КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет комп'ютерних наук та кібернетики

Кафедра інтелектуальних програмних систем

Кваліфікаційна робота

на здобуття ступеня бакалавра

за спеціальністю 121 Програмна інженерія

на тему:

ПОРІВНЯННЯ ТЕХНОЛОГІЧНИХ РІШЕНЬ ДЛЯ ОБЧИСЛЕННЯ В КЛІЄНТ-СЕРВЕРНИХ ЗАСТОСУНКАХ

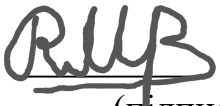
Виконав студент 4-го курсу
Ростислав МОЧУЛЬСЬКИЙ


(Підпис)

Науковий керівник:
Асистент, кандидат фіз.-мат. наук
Костянтин ЖЕРЕБ

_____
(Підпис)

Засвідчую, що в цій курсовій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент 
(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри інтелектуальних програмних систем
«25» травня 2022 р.,
протокол № 10

Завідувач кафедри
О. І. Провотар _____
(підпис)

КИЇВ-2022

РЕФЕРАТ

Обсяг роботи 41 сторінка, 14 ілюстрацій, 4 таблиці, 20 джерел посилань.

АЛГОРИТМ ФОРДА-ФАЛКЕРСОНА, ГРАФ, КЛІЄНТ, СЕРВЕР, ТРАНСПОРТНА МЕРЕЖА, ЧАС ВИКОНАННЯ, JNI, WEBASSEMBLY, .NET, BLAZOR

Об’єктом розробки є клієнт-серверні застосунки з реалізованими алгоритмами на відповідних клієнті та сервері із можливістю вимірювання часу виконання функцій розрахунку алгоритмів.

Мета роботи полягає у розробці клієнт-серверних застосунків та реалізації в них алгоритму за допомогою різних програмних засобів та порівнянні продуктивності різних варіантів реалізації.

Методи та інструменти розробки. Для розробки серверів було використано Java з Spring Framework та C# з .NET Blazor Server. Для розробки клієнту було використано React та фреймворк Blazor відповідно. Для використання реалізації алгоритмів з використанням ефективних мов було використано JNI та WebAssembly.

Новизна роботи полягає у порівнянні популярних та актуальних програмних засобів для розробки клієнт-серверних застосунків, засобів для взаємодії з ефективними мовами програмування.

В результаті було спроектовано та розроблено клієнт-серверні застосунки з реалізованими алгоритмами на клієнтській та серверній частині, реалізовані алгоритми з використанням засобів взаємодії з ефективними мовами та проведено порівняння різних варіантів реалізації.

Практичне значення одержаних результатів. Результати досліджень можуть бути використані для прийняття архітектурних рішень під час розробки нового програмного забезпечення, зокрема розділу функціоналу на серверну та клієнтську частину, та отримання більш чіткого розуміння про приріст ефективності при використанні засобів взаємодії з ефективними мовами.

ЗМІСТ

ВСТУП 4

РОЗДІЛ 1 ОПИС ВИКОРИСТАНИХ ТЕХНОЛОГІЙ 6

1.1 Вибір мови програмування для створення сервера 6

1.2. Spring Framework 8

1.3. React 9

1.4. ASP.NET Core Blazor 10

1.5. Blazor Server 12

1.6. Blazor Webassembly 14

РОЗДІЛ 2 ПРОЕКТУВАННЯ ТА РОЗРОБКА КЛІЄНТ-СЕРВЕРНОГО ЗАСТОСУНКУ 17

2.1. Вибір алгоритму 17

2.2 Алгоритм Едмондса-Карпа 19

2.3. Технологічні варіанти реалізації алгоритму 20

2.4. Сервер на мові Java 21

2.5. Клієнт з використанням React 24

2.6. Blazor WebAssembly застосунок 28

РОЗДІЛ 3 ПОРІВНЯННЯ РЕАЛІЗОВАНИХ АЛГОРИТМІВ НА КЛІЄНТІ ТА НА СЕРВЕРІ 33

3.1. Порівняння часу виконання 33

3.2. Переваги та недоліки технологій реалізації алгоритмів 34

ВИСНОВКИ 38

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 40

ВСТУП

Клієнт-серверна архітектура [1] – один із популярних шаблонів програмного забезпечення з можливістю обміну даних між його компонентами.

Дворівневі клієнт-серверні застосунки передбачають взаємодію двох програмних компонентів — клієнтського та серверного. Клієнт – модуль, який використовується для взаємодії користувача з застосунком, сервер – модуль, який відповідає за управління (збереження та обробку) даних та виконання складних обрахунків. В залежності від того, як розподіляються розрахунки між цими компонентами розрізняють такі моделі для клієнта:

- «тонкий клієнт» — модель при якій всі обрахунки логіки застосунку чи переважна їх частина відбуваються на сервері, а клієнт відповідає тільки за представлення результатів та надає користувачу можливості взаємодії з користувачем;
- «товстий клієнт» — модель при якій обробка інформації та обробка наданих користувачем даних та параметрів відбувається на стороні клієнта, а сервер виконує функції збереження та управління даними.

Звісно, даний розподіл є досить умовним і в основному прості розрахунки можуть виконуватись на клієнті, а більш складні на сервері, проте завжди існують винятки, пов’язані з конкретною задачею. Також при розробці клієнт-серверного застосунку можуть виникати ситуації, коли нам потрібно обирати на якій з компонент будуть реалізовані певні розрахунки.

Актуальність роботи полягає у важливості архітектурних рішень під час розробки нового програмного забезпечення, невеликою кількістю інформації щодо порівнянь різних архітектурних рішень для реалізації алгоритмів у клієнт-серверних застосунках.

Мета й завдання роботи. Метою роботи є розробка клієнт-серверних застосунків та реалізації в них алгоритму за допомогою різних програмних засобів та порівнянні продуктивності різних варіантів реалізації. Для досягнення мети були поставлені та виконані наступні **завдання**:

- створення клієнт серверних застосунків;
- реалізація алгоритмів на сервері з використанням тої мови, яка обрана для відповідного веб сервера;
- реалізація алгоритму на сервері з використанням більш ефективної мови і певних засобів взаємодії з такою мовою;
- реалізація алгоритму на клієнтах з використанням мови, на якій реалізований відповідний;
- реалізація алгоритму на клієнті з використанням більш ефективної мови;
- порівняння продуктивності варіантів реалізації алгоритму, їх переваг та недоліків.

Об’єктом розробки є клієнт-серверні застосунки з реалізованими алгоритмами на відповідних клієнті та сервері із можливістю вимірювання часу виконання функцій розрахунку алгоритмів.

Предметом дослідження є клієнт-серверні застосунки з варіантами реалізації алгоритму на клієнті та на сервері, реалізації алгоритму з використанням більш ефективної мови на клієнті та на сервері.

РОЗДІЛ 1
ОПИС ВИКОРИСТАНИХ ТЕХНОЛОГІЙ

1.1 Вибір мови програмування для створення сервера

Зазвичай клієнт-серверні застосунки – це досить великі за обсягом програмні рішення, які з часом масштабуються, доповнюються функціоналом. Серед варіантів мов для написання серверів для веб додатків слід відзначити Python, Go, C++, Java та C#.

Python в останні роки набув великої популярності (рис 1.1), проте в основному його використовують у стартапах та невеликих чи середніх за обсягом проектах, через його порівняно невисоку продуктивність. Go – не така популярна мова, як Python, продуктивність у Go краща, проте спільнота і кількість бібліотек не такі великі. Основною перевагою Go є направленість на розподілені обчислення, багатопоточність. Як і Python, Go не часто використовується і дійсно великих проектах. C++ має дуже високу продуктивність, проте цю мову важко використовувати в веб проектах.

# Ranking	Programming Language	Percentage (YoY Change)	YoY Trend
1	JavaScript	18.756% (+0.053%)	
2	Python	16.628% (+0.390%)	
3	Java	11.680% (+0.742%)	
4	Go	7.829% (-1.176%)	
5	Ruby	7.588% (+0.776%)	^
6	C++	6.985% (-0.439%)	v
7	TypeScript	6.604% (-0.164%)	
8	PHP	5.081% (-0.046%)	
9	C#	3.614% (-0.221%)	
10	C	3.253% (+0.072%)	

Рис 1.1. Популярність мов відповідно до кількості pull requests на ресурсі GitHub за 1 квартал 2021 року [2]

Слід розуміти, що мови JavaScript, Python, Go, C++ чи навіть Scala, яка не є порівняно настільки популярною можуть використовуватись у великих системах теж. Наприклад, для реалізації певного мікросервісу з метою

оптимізації/використання конкретної бібліотеки чи технології. Проте все ж найпопулярнішими ж у розробці та підтримці чи розширенні дійсно великих програмних продуктів є Java та C#. Порівняння цих мов для розробки веб додатків може слугувати темою для довгих дискусій, причиною чого є схожість цих мов. Загалом основною різницею цих мов у розробці клієнт-серверних застосунків є те, що Java почала раніше використовуватись як мова для бекенду в великих проектах. C# ж розвинулась трохи пізніше, відповідно через це у цієї мови може бути менша кількість бібліотек для певних задач, хоча за останні роки їх база сильно розширилась. І питання кількості бібліотек є досить спекулятивним, адже кількість не завжди означає якість. На обох мовах існують приклади дуже ефективних та зручних бібліотек.

