Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

ОСОБЛИВОСТІ РОЗРОБКИ ПРОГРЕСИВНИХ ВЕБ ЗАСТОСУВАНЬ З ВИКОРИСТАННЯМ WEBASSEMBLY

Текстова частина до курсової роботи
за спеціальністю „Програмна інженерія” 121

Керівник **курсової роботи**
ас. Калітовський Богдан Віталійович

_____
(підпис)
“____” _____ 2021 р.
Виконав студент
Петрик Ярослав Ігорович
“____” _____ 2021 р.

Київ 2021

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ
Зав.кафедри мултимедійних систем,
к.ф.-м.н., доц. О. П. Жежерун

_____
(підпис)
„____”_____ 2021 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

студенту 3-го курсу, факультету інформатики Петрик Ярославу Ігоровичу

ТЕМА: Особливості розробки прогресивних веб застосувань з використанням WebAssembly

Вихідні данні:

Результати порівнянь різний моделей виконання в веб середовищі
Результати порівнянь сучасних двигунів веб браузерів

Зміст текстової частини до курсової роботи:

Індивідуальне завдання
Анотація
Вступ
1 Детальний огляд технології WebAssembly
2 Розширення можливостей веб платформи
3 Порівняння різних моделей виконання
4. Особливості розробки прогресивних веб застосувань
Висновки
Список літератури

Дата видачі „___” _____ 2020 р.

Керівник (підпис)

Завдання отримав (підпис)

Календарний план виконання курсової роботи

Тема: Особливості розробки прогресивних веб застосувань з використанням WebAssembly

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на дипломну роботу	11.11.2020	
2.	Огляд технічної літератури за темою роботи	10.12.2020	
3.	Виконання аналізу сучасних методів розробки	20.02.2020	
3.	Початкова розробка додатку	10.03.2020	
4.	Підготовка матеріалу до доповіді “Building PWAs in SWI-Prolog”	6.04.2020	
5.	Розширення можливостей додатку задля дослідження більшої кількості особливостей.	26.04.2020	
6.	Розробка методики збирання даних	7.05.2020	
7.	Початок написання текстової частини	8.05.2020	
8.	Збір та аналіз даних	12.05.2011	
9.	Корегування роботи за уточненнями наукового керівника	16.05.2021	
10.	Остаточне оформлення пояснювальної роботи та слайдів	17.05.2021	
11.	Захист курсової роботи	24.05.2021	

Студент: **Петрик Ярослав Ігорович**

Керівник: **Калітовський Богдан Вітальович**

“_____”_____

Зміст

Анотація

Метою дослідження є ознайомлення та аналіз стандарту WebAssembly, його імплементації у різних середовищах, аспектах розвитку, та характеристиках швидкодії. Розглянуто також інноваційні можливості веб платформи пов’язані із розробкою так званих “Прогресивних Веб Застосувань”.

Досліджено особливості створення та виконання коду розробленого під WASM платформу. Для аналізу швидкодії та знаходження можливих застосувань, було розроблено прогресивний веб застосунок “wasm-web-checkers”. Для розробки та порівнянь було використано мови TypeScript, Rust та Prolog (завдяки С проекту SWI-Prolog). Проведено змістовне тестування та аналіз отриманих даних.

Ключові слова: WebAssembly, WASM, web, Progressive Web App, PWA, Rust, Prolog, SWIPL, JavaScript, Web Worker, Service Worker, Emscripten, web performance.

Перелік прийнятих скорочень

WASM – WebAssembly

WAT – WebAssembly Text Format

AOT – Ahead-Of-Time Compilation

JIT – Just-In-Time Compilation

JS – JavaScript

SWIPL – SWI-Prolog

API – Application Programming Interface (програмний інтерфейс)

GC – Garbage Collector (збирач сміття)

PWA – Progressive Web Application (прогресивний веб застосунок)

TS – TypeScript

JSON – JavaScript Object Notation

SW – Service Worker

Вступ

Від часу свого зародження у 1989 році веб платформа набирала шаленої популярності, а також нових можливостей. Веб дав спроможність швидко отримати доступ до інформації усього людства, та з часом інноваційні додатки почали вживати цю легкість використання задля того, щоб поширювати свої застосунки. Нині майже кожна компанія мають свої сторінки чи додатки у всесвітній павутині.

