Overflow Developer Survey 2021: [он-лайн ресурс]. URL: https://insights.stackoverflow.com/survey/2021(дата останнього звернення: 23.03.2022).

6. Нахождение кратчайших путей от заданной вершины до всех остальных вершин алгоритмом Дейкстры: [он-лайн ресурс]. URL: https://e-maxx.ru/algo/dijkstra(дата останнього звернення: 23.03.2022).

7. Алгоритм Форда-Беллмана: [он-лайн ресурс]. URL: https://e-maxx.ru/algo/ford_bellman(дата останнього звернення: 23.03.2022).

8. Алгоритм Левита нахождения кратчайших путей от заданной вершины до всех остальных вершин: [он-лайн ресурс]. URL: https://e-maxx.ru/algo/levit_algorithm(дата останнього звернення: 23.03.2022).

9. Алгоритм Флойда-Уоршелла нахождения кратчайших путей между всеми парами вершин: [он-лайн ресурс]. URL: https://e-maxx.ru/algo/floyd_warshall_algorithm(дата останнього звернення: 23.03.2022).

10. Single-page application: [он-лайн ресурс]. URL: https://en.wikipedia.org/wiki/Single-page_application(дата останнього звернення: 23.03.2022).

11. Что такое SPA-приложения: [он-лайн ресурс]. URL: https://wezom.com.ua/blog/chto-takoe-spa-prilozheniya(дата останнього звернення: 23.03.2022).

12. Scott E. SPA Design and Architectue, – Manning, 2015. – 314 с.

13. React - A JavaScript library for building user interfaces: [он-лайн ресурс]. URL: https://reactjs.org/(дата останнього звернення: 23.03.2022).

14. Graphin 图的分析洞察: [он-лайн ресурс]. URL: https://graphin.antv.vision/ (дата останнього звернення: 23.03.2022).

15. G6 图可视化引擎: [он-лайн ресурс]. URL: https://g6.antv.vision/ (дата останнього звернення: 23.03.2022).