Так як Java набула свою популярність раніше, існує велика кількість проектів, які були розроблені використовуючи її, та потребують додавання нового функціоналу/підтримки. Саме через це, незважаючи на те наскільки зможе покращитись C# з платформою .NET, часові та фінансові витрати на перенесення проектів на іншу технологію є занадто великими.

Проте при створенні нового проекту не завжди слід звертати увагу на вибір технологій у своїх попередніх розробках, завжди варто аналізувати, обираючи ефективні та зручні технології для конкретної задачі та цілі. Зараз є багато сучасних рішень, які надають можливість будувати великі та продуктивні проекти та застосунки, використовуючи хмарні технології. Їх найпопулярнішими прикладами є: Amazon Web Services та Microsoft Azure. Microsoft будує свою екосистему, яка буде зручна для розробки та підтримки, використовуючи від початку і до кінця їх технології – від мови програмування та середовища розробки і до хмарних сервісів для розгортання проектів, збереження даних, обміну даними тощо.

Зважаючи на те, що найпопулярнішою “enterprise” мовою для розробки є Java та стрімкий розвиток платформи .NET для C# та бажання

Microsoft побудувати ідеологічно схожий, проте покращений продукт для розробки – у роботі було обрано їх для реалізації серверної частини.

1.2. Spring Framework

Для розробки веб додатку, написаного на мові Java, будемо використовувати Spring Framework [3]. Цей фреймворк об’єднує в собі декілька модулів, кожен з яких містить в собі багатий функціонал. Основними перевагами цього фреймворку при розробці клієнт-серверних систем є:

- Інверсія управління: конфігурація компонентів додатків і управління життєвим циклом об'єктів в програмному застосунку, здійснюється головним чином через інверсію управління;
- Модель-Вигляд-Контролер (Model-View-Controller) [4]: програмний шаблон, а в даному випадку каркас на основі HTTP сервлету, що забезпечує можливості для створення RESTful веб-додатків і веб-служб;
- Управління та доступ до даних – можливість працювати з SQL та NoSQL базами даних;
- Можливості для написання юніт тестів та інтеграційних тестів.

Звісно, це не всі переваги та можливості Spring Framework. Найважливішим аспектами цього фрейморку, які будуть використовуватись в даній роботі і який використовується у кожному веб-додатку є шаблон MVC. Він дозволяє нам створити на сервері певну модель даних та реалізовувати RESTful сервіси для того щоб ми могли звертатись до даних, викликати методи для розрахунків, додавати нові дані тощо, за допомогою HTTP запитів з клієнту. Для цього на сервері нам потрібно буде створити набір контролерів, що будуть приймати й обробляти запити, посилати відповіді.

```
package com.example.restservice;

import java.util.concurrent.atomic.AtomicLong;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class GreetingController {

        private static final String template = "Hello, %s!";
        private final AtomicLong counter = new AtomicLong();

        @GetMapping("/greeting")
        public Greeting greeting(@RequestParam(value = "name", defaultValue =
"World") String name) {
                return new Greeting(counter.incrementAndGet(),
String.format(template, name));
        }
}
```

Рис 1.2. Приклад REST контролера написаного, використовуючи Spring MVC [5]

1.3. React

Окрім сервера нам потрібно реалізувати і клієнт. Популярним підходом сьогодення для веб-сайтів є SPA (Single-page application) [6]. Основною перевагою односторінкових додатків є те, що весь код: розмітка (HTML), стилі (CSS) та функції (Scripts) завантажуються один раз під час користування додатком, лише дані будуть передаватись при надсиланні та отриманні запитів на сервер. Таким чином досвід користувача значно покращується, адже сторінка не оновлюється і не перенаправляє користувача на інші сторінки, у такий спосіб уникаючи багатьох завантажень та очікувань.

Зважаючи на вище сказане, будемо обирати бібліотеку JavaScript для побудови SPA вебсайтів. Звернувшись до статистики популярності (рис. 1.3), бачимо що на сьогодні React [7] є найпопулярнішою бібліотекою JavaScript для створення односторінкових додатків і його популярність зростає з кожним роком досить стрімко.

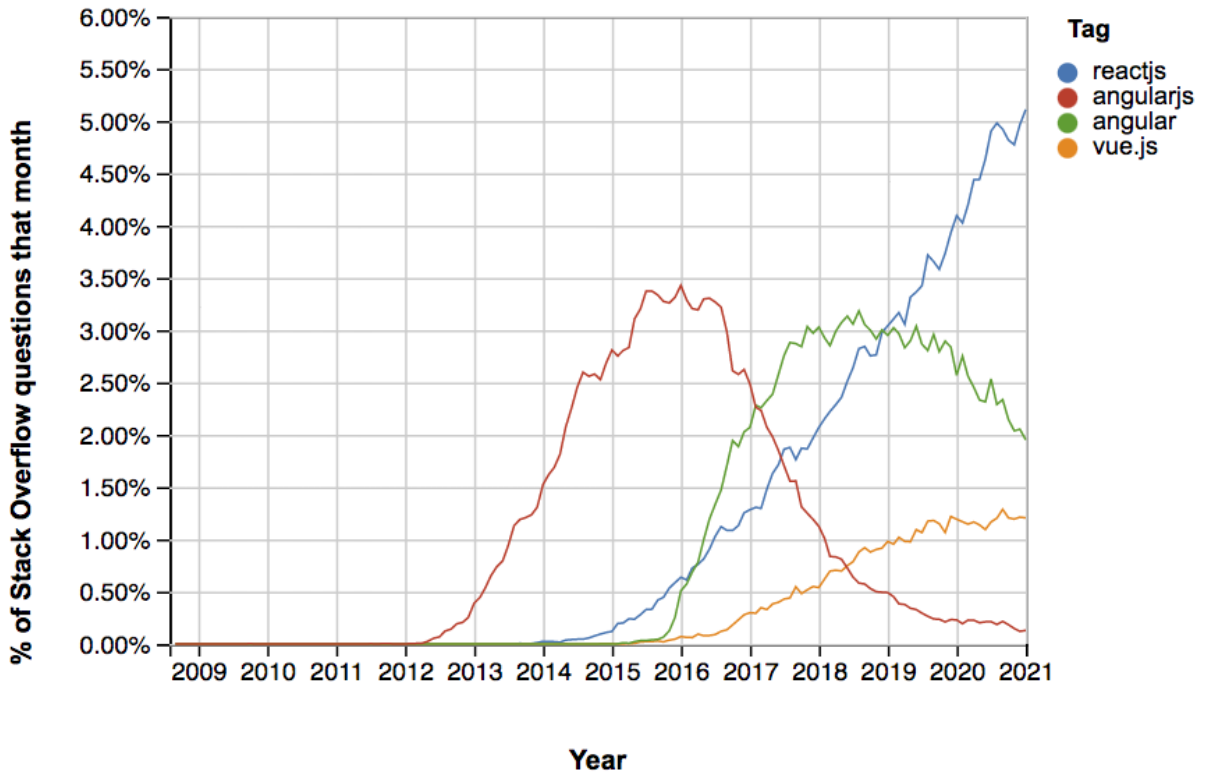


Рис 1.3. Популярність front-end бібліотек на ресурсі Stack Overflow [8]

Серед переваг React слід відмітити:

- Декларативність – розробнику достатньо описати як інтерфейс веб-сайту має виглядати при тих чи інших даних. React візьме на себе їх вчасне оновлення при зміні даних;
- Компоненти – React дозволяє створювати інкапсульовані компоненти, які потім можна об’єднувати в складні користувацькі інтерфейси, передавати в них різні аргументи;
- Відкритий код;
- Велика кількість бібліотек.

1.4. ASP.NET Core Blazor

Для реалізації клієнт-серверного застосунку з сервером, створеним на мові C# було обрано фреймворк ASP.NET Core Blazor [9]. Він є дуже сучасним і новим в порівнянні з іншими фреймворками для частини клієнту, бібліотеками на мові JavaScript. Перший реліз був у 2018 році і зараз він

активно розвивається та підтримується компанією Microsoft. Перевагами фреймворку Blazor є:

- Можливість створення красивих інтерактивних користувацьких інтерфейсів, використовуючи мову C#, а не JavaScript, що зазвичай використовується в таких цілях;
- Можливість створення спільної частини для клієнту та серверу. Це є зручним та дійсно корисним для класів, які використовуються в обох частинах, наприклад класів для запитів та відповідей;
- Рендер користувацьких інтерфейсів як HTML та CSS з підтримкою багатьма браузерами, включаючи мобільні браузери;
- Інтеграція з сучасними платформами розгортки, Docker;
- Можливість розробки гібридних додатків для комп’ютерів та мобільних платформ;
- Легкість у використанні для нових користувачів, адже все побудовано на стандартних мовах розмітки та мові C#, яка має простий та зручний синтаксис;
- Стабільність та різноманітність функціоналу;
- Можливість використання на різних операційних системах – Windows, Linux чи macOS.