Із ростом потреб користувачів, ростуть і можливості веб платформи. JavaScript нині є головним двигуном прогресу веб застосунків у браузерах користувачів. Від сайтів магазинів, до складних веб застосувань, JavaScript виконує ключову роль для надання сервісу користувачам. JavaScript є ексклюзивною мовою додатків платформи. Проте веб росте, і потреби людей разом із ним.

З одного боку існуючі застосування мають на меті отримати гнучкість моделі розповсюдження вебу. Для сучасних розробників мати можливість працювати на будь-якому пристрої в світі є доволі привабливою пропозицією. Проте ексклюзивність веб платформи лише для JavaScript застосунків із власною моделлю виконання не сприяє можливості використовувати веб в таких цілях.

З іншого боку багато застосунків мають дуже високі потреби у очікуваній швидкодії. За роки покращень JavaScript двигунів, JS більше не має клейма повільної мови, та проте через особливості динамічної натури середовища, та моделі виконання, JavaScript часто не може надати достатньо передбачуваних результатів (особливо від різних вендорів середовищ виконання). Також високорівневість не надає можливості контролювати деякі низькорівневі аспекти середовища, через що маємо деякі перепони у реалізації низькорівневих оптимізацій.

І остання проблема веб платфоми станом на 2021 рік, є залежність від мережі. Користувачі звикли отримувати очікуваний досвід від використання нативних додатків. Зазвичай веб додатки не мають “блиск” та можливостей своїх рідних застосувань.

Задля вирішення цих проблем, веб платформа надає відповідний інструментарій: WebAssembly та функціонал Прогресивних Веб Застосунків. Саме ці аспекти будуть вивчені в ході курсової роботи.

РОЗДІЛ 1: Детальний огляд технології WebAssembly

1.1 Загальні відомості

WebAssembly (також відомий як WASM) – Це відкритий стандарт для портативного бінарного коду, призначений для виконуваних програм[1]. Це стандартизований набір інструкцій, та модель виконання розроблена в першу чергу для ви-конання високоефективного коду.

Сам формат WASM не перезначений для написання власноруч. Він виступає як ще одна кінцева архітектура для компіляції. Тобто замість того щоб компілювати C, C++, Rust чи інший код до x86 чи arm машинного коду, програма компілюється до WASM формату.

1.2 Модель виконання

Першочерговою задачею WebAssembly було створення безпечного, високо-ефективного та портативного формату для виконання коду на веб платформі. Тому WASM працює в «пісочниці», що не має жодного доступу до зовнішнього оточення. Всі взаємодії із хост-середовищем лімітовані до функціоналу наданого самим середовищем (детальніше про це буде пізніше).

Окрім стандартизованого набору інструкцій, стандарт WebAssembly також описує віртуальну машину та модель виконання. Віртуальна машина базована на стеку. Це відмінно контрастує із реальними машинами, що засновані на моделі виконання з багатьма регістрами, проте такий підхід дозволяє бути портативним для багатьох платформ.

Модель пам’яті в WASM світі також слідує спрощеним правилам. Для модуля доступна лінійна пам’ять із доступом на читання та запис. Також значення локальні для WebAssembly функцій, обмежені в статично відомі розміри. Все це означає, що керування пам’яттю напряму надається розробнику WASM модуля. Жодних гарантій щодо існування системи автоматичного збору сміття не існує.