Blazor, як і популярні фреймворки, що використовуються для розробки фронтенду, написані на мові JavaScript базується на компонентах. Компоненти являють собою .NET C# класи та використовують в собі звичайну HTML розмітку. Таким чином, комбінуючи використання мови розмітки та привичної мови для розробника, який не має окремого досвіду роботи з фронтенд частиною чи використання JavaScript.

```
<div class="card" style="width:22rem">
    <div class="card-body">
        <h3 class="card-title">@Title</h3>
        <p class="card-text">@ChildContent</p>
        <button @onclick="OnYes">Yes!</button>
    </div>
</div>

@code {
    [Parameter]
    public RenderFragment? ChildContent { get; set; }

    [Parameter]
    public string? Title { get; set; }

    private void OnYes()
    {
        Console.WriteLine("Write to the console in C#! 'Yes' button selected.");
    }
}
```

Рис 1.4 Приклад простого Blazor компоненту [9]

Варто відмітити, що для внутрішніх проектів Blazor є чудовим вибором, адже його легко опанувати та почати створювати інтерфейси, не маючи до цього досвіду роботи з JavaScript та його фреймворками. Для розробника, який раніше використовував мову C# вивчення буде легким та інтуїтивним.

Звичайно, при розробці складних інтерфейсів та візуалізацій можуть виникнути якісь труднощі у порівняння з фреймворками JavaScript, де присутня безліч різних бібліотек та готових рішень, проте технологія Blazor є дійсно новою і стрімко розвивається, у системі керування пакунками NuGet [10] постійно з’являються нові користувацькі бібліотеки.

1.5. Blazor Server

Blazor підтримує три моделі розміщення: Blazor Server, Blazor Webassembly та експериментальний Blazor Hybrid [11]. У моделі розміщення Blazor Server програма запускається на сервері з додатку ASP.NET Core. Оновлення інтерфейсу користувача, обробка подій і виклики JavaScript обробляються через з’єднання SignalR. Стан на сервері, пов'язаний з кожним підключеним клієнтом, називається каналом. Схема

може витримувати тимчасові перебої мережі та повторні спроби клієнта знову підключитися до сервера, коли з'єднання втрачено. Такий механізм роботи має свої особливості. Серед переваг слід відмітити:

- Розмір завантажуваного користувачем місту при відкриванні сторінки є значно меншим, ніж у моделі a Blazor WebAssembly;
- Додаток використовує всі переваги сервера, включаючи використання .NET Core APIs;
- .NET Core на сервері використовується для роботи застосунку, а це означає, що всі інструменти для відлагодки та інші працюють у повному об'ємі;
- Підтримується модель тонкого клієнту, це дає переваги у тому, що додатки, які використовують модель Blazor Server можна використовувати у браузерах, що не підтримують WebAssembly та на пристроях, які суттєво обмежені у ресурсах;
- Весь код додатку, включаючи код компонентів, написаний на .NET/C# не доступний для перегляду користувачем.

Проте, як і у будь-якого рішення у програмуванні, є і свої обмеження та недоліки:

- Зазвичай присутні відчутні затримки, адже кожна взаємодія користувача з інтерфейсом включає в себе передачу даних серверу;
- Немає підтримки роботи під час відсутності з'єднання, як тільки зникає зв'язок додаток перестає працювати;
- Так як основну роботу здійснює сервер, то очевидно не можливе розгортання, використовуючи "Serverless" технології;
- Масштабування додатків, якими будуть користуватись багато користувачів потребує значних ресурсів серверу, щоб він міг справлятись з усіма каналами користувачі та збереженням стану для них.

Якщо труднощі масштабування вирішуються за допомогою використання спеціальних технологій у хмарі, як наприклад Azure SignalR

Service, то інші ж обмеження більше пов’язані з загальною ідеє технології і вирішити їх не так просто. Тому, якщо ці недоліки є значними, варто використовувати модель розміщення Blazor WebAssembly.

1.6. Blazor Webassembly

Blazor WebAssembly є Single-page application (SPA) фреймворком для побудови інтерактивних веб застосунків на клієнтській стороні, використовуючи .NET. Blazor WebAssembly використовує відкриті веб стандарти без плагінів чи перекомпіляції коду у вигляд інших мов програмування. Blazor WebAssembly працює в усіх сучасних веб браузерах, в тому числі у мобільних веб браузерах.

Додатки, що використовують модель Blazor WebAssembly (скорочено WASM) працюють на клієнтській стороні, основуючись на середовищі .NET, яка завантажується у браузер для виконання. Застосунок працює прямо на потоці інтерфейсу у браузері. Оновлення користувацького інтерфейсу і обробка подій відбуваються в рамках одного процесу. Ресурси додатку розгортаються у вигляді статичних файлів на сервері чи на іншій службі, яка може їх доставляти до клієнтських частин користувачів.

Коли застосунок Blazor WebAssembly створений для розгортання без використання бекенд частини ASP.NET Core, яка буде доставляти його файли до клієнтів, тоді він називається автономним Blazor WebAssembly застосунком, в іншому ж випадку при використанні бекенд частини для доставки статичних файлів він називається розміщеним.

Для того, щоб отримати досвід full-stack розробки на .NET слід використовувати розміщений Blazor WebAssembly. Це дає можливість створення спільного коду та використання його на сервері та на клієнті, підтримку пререндеру та використання шаблону MVC. Клієнтська частина розміщеного застосунку має можливість взаємодіяти з серверною через мережу за допомогою різноманітних методів обміну повідомлень та

протоколів, таких як веб API, gRPC-web і SignalR (Use ASP.NET Core SignalR).

Blazor використовує скрипт blazor.webassembly.js, який виконує 2 основні задачі: Завантаження середовища .NET, самого застосунку та його залежностей та ініціалізація середовища для запуску застосунку у браузері.

Загалом модель розміщення Blazor WebAssembly має наступні переваги:

- Завантаження відбувається в один етап, тобто якщо з серверною стороною .NET, яка доставляє статичні файли, втрачається зв'язок застосунок залишається функціональним, функції інтерфейсу продовжують бути інтерактивними;
- Ресурси та можливості клієнту використовуються у повній мірі;
- Робоче навантаження з сервера розподіляється по клієнтах;
- ASP.NET Core веб сервер не є обов'язковою умовою розгортання, можна розгорнути застосунок, використовуючи "Serverless" технології, як наприклад Content Delivery Network (CDN).

Проте як завжди є свої обмеження теж, обираючи Blazor WebAssembly слід звертати увагу на те, що:

- З розподілу робочого навантаження на клієнтів випливає те, що застосунок обмежений ресурсами клієнта;
- Для роботи застосунку потрібна достатня кількість ресурсів браузеру та програмна складова, яка буде його підтримувати, наприклад браузер з підтримкою технології WebAssembly;
- Початковий час завантаження застосунку є більшим, ніж у варіанту розміщення Blazor Server та у фронтенд фреймворків на мові JavaScript.

Загалом можна зробити висновок, що моделі розміщення Blazor Server та Blazor WebAssembly є протилежностями один одної. Обмеження однієї моделі є перевагами іншої та навпаки. У розробці завжди слід обирати, враховуючи конкретні вимоги проекту, можливість серверів утримувати

належне навантаження у моделі Blazor Server та те, що при перебоях зв’язку з мережею досвід користувача буде не найкращим. Blazor WebAssembly ж в свою чергу є досить хорошим рішенням для сучасних пристроїв, які вже мають досить потужні обчислювальні можливості, також зараз переважно достатня швидкість з’єднання з мережею для завантаження більшого об’єму файлів.

Розуміючи недоліки моделі Blazor WebAssembly, їх можна вирішити за допомогою поєднання переваг розрахунків на сервері та у браузері, використовуючи WebAssembly, для цього потрібно зробити додаток розміщеним. Тоді ASP.NET Core сервер, який доставляє статичні файли може також слугувати пунктом для створення логіки та контролерів, які будуть її запускати на сервері та доставляти результати до клієнтів. Саме такий варіант розглянуто у роботі.

РОЗДІЛ 2
ПРОЕКТУВАННЯ ТА РОЗРОБКА КЛІЄНТ-СЕРВЕРНОГО ЗАСТОСУНКУ

2.1. Вибір алгоритму

Для того щоб здійснити порівняння реалізації обчислень на клієнті та на сервері, з та без використання програмних засобів для виконання коду на більш ефективній мові, нам потрібно обрати алгоритм. Для цього алгоритму в подальшому ми будемо заміряти час обчислень, аналізувати різні особливості реалізації в кожному з рішень.

Варіантів алгоритмів, які застосовуються сьогодні безліч. Одним з підрозділів задач, які часто потрібні та час виконання яких на практиці може бути великим є алгоритми на графах. У даній роботі серед задач з теорії графів була обрана задача про максимальний потік [12]. В цій задачі у нас є задана транспортна мережа, яка в себе включає:

- Орієнтований граф $G = (V, E)$;
- Вершину-джерело $s \in V$, вершину-стік $t \in V$;
- Пропускні здатності ребер $E \to \mathbb{R} \geq 0$.

Сама задача про максимальний потік полягає в знаходженні такого потоку між вершинами джерела і стоку, щоб його величина була максимальна.

Наприклад, для графа на рисунку 2.1 (числа в середині вершин позначають номер вершин, числа біля ребер – пропускну здатність відповідного ребра) максимальний можливий потік між вершинами 0 і 5 має значення 23, це значення потоків у кожному ребрі, що забезпечують таку величину потоку зображені на рисунку 2.2 (числа біля ребер позначають величину потоку у відповідному ребрі).

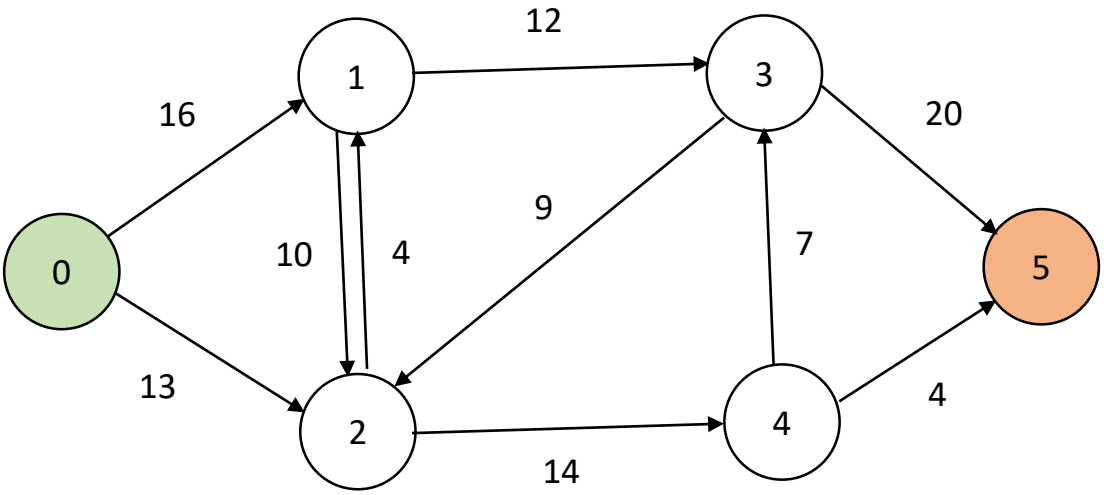


Рис 2.1 Граф з пропускними здатностями ребер, вершина 0 – джерело, вершина 5 - стік

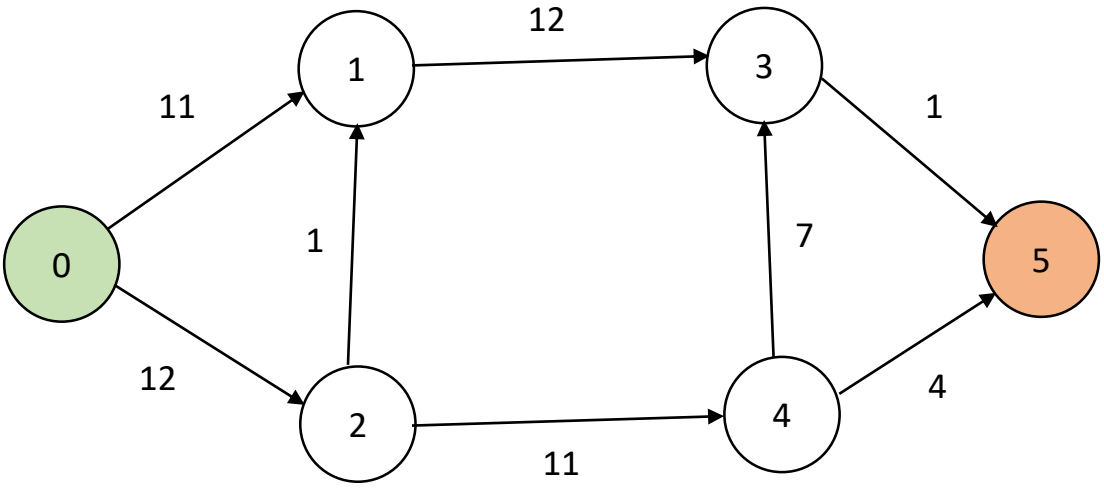


Рис 2.2. Потоки, що забезпечують максимальний потік між вершинами 0 і 5 графу на рисунку 2.1

Сфер застосування у цієї задачі дійсно багато. Прикладом практичного застосування задачі про максимальний потік є розрахунок максимально можливого потоку в системі труб (водопроводів, газопроводів, нафтопроводів тощо). Іншим варіантом застосування даної задачі може бути розрахунок кількості трафіку, який може проїхати з одного пункту (вершини джерела) в інший (стік) за певний проміжок часу. Це може бути корисним як при плануванні системи доріг, нових районів міста тощо. Також цікавим варіантом застосування цієї задачі може бути у

майбутньому, коли автомобілі будуть повністю автономними. В такому випадку автомобілі, що використовують автопілот, зможуть порівнювати максимальний можливий потік через певну частину дороги з кількістю автомобілів, які на ньому вже знаходяться. Таким чином, максимально уникаючи при плануванні маршруту частин дороги, де потік вже близький до максимального і, відповідно, уникати створення заторів на дорогах. Також ця задача може бути корисна у логістиці, проектуванні комп'ютерних мереж таким чином, щоб потік пакетів був максимальним.

Для вирішення цієї задачі існує декілька методів. Серед них найпопулярніший – алгоритм Форда-Фалкерсона. Його, а саме його часткових – алгоритм Едмондса-Карпа був використаний у роботі.

2.2 Алгоритм Едмондса-Карпа

Алгоритм Едмондса-Карпа є частковим випадком алгоритму Форда-Фалкерсона, тому спочатку варто описати загальні твердження для алгоритму Форда-Фалкерсона.

На максимальний потік між вершинами джерела і стоку накладаються такі обмеження:

- Потік через ребро не може не може перевищувати його пропускну здатність;
- Для всіх вершин, крім джерела та стоку, вхідний та вихідний потоки мають бути рівними.

Основною структурою, яка нам потрібна в алгоритмі виступає залишковий граф. Він вказує можливі додаткові потоки для заданої транспортної системи. Якщо у залишковому графі є шлях між джерелом та стоком, то це означає що можна додати потік. Кожне ребро залишкового графа має значення – залишкову пропускну здатність, яка рівна різниці пропускної здатності ребра в транспортній системі та потоку через нього в даний момент, тобто це пропускна здатність ребра на даний момент.

Для програмної реалізації алгоритму нам потрібно створити матрицю суміжності, значення в якій будуть відповідати залишковим пропускним здатностям. Якщо наша транспортна система задана матрицею суміжності, ми можемо за допомогою неї початково задати залишковий граф. Це можливо, адже якщо залишковий потік 0 – це означає, що між вершинами немає ребра, так само і в матриці суміжності графу, для якого знаходимо максимальний потік. Також, відповідно, залишковий потік через ребро в початковий момент буде рівний його загальній пропускній здатності, адже початковий потік у нас відсутній. Для знаходження додаткових шляхів для потоку у залишковому графі, ми можемо використовувати пошук в ширину або пошук в глибину. Знайшовши шлях, ми додаємо його до поточного потоку з величиною, рівною мінімальній залишковій пропускній здатності ребра на всьому шляху. Після того як ми знайшли шлях, потрібно оновити пропускні здатності в залишковому графі. Для всіх ребер, що знаходяться на шляху нового потоку ми віднімаємо від їх залишкової пропускної здатності величину нового потоку. Важливо також оновити залишкові пропускні здатності зворотних ребер, додавши до їх пропускних здатностей величину потоку, адже вони можуть нам знадобитись на наступних ітераціях алгоритму.

Як було сказано раніше, в роботі використаний алгоритм Едмондса-Карпа. Його основною ідеєю є використання пошуку в ширину в алгоритмі Форда-Фалкерсона, адже пошук в ширину завжди обирає шлях з найменшою кількістю ребер. При використанні пошуку в ширину, час в найгіршому випадку може бути зменшений до $O(VE^2)$ [13], при використанні ж матриці суміжностей для задання залишкового графу та транспортної мережі часова складність $O(EV^3)$.

2.3. Технологічні варіанти реалізації алгоритму

Обравши основні технології, які будуть використовуватись у роботі та розглянувши різні моделі клієнтського застосунку (“тонкий” та “товстий”)

переходимо до конкретного опису потрібних нам технологічних реалізацій. Для порівняння часу виконання алгоритму та зручності використання, переваг та недоліків технологій було обрано такі технологічні варіанти реалізації алгоритму:

- На сервері на мові Java;
- На сервері на мові Java з використанням ефективної мови програмування C++ за допомогою Java Native Interface (JNI);
- На клієнті на мові JavaScript;
- На клієнті з використанням ефективної мови програмування C++ за допомогою технології WebAssembly;
- На сервері на мові C#;
- На клієнті Blazor WebAssembly.

Для цього розроблено наступні програмні компоненти:

- Сервер на мові Java;
- Клієнт з використанням JavaScript фреймворку React;
- Blazor WebAssembly застосунок, розміщений на сервері ASP.NET Core (включає в себе клієнтську та серверну частину).