Через те що WebAssembly описує поведінку віртуальної машини, перед виконанням на реальному залізі, WASM код потребує додаткових перетворень. Режим виконання залежить від реалізації рантайму, але традиційно можна використовувати один із трьох підходів: інтерпретація, Ahead-of-time compilation (AOT) чи Just-in-time compilation (JIT). Ці три різні підходи буде описано далі при порівнянні різних мов та моделей виконань.

Спрощений набір інструкцій, компактний двійковий формат, статична “Ahead-of-time” компіляція та повний контроль над пам’яттю зазвичай дозволяє WASM програмам виконуватись швидше та використовувати менше оперативної пам’яті ніж порівняно еквівалентний код динамічної мови, як JavaScript із використанням JIT компіляції.

1.3 Набір інструкцій та WebAssembly Text Format (WAT)

Формат WASM бінарний, і створений спеціально із урахуванням швидкості парсингу інструкцій. Читати людям операційні коди побайтово далеко не зручно, і саме тому стандартом зазначений текстовий формат, призначений для людського сприйняття – WebAssembly Text Format (WAT). До огляду пропонується наступний фрагмент WAT коду:

```
(module
    (import "Math" "random" (func $random (result f64)))
    (import "console" "logstr" (func $logstr (param i32 i32)))
    (import "js" "memory" (memory 1))

    (func $randomInt (param $from i32) (param $to i32) (result i32)
        (local $distance i32)
        local.get $to
        local.get $from
        i32.sub
        local.set $distance
        call $random
        local.get $distance
        f64.convert_i32_u
        f64.mul
        i32.trunc_f64_u
        local.get $from
        i32.add
    )

    (func $main
        i32.const 0 ;; offset to string data
        i32.const 25 ;; size of the string
        call $logstr
    )

    (start $main)

    (data (i32.const 0) "Random module initialized")

    (export "randomInt" (func $randomInt))
)
```

Оголошення модуля

Імпорти

Оголошення фнкції

Тіло функції

Старт-функція

Статичні данні

Експорти

Рисунок 1.1 – Фрагмент WAT коду

Хочу зауважити, що зі всіх синтаксисів ассемблер-мов, синтаксис WAT формату, мені виявися дуже легким і сприятливим для розуміння.

Розглянемо кожну із частин окремо.

```
module
```

Рисунок 1.1а – Оголошення модуля

Оголошення модуля – Умовне позначення початку WASM модуля. При трансляції до бінарного варіанту, перетворюється у послідовність «магічних чисел» для визначення модуля.

```
(import "Math" "random" (func $random (result f64)))
```

Рисунок 1.1б – Імпорт

Імпорт – Оголошення залежностей від хост-середовища. В даному випадку, ми потребуємо визначення функції “random” в модулі “Math”, що має мати вигляд функції що повертає одне значення типу “f64”. Також можна помітити назву “$random”. Це позначення дозволяє звертатись до цього об’єкту із середини WASM коду за текстовим індетифікатором “$random”. Це полегшення для рукописного коду, проте зазвичай замість ідентифікаторів, під час трансляції, будуть підставленні числа-ідентифікатори.

```
(func $randomInt (param $from i32) (param $to i32) (result i32)
```

Рисунок 1.1в – Оголошення функції

Оголошення функції – В цьому фрагменті можна побачити оголошення нової функції з ідентифікатором “$randomInt”, що прийматиме в себе два параметри: “$from” типу “i32” та “$to” також із типом “i32”. Повертатиме в той час функція одне значення із типом “i32”. Так само, як і в оголошенні імпортів, іменовані ідентифікатори виступають для полегшення сприйняття людьми. При трансляції до бінарного формату, використовуються упорядковані числа-ідентифікатори.

```
(local $distance i32)
local.get $to
local.get $from
i32.sub
local.set $distance
call $random
local.get $distance
f64.convert_i32_u
f64.mul
i32.trunc_f64_u
local.get $from
i32.add
```