2.4. Сервер на мові Java

Після такого як ми визначились з алгоритмом, основними технологіями та технологічними варіантами, переходимо до програмної реалізації. Розробка була розпочата з сервера на мові Java.

Так як нам потрібно заміряти час виконання алгоритму, для порівняння продуктивності різних варіантів реалізації (на клієнті та на сервері, з чи без використання додаткових засобів для взаємодії з більш ефективною мовою програмування), варто розпочати з генерації даних для нашої задачі. Це потрібно, адже для більш точних замірів часу та для зручності, нам потрібен буде досить великий граф, задавати який вручну довго та незручно.

Так як граф для нашого алгоритму потрібно задати матрицею суміжності, в коді було створено сервіс MatrixService. Його задачами є – генерація матриці суміжності заданого аргументом розміру та збереження останньої згенерованої матриці, для того щоб її можна було отримати та використати для усіх методів розрахунку алгоритму в системі. Схема генерації наступна:

$\begin{pmatrix} a & \cdots & b \\ \vdots & \ddots & \vdots \\ b & \cdots & a \end{pmatrix}$, де a – число від 1 до максимальної пропускної здатності ребра (константи), b – число від 0 до максимальної пропускної здатності ребра. Так як в матриці суміжності елемент і-го рядка та j-го стовпчика вказує на пропускну здатність ребра між вершинами і та j чи на його відсутність (у випадку 0). Така схема потрібна, адже будемо рахувати потік між першою (нульовою) вершиною і останньою і для того щоб гарантовано був мінімум один шлях між ними (послідовний з першої вершини у другу, з другої в третю і так далі).

Крім цього, на сервері реалізовані і самі обчислення алгоритму Форда-Фалкерсона. Перший варіант реалізації алгоритму – на мові програмування Java. Він реалізований у сервісі FordFulkersonMaxFlowService. Так як Java – строго типізована об'єктно-орієнтована мова програмування, з синтаксисом схожим на C/C++, особливостей реалізації чи певних додаткових складностей немає. Основний метод в сервісі fordFulkerson приймає такі алгоритми: граф (матриці суміжності) та номери вершин джерела та стоку.

Наступним пунктом в реалізації був алгоритм Форда-Фалкерсона, використовуючи засоби взаємодії з більш ефективною мовою. На сьогодні більшість задач, які потребують дійсно високої продуктивності реалізовують на C++. Тому для того, щоб ми могли використати код написаний на C++ у нашому сервері, нам знадобиться JNI (Java Native Interface) [14].

Для виклику алгоритму Форда-Фалкерсона на C++, використовуючи JNI в проекті сервера присутній сервіс FordFulkersonJNI. В ньому

відбувається виклик статичного методу System.loadLibrary – методу, який завантажує бібліотеки з файлу системи в пам’ять і надає доступ до експортованих функції нашому Java коду. Також для того, щоб код з завантаженої бібліотеки використовувався методом, який ми будемо викликати для обчислення алгоритму Форда-Фалкерсона, нам потрібно додати до нього ключове слово “native”. Всі методи позначені ключовим словом “native” повинні бути реалізовані в завантажуваних бібліотеках.

Після цього було реалізовано код мовою C++ у файлах FordFulkerson.h та FordFulkerson.cpp. Для того щоб ми могли його використовувати потім у Java коді у цих файлах нам теж потрібні деякі ключові слова:

- JNIEXPORT- позначає функцію в бібліотеці як експортовану, тобто вона буде включена в таблицю функції і JNI зможе знайти її;
- JNICALL – в поєднанні з JNIEXPORT, це ключове слово забезпечує доступ до наших методів з JNI;
- JNIEnv – структура, що містить методи, які ми можемо використовувати в native коді для доступу до елементів Java;
- JavaVM – структура, що дозволяє нам керувати поточною JVM, додавати нові потоки тощо.

Створивши потрібний Java та потрібні C++ файли, наступним кроком потрібно було скомпілювати C++ код так, щоб ми могли його використати в нашому сервері. Для цього нам потрібен G++ компілятор (в нашому випадку це встановлений MinGW-w64 на системі з Windows 10). Далі за допомогою нього був створений Compiled Object File, використовуючи наступну команду:

```
g++ -c -I%JAVA_HOME%\include -I%JAVA_HOME%\include\win32
com_rmv_jni_FordFulkersonJNI.cpp -o
com_rmv_jni_FordFulkersonJNI.o -O3
```

Важливим при компіляції є параметр оптимізації -O3, адже він суттєво буде впливати на час виконання native функції. Після того, як був отриманий

Compiled Object File, компілюємо dll бібліотеку, яку і буде завантажувати Java:

```
g++ -shared -o native.dll com_rmv_jni_FordFulkersonJNI.o -Wl, --add-stdcall-alias
```

Після того, як у нас є скомпільована бібліотека, нам потрібно при запуску програми використовувати додатковий аргумент -Djava.library.path, який буде вказувати, де знаходяться бібліотеки, які мають бути завантаженні для роботи native коду.

Крім реалізованих генерації матриці суміжності та реалізації алгоритмів, нам потрібен також сервіс, що буде вимірювати час виконання алгоритмів Форда-Фалкерсона на Java та на C++ з використанням JNI. Для цього в коді є FordFulkersonTimeFlowService, який містить 2 методи що повертають відповідний час виконання та результат алгоритму (максимальний потік).

Для того щоб до нашого сервера можна було здійснювати запити та отримувати необхідну для клієнта інформацію є 2 контролери, в яких реалізовані наступні запити:

- GET “/matrix”, з параметром size – у відповідь повертає нову згенеровану матрицю розміру size, зберігає її на сервері як останню згенеровану;
- GET “/time/java” – у відповідь повертає час виконання та результат алгоритму Форда-Фалкерсона на Java у мілісекундах для останньої згенерованої матриці;
- GET “/time/jni” – у відповідь повертає час виконання та результат алгоритму Форда-Фалкерсона на C++, використаного за допомогою JNI у мілісекундах для останньої згенерованої матриці.

2.5. Клієнт з використанням React

Поставленими задачами клієнта, що мають бути реалізованими та які він має виконувати були:

- Відображення графу (для графів невеликого розміру)
- Відправлення запиту на сервер для отримання нового графу
- Розрахунок алгоритму Форда-Фалкерсона мовою JavaScript
- Розрахунок алгоритму Форда-Фалкерсона використовуючи більш ефективну мову та засобів взаємодії з нею
- Вимірювання часу алгоритмів, що реалізованих у клієнті
- Відправлення запитів на сервер для отримання часу виконання та результатів алгоритмів Форда-Фалкерсона на мові Java та C++ з використанням JNI

Для відображення графу в клієнті було використано react-graph-vis [15]. Вершина джерело позначена зеленим кольором, стік – червоним, приклад відображення на рисунку 2.3.

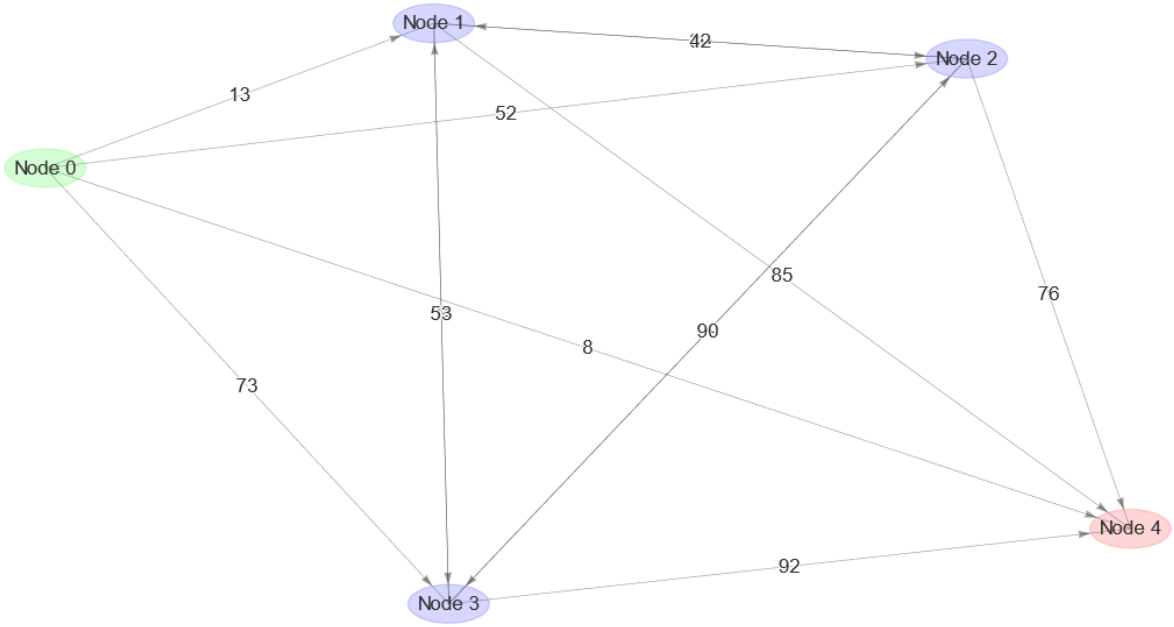


Рис 2.3. Приклад відображення графу в клієнті

Під відображенням графа знаходяться (рис. 2.4):

- Поле для вводу кількості вершин нового графу
- Кнопка для відправлення запиту на сервер для генерації нового графу (його матриці суміжності)
- Кнопка для виклику алгоритму Форда-Фалкерсона реалізованого в клієнті мовою JavaScript

- Кнопка для виклику алгоритму Форда-Фалкерсона реалізованого в клієнті використовуючи WebAssembly
- Кнопка відправлення запиту на сервер для отримання часу виконання та результату алгоритму Форда-Фалкерсона реалізованого на сервері мовою Java
- Кнопка відправлення запиту на сервер для отримання часу виконання та результату алгоритму Форда-Фалкерсона реалізованого на сервері, використовуючи C++, за допомогою JNI

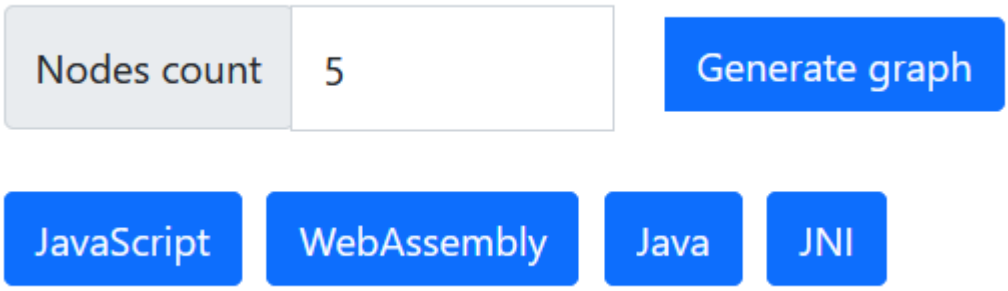


Рис 2.4. Поле вводу кількості вершин для генерації графу, кнопки для виклику методів

Кожна з кнопок, що запускає алгоритм на клієнті чи відправляє запит на сервер для отримання часу виконання та результату по завершенню методу додає новий запис у таблицю (рис 2.5) – у відповідну колонку час та значення максимального потоку.

Calculation time, ms

JavaScript	WebAssembly	Java	JNI	Max flow
2923	1732	1551	649	JS: 48935 WASM: 48935 Java: 48935 JNI: 48935
2958	1729	1577	652	JS: 48935 WASM: 48935 Java: 48935 JNI: 48935
3229	1731	1558	652	JS: 48935 WASM: 48935 Java: 48935 JNI: 48935

Рис 2.5. Приклад таблиці результатів 3 повторів запусків всіх методів розрахунку для графу з 1000 вершин

Реалізація алгоритму Форда-Фалкерсона значних особливостей немає. Тому перейдемо до реалізації, використовуючи WebAssembly для запуску C++ коду.

Спочатку було створено файл FordFulkerson.cpp з реалізацією алгоритму на мові C++. Наступним кроком нам потрібно скомпілювати код в WebAssembly. Для цього використано компілятор Emscripten [16]. Компіляцію було виконано за допомогою наступної команди:

```
emcc cpp/FordFulkerson.cpp -s WASM=1 -s MODULARIZE=1 -s
EXPORTED_FUNCTIONS=_fordFulkerson,_malloc,_free -s
EXPORTED_RUNTIME_METHODS=ccall,cwrap -o FordFulkerson.js
-s ALLOW_MEMORY_GROWTH=1 -O3
```

Використані аргументи в компіляції:

- -s WASM=1 – вказує, що ми хочемо отримати wasm модуль;
- -s MODULARIZE=1 – поміщає весь згенерований код в фабричну функцію для створення модуля;
- -s EXPORTED_FUNCTIONS – вказує, які функції ми хочемо експортувати;
- -s EXPORTED_RUNTIME_METHODS=ccall,cwrap – надає доступ до методів ccal та cwrap для виклику C++ функції з JavaScript коду;
- -O3 – параметр оптимізації C++ коду при компіляції;
- -s ALLOW_MEMORY_GROWTH=1 – дозволяє збільшувати розмір пам’яті, в залежності від потреб.

Після цього можна підключити створені файли у коді клієнта та використовувати експортовані методи. Для передачі матриці в файл використовується код з рисунку 2.6

```
const input_array = new Int32Array(arr);
const len = input_array.length;
const bytes_per_element = input_array.BYTES_PER_ELEMENT;
const input_ptr = Module._malloc(len * bytes_per_element);
Module.HEAP32.set(input_array, input_ptr / bytes_per_element);
```

Рис 2.6. Передача матриці суміжності з JavaScript коду для WebAssembly

Більш детально про отримання доступу до пам’яті з JavaScript описано в документації Emscripten [17], інформація з якої була використана для передачі матриці суміжності. Так як в Heap пам’яті байти записані послідовно, то і передаємо ми одновимірний масив байтів, для запису матриці в одновимірний масив було використано порядковий порядок запису матриці [18]. Це призвело до змін у коді алгоритму Форда-Фалкерсона, а саме в початковому заповнені залишкового графу, елемент матриці a_{ij} буде знаходитись так: $graph[i \times size + j]$, де $graph$ - матриця суміжності графу транспортної мережі, $size$ – кількість вершин у графі.

2.6. Blazor WebAssembly застосунок

Для створення Blazor WebAssembly застосунку було використано середовище розробки Microsoft Visual Studio 2022, адже воно має пряму та повну підтримку технології Blazor. Воно зручне тим, що вхід у технологію є легким та зручним, адже є декілька шаблонів для проектів, які представлені візуально, тому проект не потрібно створювати, використовуючи .NET CLI [19], що досить схожий на те, як працює скрипт Create React App [20]. Для нашого проекту потрібен шаблон, який зображено на рисунку 2.7:

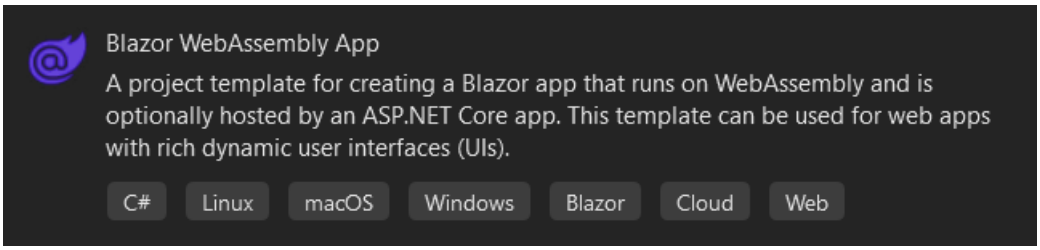


Рис 2.7. Шаблон для Blazor WebAssembly у середовищі розробки Microsoft Visual Studio 2022

Після обрання шаблону далі йдуть звичні для багатьох середовищ розробки поля назви проекту тощо. Проте для нас особливо важливе поле “ASP.NET Core hosted” серед налаштувань:

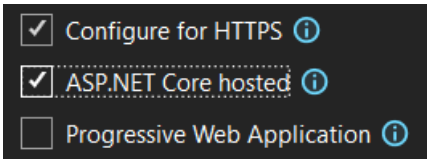


Рис 2.8. Частина налаштувань проекту Blazor WebAssembly

Саме це поле відповідає за те, що наш застосунок буде розміщеним, адже стандартний застосунок Blazor WebAssembly не містить в собі серверної частини та є досить схожим на JavaScript фронтенд фреймворк, з основною різницею у тому, що використовується мова C# для написання логіки, а не звичний для таких задач JavaScript.

Завершивши кроки налаштувань та створивши проект, отримаємо таку структуру:

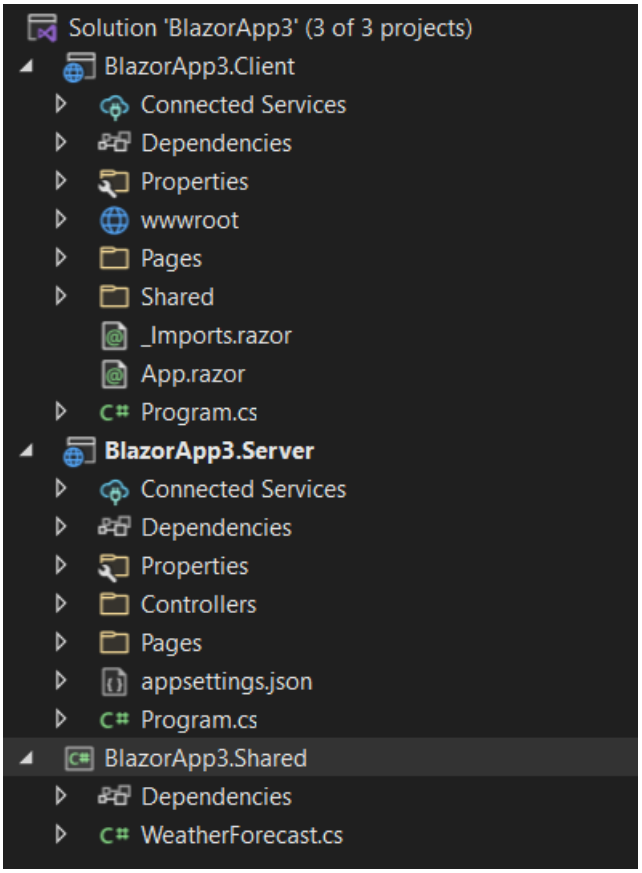


Рис 2.8 Структура Blazor WebAssembly ASP.NET Core розміщеного застосунку

Проект містить у собі три основні частини:

– Клієнт – присутній у будь-якому Blazor WebAssembly застосунку;
– Сервер, адже у нас ASP.NET Core розміщений застосунок;

– Спільну частина – класи, які можна використовувати і на клієнті.