Рисунок 1.1г – Тіло функції

Тіло функції – це оголошення локальних змінних та набір інструкцій, призначених для послідовного виконання. Оголошення змінних відбувається у стрічці “(local $distance i32)”, де ми опціонально можемо зазначити символьний

ідентифікатор. Оголошення локальних змінних завжди передує оголошенню інструкцій.

```
(start $main)
```

Рисунок 1.1д – Старт-функція

Старт-функція – директива що зазначає, яка функція буде виконана відразу після завантаження модулю. В даному випадку функція з ідентифікатором “$main” буде викликана після завантаження модуля

```
(data (i32.const 0) "Random module initialized")
```

Рисунок 1.1ж

Статичні данні – це ділянка WASM файлу, в якій ми можемо записувати будь-які данні. Функціонал аналогічний “.data” сегменту в виконуваних файлах операційних систем. Першим аргументом операнду виступає зміщення початку даних, другим – стрічка даних закодованими в utf-8.

```
(export "randomInt" (func $randomInt))
```

Рисунок 1.1з

Експорт – це оголошення об’єктів, до яких наш модуль надає доступ для хост-середовища. В даному випадку, ми “експортуємо” функцію “$randomInt” до хоста під назвою “randomInt”.

Також хочу зазначити, що даний приклад, і мій розбір базових принципів WebAssembly формату не покривають такі можливості, як оголошення типів, глобальні об’єкти, та WASM таблиці.

Увесь цей час для нас фігурували такі типи як “i32” та “f64”. Ці позначення являють собою типи даних, якими ми можемо оперувати. Стандарт WebAssembly зазначає наступні типи даних:

i32	32 бітне ціле число
i64	64 бітне ціле число
f32	32 бітне дійсне число стандарту IEEE 754
f64	64 бітне дійсне число стандарту IEEE 754
func (…params) -> (return)	Функція від зазначених аргументів до зазначеного результуючого
funcref	Посилання на будь-яку функцію, незалежно від типу її параметрів чи типу результату
externref	Посилання на об’єкт, що не належить до WebAssembly, натомість репрезентує об’єкт із хост-середовища
memory { size }	Зарезервована ділянка пам’яті розміром в “size” сторінок пам’яті[2]
table	Аналог віртуальних таблиць функцій чи “externref” об’єктів.

Таблиця 1.1 – Типи даних WASM стандарту

Варто також зазначити поточну пропозицію, щодо включення системних інтерфейс-типів до WASM стандарту, що має на меті розширити доступні для користувача типи[4].

Повертаючись до тіла функції, ми можемо помітити лише присутність інструкцій пов’язаних із читанням та записом до пам’яті, викликом функцій, та арифметичними операціями. Це весь набір інструкцій доступних для WASM модулів[5]. В цьому і закладається одна із головних особливостей WebAssembly перед іншими форматами виконуваних файлів: безпечність виконання. Жодна інструкція середовища WASM не може нашкодити хост-оточенню, окрім тих можливостей, до яких хост сам надав доступ завдяки імпорт виразам.

Незважаючи на лімітований набір типів даних та інструкцій, стандарту більш ніж вистачає для написання складних систем та алгоритмів. А будь-який об’єкт, що можна представити у лінійній, обмеженій пам’яті, може бути використаний між хост та WASM середовищами.

1.4 Використання у серверному середовищі

Портативність та ізольованість рантайму є значною перевагою для використання в серверному середовищі. Ізоляція дозволяє отримати безпечне для виконання середовище, для багатьох WASM модулів одночасно. “Пісочниця”, в якій оперують модулі, надає можливості менеджменту ресурсів, ізоляції мережевого трафіку та дискового простору, без використання можливостей ізоляції та віртуалізації системних викликів на рівні ядра ОС (що використовує “Docker”) та без гіпервізора систем віртуальних машин. Повний контроль над рантаймом прибирає накладні витрати на системні виклики операційної системи. Це в свою чергу, поєднуючи доволі високу швидкість WASM систем, та відносно низьким використанням оперативної пам’яті, надає дуже ефективне використання ресурсів на серверному залізі. Такий підхід дозволяє одночасно розгортати велику кількість систем одночасно, на одній машині.