Спільна частина є тим, чого немає у інших звичних технологічних варіантах розробки клієнт-серверних застосунків, адже зазвичай вони написані на 2 різних мовах програмування – JavaScript та мова серверу. Можливість писати на одній мові програмування – C# і на сервері, і на клієнті дозволяє уникати дублювання коду, створення у двох місцях (на сервері та на клієнті) класів для запитів, відповідей та будь-яких інших класів.

Перейдемо до реалізації функціоналу у застосунку. Для заміру часу роботи алгоритму необхідно для початку його реалізувати. Враховуючи те, що алгоритм нам буде потрібний і на клієнті, і на сервері, то замість того, щоб реалізовувати його два рази – на клієнт та на сервері, можна винести його одразу у спільну частину, адже для цього вона і присутня у розміщених проектах Blazor WebAssembly. Сам алгоритм реалізовано у класі FordFulkersonMaxFlowService. C# має досить звичний синтаксис, схожий на C++ та Java. Окрім синтаксису двовимірного масиву (матриці), особливостей небагато. Головний метод, який реалізує алгоритм Форда-Фалкерсона - public static int fordFulkerson(int[,] graph, int s, int t). Його аргументами є граф, для якого розраховується значення максимального потоку, початкова та кінцева вершина між якими знаходиться значення потоку.

Для заміру часу та порівняння роботи реалізацій у спільній частині створено сервіс FordFulkersonTimeFlowService, метод якого getTimeFlow(int[,] matrix) повертає час виконання та результат роботи алгоритму – максимальний потік у вигляді класу TimeFlowResponse, що теж знаходиться у спільній частині.

Ці основні два сервіси нам будуть потрібні для використання для проведення тестів. Першою імплементацією буде серверна. Для цього потрібно контролер, який буде викликати метод getTimeFlow(int[,] matrix) сервісу FordFulkersonTimeFlowService, що знаходиться у спільній частині.

Для запуску алгоритму нам потрібно зчитати матрицю суміжності, що знаходиться у файлі. Задля цієї мети створений окремий сервіс – ReadMatrixService, метод якого readMatrix() і виконує усю роботу. Маючи усі три сервіси: ReadMatrixService, FordFulkersonMaxFlowService, FordFulkersonTimeFlowService, наступним кроком було створення контролера – FordFulkersonTimeController, який на запит GET /FordFulkersonTime буде зчитувати матрицю суміжності графу та запускати алгоритм, після чого відповість нам об’єктом класу TimeFlowResponse, що містить у собі максимальний потік та час виконання алгоритму.

На цьому серверна частина була завершена, далі потрібен клієнт, який буде викликати по API запуск алгоритму на сервері та також вміти обраховувати алгоритм сам у собі. Це все буде розміщено на одній сторінці, вигляд якої можна побачити на рисунку 2.9:

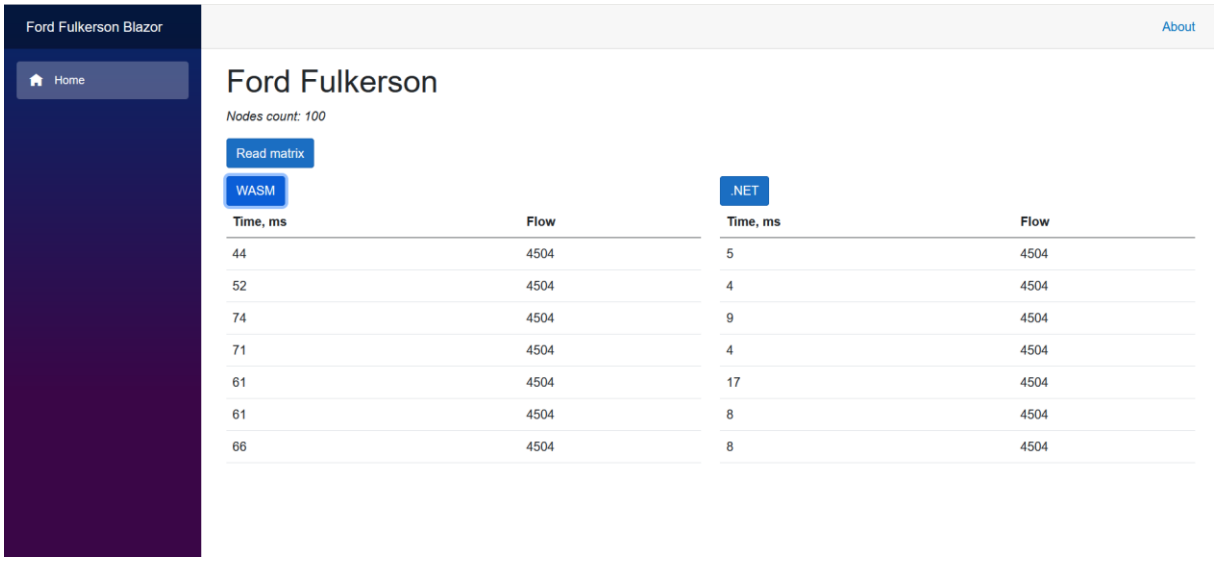


Рис 2.9 Вигляд інтерфейсу застосунку Blazor

На сторінці є 3 кнопки:

– Read matrix – читає матрицю з файлу;
– WASM – запускає алгоритм у Blazor WebAssembly, тобто на клієнті, додає запис результату та часу виконання у відповідну таблицю під нею;

– .NET – робить запит до сервера для обрахунку алгоритму на ньому, додає запис результату та часу виконання у відповідну таблицю під нею.

Blazor має допоміжні класи, які дають можливість легко зчитувати статичні файли та робити запити до API в асинхронному режимі. Зчитавши матрицю суміжності графу з статичного файлу можемо запускати обчислення алгоритму для нього. Для цього використовується метод на рисунку 2.10:

```
private void onClickWasm()
{
    if (matrix != null)
    {
        var timeFlowWasm = FordFulkersonTimeFlowService.getTimeFlow(matrix);
        timeFlowWasmList.Add(timeFlowWasm);
    }
}
```

Рис 2.10 Виклик алгоритму у Blazor WebAssembly

Як видно на рисунку, у фреймворку Blazor це дуже зручно, нам не потрібно писати якусь додаткову логіку на JavaScript, як зазвичай при розробці клієнтів, ми використовуємо одну мову для написання усього в проекті.

РОЗДІЛ 3
ПОРІВНЯННЯ РЕАЛІЗОВАНИХ АЛГОРИТМІВ НА КЛІЄНТІ ТА НА СЕРВЕРІ

3.1. Порівняння часу виконання

Для порівняння ефективності алгоритмів, були згенеровані графи з кількістю вершин 300, 500, 1000 та 2000. Відповідні часи виконання представлені у таблицях 3.1, 3.2, 3.3 та 3.4.

Таблиця 3.1 - Час виконання алгоритму Форда-Фалкерсона у мілісекундах для графу з 300 вершинами:

№ запуску	**JavaScript**	**WebAssembly**	**Java**	**JNI**	**Blazor WASM**	**C#**
1	121	56	52	22	1277	81
2	113	64	50	23	1382	78
3	111	58	55	30	1455	83
4	112	54	52	24	1388	80
5	106	61	53	29	1371	76
Середній	113	59	52	26	1375	80

Таблиця 3.2 - Час виконання алгоритму Форда-Фалкерсона у мілісекундах для графу з 500 вершинами:

№ запуску	**JavaScript**	**WebAssembly**	**Java**	**JNI**	**Blazor WASM**	**C#**
1	427	229	199	105	5614	397
2	397	216	205	91	4935	371
3	398	229	201	88	6060	379
4	402	220	206	104	4883	371
5	407	224	198	100	5719	370
Середній	406	223	202	98	5442	378

Таблиця 3.3 - Час виконання алгоритму Форда-Фалкерсона у мілісекундах для графу з 1000 вершин:

№ запуску	**JavaScript**	**WebAssembly**	**Java**	**JNI**	**Blazor WASM**	**C#**
1	3133	1779	1627	694	47477	2540
2	3407	1899	1608	686	52645	2752
3	3138	1894	1603	695	52542	2726
4	3418	1779	1604	689	50378	2681
5	3150	1850	1600	697	48479	2680
Середній	3249	1840	1608	692	50304	2676

Таблиця 3.4 - Час виконання алгоритму Форда-Фалкерсона у мілісекундах для графу з 2000 вершин:

№ запуску	**JavaScript**	**WebAssembly**	**Java**	**JNI**	**Blazor WASM**	**C#**
1	22645	15361	13494	5571	322743	22862
2	25505	15349	13439	5574	324623	22768
3	22346	15625	13427	5562	328910	23024
4	22694	15632	13477	5769	301536	22813
5	22773	15318	13442	5543	319034	22886
Середній	23193	15457	13456	5604	319369	22871

В усіх тестах середній час виконання на JNI найкращий, після нього другий результат – у Java, третій – у WebAssembly, четвертий – у C# (Blazor серверна частина), п’ятий – у JavaScript і шостий – у Blazor.

3.2. Переваги та недоліки технологій реалізації алгоритмів

За часовою ефективністю бачимо, що у всіх тестах середній час виконання JNI був найкращим, отже JNI можна і варто використовувати у

випадках, коли нам необхідне якесь складне обчислення на сервері, мова C++ дає відчутний приріст у швидкості.

Другий результат у тестах у мови Java – вона є достатньо ефективною для обчислень, але не настільки, як мова C++.

Третій результат у WebAssembly – запуск алгоритму на мові C++ на стороні клієнта. Не зважаючи на хороший результат, слід також пам’ятати що ресурси клієнта обмежені, у кожного користувача вони різні, тому слід думати чи вистачить у користувачів, які будуть запускати клієнт у браузері ресурсів для швидких обчислень. Але якщо нам необхідно щось обчислювати на стороні клієнту, то це рішення є дійсно ефективним.

Четвертий результат у сервера на мові C# - в цих тестах мова проявила себе повільнішою, ніж Java, але у неї присутня велика кількість бібліотек, які є дуже ефективними в плані продуктивності.

П’ятий результат у JavaScript – очікувано розрахунки на стороні клієнта є повільнішими, ніж на сервері. Варто враховувати те, що складні розрахунки за можливості краще виконувати на сервері.

Шостий результат у Blazor WebAssembly – на порядок гірший за часом, ніж у інших технологічних варіантів. Але ця технологія стрімко розвивається та серед її плюсів є інші речі, не враховуючи швидкість розрахунку алгоритмів. Якщо присутні складні розрахунки, слід за можливості їх переносити на серверну частину Blazor.

Крім часової ефективності важливо також, щоб технології були зручними для використання, тому хочеться звернути увагу на деякі переваги та недоліки, використаних у роботі технологій.

Java:

+ Популярна мова програмування з класичним синтаксисом;
+ Велика кількість інформаційних ресурсів;
+ Строга типізація – може бути зручною у реалізації алгоритмів;
+ Кросплатформність;
– Компіляція в байткод сповільнює час виконання.

JNI:

+ Досить популярний механізм для прискорення виконання алгоритмів (запуску їх на C++);
+ Достатня кількість інформаційних ресурсів;
– Використання JNI позбавляє мову Java однієї з переваг – кросплатформності.
– Потрібно додавати конвертацію даних з Java для використання їх у коді C++;
– Відлагодка.

JavaScript:

+ Популярна мова програмування;
+ Велика кількість інформаційних ресурсів;
– Динамічна типізація – може ускладнювати реалізацію та відлагодку алгоритмів;
– Передостанній час виконання алгоритмів серед технологій використаних у роботі.

WebAssembly:

+ Можливість запуску ефективних мов програмування (C/C++/Rust) у браузері;
+ Відчутне прискорення порівняно з JavaScript;
– При використанні потребує додаткових налаштувань, у React потрібно настроювати file loaders, використовувати webpack;
– Потребує додаткової роботи з пам’яттю в JavaScript.

Blazor WebAssembly:

+ Можливість писати клієнтську частину без знань JavaScript;
+ Легкий для освоєння;
+ Строга типізація у методах з логікою;
+ Можливість створення спільних класів для сервера та клієнта;

– Не дає приросту ефективності, як звичний WebAssembly та C++, а навпаки має повільну швидкодію;
– Розмір сайтів досить великий, через те що все завантажується у браузер.

C#:

+ Одна з найпопулярніших мов програмування з класичним синтаксисом мов родини C;
+ Достатня кількість інформаційних ресурсів;
+ Строга типізація – може бути зручною у реалізації алгоритмів;
+ Наявність ефективних бібліотек, наприклад для роботи з файлами;
– Не показала кращої ефективності, ніж Java у тестах.

ВИСНОВКИ

У даній роботі реалізовано клієнт-серверні застосунки, використовуючи Java з Spring Framework для сервера та JavaScript з React для клієнта та мову C# з Blazor WebAssembly. Для реалізації алгоритмів на клієнті та на сервері було використано мову Java, мову C# мову C++ та JNI для взаємодії сервера з нею, JavaScript, C++ та WebAssembly для взаємодії клієнта, C# та Blazor WebAssembly для запуску у браузері.

Отримані часові результати виконання алгоритмів на різних наборах даних вказують на те, що обчислення на сервері є ефективнішими, ніж на клієнті.

Якщо ж обчислення на сервері здійснювати неможливо (потрібні постійні складні обчислення для кожного користувача клієнту), то для розрахунків складних алгоритмів у клієнті (для ігор, симуляцій) можна використовувати WebAssembly, який надає відчутну перевагу в часі виконання алгоритмів, порівняно з JavaScript, але потребує додаткових налаштувань.

Для прискорення обчислень в Java застосунках, можна використовувати JNI для запуску коду написаного мовою C++ зі значною перевагою у часі виконання. Проте під час використання JNI слід звертати увагу на його недоліки – позбавлення кросплатформності застосунку та необхідну реалізацію конвертації даних з Java.

У роботі розглянуто сучасний фреймворк – Blazor WebAssembly. Він, на відміну від JavaScript фреймворків, використовує мову C# для написання логіки і запускає її за допомогою технології WebAssembly у браузері. Це є дійсно новим та має ряд переваг: можливість уникнення дублювання однакового коду для запитів і відповідей на клієнті та на сервері на різних мовах – замість цього їх можна винести у спільну частину, адже клієнт та сервер написані на одній мові. Це також полегшує опанування фронтенд технології для тих, хто не стикався з JavaScript у попередньому досвіді.

Отримані результати можуть бути корисними при прийнятті рішень при розробці клієнт-серверних застосунків щодо місця реалізації алгоритму – клієнт чи сервер. Також результати демонструють переваги засобів JNI та WebAssembly для прискорення роботи складних алгоритмів на сервері та на клієнті відповідно.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Distributed Application Architecture. Sun Microsystem, 2009. 128 с.

2. GitHut 2.0 [Електронний ресурс]. – Режим доступу: https://madnight.github.io/githut/#/pull_requests/2021/1

3. Spring Framework [Електронний ресурс]. – Режим доступу: https://spring.io/

4. Model–view–controller [Електронний ресурс]. – Режим доступу: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

5. Spring. Building a RESTful Web Service [Електронний ресурс]. – Режим доступу: https://spring.io/guides/gs/rest-service/

6. Single-page application vs. multiple-page application [Електронний ресурс]. – Режим доступу: https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58

7. React [Електронний ресурс]. – Режим доступу: https://uk.reactjs.org/

8. Front-end frameworks popularity (React, Vue and Angular) [Електронний ресурс]. – Режим доступу: https://gist.github.com/tkrotoff/b1caa4c3a185629299ec234d2314e190

9. ASP.NET Core Blazor [Електронний ресурс]. – Режим доступу: https://docs.microsoft.com/uk-ua/aspnet/core/blazor/?view=aspnetcore-6.0

10. NuGet [Електронний ресурс]. – Режим доступу: https://www.nuget.org/

11. ASP.NET Core Blazor hosting models [Електронний ресурс]. – Режим доступу: https://docs.microsoft.com/uk-ua/aspnet/core/blazor/hosting-models?view=aspnetcore-6.0

12. Maximum flow problem [Електронний ресурс]. – Режим доступу: http://www.mi.fu-berlin.de/wiki/pub/Main/GunnarKlauP1winter0708/discMath_klau_maxflow.pd

13. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. The MIT Press. Cambridge, Massachusetts London, England, 2009. 727 с.

14. Guide to JNI (Java Native Interface) [Електронний ресурс]. – Режим доступу: https://www.baeldung.com/jni

15. React graph vis [Електронний ресурс]. – Режим доступу: https://www.npmjs.com/package/react-graph-vis

16. Emscripten [Електронний ресурс]. – Режим доступу: https://emscripten.org/index.html

17. Emscripten. Access memory from JavaScript [Електронний ресурс]. – Режим доступу: https://emscripten.org/docs/porting/connecting_cpp_and_javascript/Interacting-with-code.html#access-memory-from-javascript

18. Row- and column-major order [Електронний ресурс]. – Режим доступу: https://en.wikipedia.org/wiki/Row-_and_column-major_order

19. .NET CLI для Blazor [Електронний ресурс]. – Режим доступу: https://docs.microsoft.com/en-us/learn/modules/build-blazor-webassembly-visual-studio-code/3-exercise-configure-enviromnent?pivots=vscode

20. Create React App [Електронний ресурс]. – Режим доступу: https://github.com/facebook/create-react-app