Портативність в свою чергу дозволяє будувати динамічні системи, в яких компоненти можуть бути замінені чи доповнені в будь-який момент. Безпека виконання гарантує, що навіть сторонній код (враховуючи можливості, що ваша платформа надає для нього) можна виконувати безпечно.

Станом на другий квартал 2021 року, існує доволі велика кількість незалежних імплементацій WASM рантайму[6]. Існують також комерційні рішення на базі сервісів оркестрації, таких як “kubernetes”[7]. Проте екосистема WebAssembly на сервері, ще в етапі свого зародження. Потрібна ще достатня кількість досліджень. Та попри це, WASM на сервері надає надію стати де-факто системою розгортання застосунків[8].

1.5 Використання у браузерному середовищі

Як зазначалось раніше, WebAssembly розроблявся з оглядом на веб платформу. Включення цієї технології дозволяє існуючій екосистемі веб додатків отримати доступ до екосистем інших мов, та портувати застосунки, раніше не адаптовані до веб платформи.

Основною мовою модернової всесвітньої павутини є JavaScript (JS). Тому головною задачею WebAssembly на веб платформі є взаємодія із існуючими інтерфейсами доступні для JS. В даній конфігурації браузер та JavaScript двигун виступають як хост платформа, що керує виконанням WASM модулів. Маючи попередній приклад WebAssembly коду із ілюстрації 1.1, використання цього модулю в JS контексті буде виглядати наступним чином:

```
const memory = new WebAssembly.Memory({ initial: 1 })

function logstr(strPtr, length) {
    const bytes = new Uint8Array(memory.buffer, strPtr, length)
    const string = new TextDecoder("utf8").decode(bytes)
    console.log(string)
}

const { instance } = await WebAssembly.instantiateStreaming(
    await fetch("/random.wasm"),
    { Math, js: { memory }, console: { logstr } }
)
const { randomInt } = instance.exports

for (let i = 0; i < 10; i++) {
    console.log(randomInt(4, 20))
}
```

Рисунок 1.2 – Взаємодія з WASM модулем із JS

Повернемось до вимог визначених у імпорт виразах нашого WAT файлу:

```
(import "Math" "random" (func $random (result f64)))
(import "console" "logstr" (func $logstr (param i32 i32)))
(import "js" "memory" (memory 1))
```

Рисунок 1.3 – Визначення вимог WASM модуля

Як можемо побачити, наш модуль потребує визначення функції “random” в модулі “Math”, функції “logstr” в модулі “console”, та однієї сторінки пам’яті під ім’ям “memory” у модулі “js”. Якщо ми не задовільнимо ці вимоги, то завантаження

цього модуля не буде можливим. Наступний фрагмент коду демонструє, як можна ініціювати створення WASM модулю із JS скрипту:

```
const memory = new WebAssembly.Memory({ initial: 1 })
const { instance } = await WebAssembly.instantiateStreaming(
    await fetch("/random.wasm"),
    { Math, js: { memory }, console: { logstr } }
)
const { randomInt } = instance.exports
```

Рисунок 1.2а – Створення WASM модулю в JS

Основний виклик робиться до функції “WebAssebmly.instantiateStreaming”, що ініціює компіляцію, та створення екземпляру нашого модуля. Першим аргументом ми передаємо “Response” об’єкт, другим об’єкт імпортів WASM модуля. Як бачимо, ми передаємо до WebAssembly контексту JS об’єкти із функціями “Math.random”, та “console.logstr”. Для чого нам потрібна пам’ять, та реалізація функції “logstr” буде розглянуто пізніше.

Тепер, маючи екземпляр модуля, та функцію “randomInt”, що цей модуль нам надає, можемо перевірити виконання коду завдяки простому виклику:

```
for (let i = 0; i < 10; i++) {
    console.log(randomInt(4, 20))
}
```

Рисунок 1.2б – Виклик функцій із WASM модулю

І отримаємо очікуваний результат:

```
Random module initialized   random.js:6
5                           random.js:16
4                           random.js:16
11                          random.js:16
6                           random.js:16
8                           random.js:16
13                          random.js:16
6                           random.js:16
16                          random.js:16
5                           random.js:16
4                           random.js:16
```

Рисунок 1.4 – Результат виводу функції з WASM модулю

Проте єдиний інтерфейс взаємодії між JS та WASM середовищами є лише числа типів “i32”, “i64”, “f32”, “f64” та об’єктом типу “externref”, які WASM середо-

вище жодним чином не вміє маніпулювати. Що, якщо ми хочемо передати об'єкт чи стрічку між двома середовищами? Для вирішення цього питання, нам будуть потрібні ручні маніпуляції із WASM пам'яттю, на прикладі функції "logstr":

```
function logstr(strPtr, length) {
    const bytes = new Uint8Array(memory.buffer, strPtr, length)
    const string = new TextDecoder("utf8").decode(bytes)
    console.log(string)
}
```

Рисунок 1.2в – Функція опрацювання даних між WASM та JS за допомогою маніпуляції із пам'яттю та чисел-вказівників

Маючи об'єкт пам'яті ("WebAssembly.Memory"), що є спільним між JS та WASM світами, ми можемо створити функцію, що прийматиме вказівник на стрічку в WASM пам'яті, та довжину стрічки, і інтерпретуватимемо байти, як JS стрічку. Тобто для передачі складних об'єктів, ми спочатку зберігаємо його в спільному сегменті пам'яті, та передаємо на нього вказівник. Є і інші методи передачі нетривіальних типів, що базуються на наданні WASM модулю можливостей для маніпулювання JS об'єктами напряму.

Хоч і всі сучасні браузери вже достатньо часу мають підтримку WebAssembly, деякі старі версії браузерів його не підтримують (наприклад Internet Explorer). Для цього компанією "Mozilla" було розроблено стандарт, що передує WASM і є його ідейним прабатьком: "asm.js". "asm.js" – це відносно невелика підмножина мови JavaScript, проте до якого можна зазвичай компілювати проекти написані на компільованих мовах, як наприклад C / C++ / Rust. Ідеєю було те, що браузери, що розпізнають формат "asm.js" можуть пришвидшити виконання коду, адже правила і доступний функціонал JS зменшено в цьому контексті, а браузери, що не підтримують цей формат, можуть виконувати його як звичайний JavaScript сценарій.

1.6 Висновки огляду технології WebAssembly

Технологія WASM дозволяє розширити можливості веб платформи завдяки інтеграції коду із різних мов програмування, використовуючи переваги тієї чи іншої, більш доцільної практики та екосистеми присутньої іншим мовам.

AOT компіляція, ручне керування пам’яттю, зменшений набір інструкцій та компактний бінарний формат дає значний приріст у ефективності та швидкодії коду, порівняно із JIT реалізаціями для JavaScript мови.

Поза веб платформи WASM починає отримувати оберти для світу серверного програмного забезпечення. Ізольованість та портативність дозволяє отримати значний приріст в ефективності і модульності сучасних серверних застосувань.

РОЗДІЛ 2: Розширення можливостей веб платформи

2.1 Нові можливості для веб платформи

Завдяки WebAssembly веб платформа може інтегрувати до себе програмний код розроблений для застосування для інших платформ чи розроблений на інших мовах, відмінних від JS. Це дозволяє багатьом продуктам та компаніям використовувати існуючі рішення для того, щоб отримати доступ до переваг веб платформи.[9]

Наприклад, продукт “Figma”[27] змогли адаптувати ядро додатку написаному на C++ до веб платформи, і тепер один і та сама високоефективна кодова база C++ використовується для роботи як настільного нативного додатку, так і для еквівалентної за функціоналом веб версії

Переваги у швидкості обчислень, дозволяє більш ефективно використовувати веб додатки для високооб’ємних обчислень. Застосунки для машинного навчання, як наприклад бібліотека Tensorflow, мають доступ до кращої утилізації процесорних ресурсів, особливо із стандартизацією та поширенням SIMD інструкцій[10]. Також алгоритми для зменшення розмірів файлів, картинок, кодеки для програвання відео чи аудіо стають легкодоступними використовуючи нині існуючі високоефективні реалізації алгоритмів написаних на C / C++.

WASM формат швидко набирає популярність серед розробників ігор. Поєднуючи можливості “WebGL" та потужність WebAssembly. До веб платформи долучаються як нові ігри, так і портовані старі, як наприклад “Diablo 1”[29]. Сучасні ігрові двигуни, такі як “Unreal Engine”, та “Unity” дозволяють використовувати існуючі проекти з використанням мови програмування C# та C++ для компіляції до WASM формату.

2.2 Адаптація існуючого C коду до веб платформи за допомоги Emscripten

Для цілей дослідження характеристик поведінки WebAssembly було обрано C проект SWI-Prolog (SWIPL). Цей проект реалізує специфікацію логічної мови

програмування “Prolog”. Станом на травень 2021 року, проекту SWIPL вже понад 34 роки. Під час його заснування і подальшої розробки, навіть натяку на всесвітню веб платформу не існувало (Сам концепт всесвітньої павутини був запропонований Тім Бернс-Лі у 1989 році[11]).

Скомпілювати C код до WASM формату не буде достатнім у даному випадку. Як і більшість проектів, SWIPL залежить від стандартної бібліотеки C, та наявності POSIX-подібної операційної системи:, із підтримкою системних викликів, віртуальною пам’яттю, файлами, сокетами, та пристроями. Для емуляції POSIX слою підтримки нам знадобиться інструментарій “Emscripten”.

“Emscripten” – це набір застосувань для компіляції та адаптації проектів для роботи в WASM та “asm.js” середовищах. “Emscripten” надає підтримку функціоналу стандартної бібліотеки та операційної системи, завдяки емуляції подібного функціоналу використовуючи можливості веб платформи.

В нашому випадку, додаток SWIPL скомпільований за допомоги інструментарію “Emscripten” не розрізняє відмінності від свого звичного середовища виконання. Замість роботи на реальному залізі, SWIPL тепер працює в браузерному оточенні, використовуючи функціонал веб платформи. Запустивши звичайну версію SWIPL для настільних комп’ютерів та версію скомпільовану для WASM, ми дійсно можемо побачити схожість обох версій.

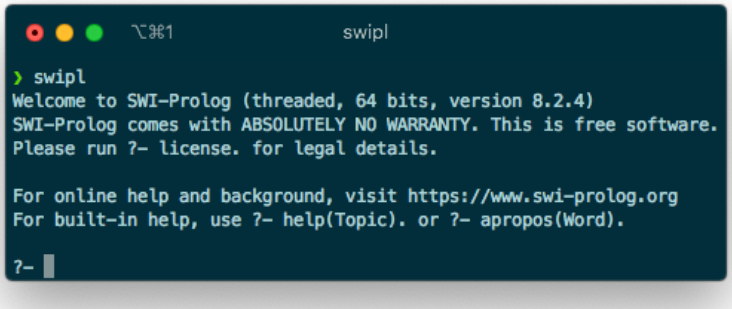


Рисунок 2.1 – Запуск SWIPL під керуванням операційної системи

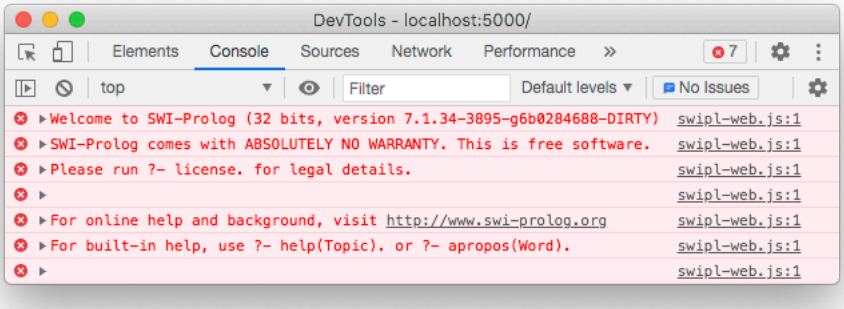


Рисунок 2.2 – Запуск WASM версії SWIPL в браузері

Як можемо помітити, то обидві версії при запуску показують дуже схожі стартові тексти до консолі. Ми використовуємо “console” об’єкт веб платформи замість “stdout” файлу в POSIX-подібних системах, та проте сам SWIPL про це не відомо. Але все таки відмінності є.

Перша відмінність полягає в тому, що WASM версія збудована в 32-бітному режимі. Друга – відсутність підтримки потоків. Хоч і підтримка WASM потоків[12] набирає обертів у різних вендорах, функціонал ще рахується експериментальним, і на час компіляції була відсутня підтримка у жодному браузері. Третя – це версія SWIPL. На відміну від стандартної дистрибуції SWIPL, WASM версія експериментальна, та потребує компіляції розробницької версії.

2.3 Використання C API SWI-Prolog із середовища JS

Для взаємодії SWIPL нам надає C інтерфейс під назвою “Foreign Language Interface” (FLI)[13]. Це набір C функцій, що дозволяють маніпулювати та виконувати Prolog предикати. Документація також описує використання деяких числових значень, визначених як константи в заголовковому файлі “swipl.h”.

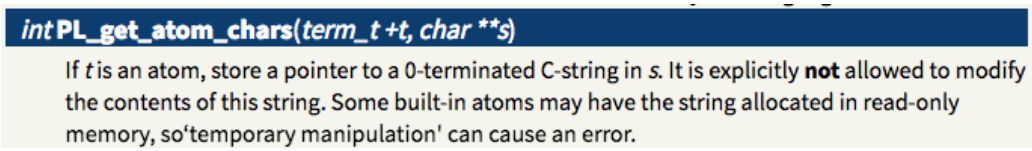


Рисунок 2.3 – Приклад визначеної функції в SWIPL FLI

Як нам уже відомо, інтерфейс взаємодії із WASM модулем доволі обмежений. Ми можемо використовувати здебільшого лише числа для комунікацій. Складніші об’єкти ми маємо передавати через вказівники на спільну пам’ять. Тут нам на допомогу приходять функції із “Emscripten”. “Emscripten” визначає для контексту JS функції керування пам’яттю, такі як: “malloc(size)”, “free(ptr)”, “stackSave()” та “stackRestore(ptr)”. Ці чотири функції нам дозволяють використовувати пам’ять в купі (“malloc” та “free”), та пам’ять на стеку (“stackSave” ти “stackRestore”). На відміну від JS, пам’ять купи не є керованою автоматично, та на відміну від C, навіть змінні стеку автоматично не видаляються. Варто також відзначити відмінний від JS спосіб обробки помилок. SWIPL використовує методику, під назвою “return error codes”, що значно відрізняється від звичного для JS розробників способу “exception throwing”.

“Emscripten” також надає зручні утиліти для роботи із WASM модулем. Функції читання / запису до пам’яті чи емульованої файлової системи; Функції обернення виклику C функцій в їх відповідники у JS, та інші. Більше про використання SWIPL у JS можна дізнатись із моєї попередньої роботи[26].

Проте постачання всього коду для емуляції функціоналу операційних систем займає достатньо великий об’єм. Для розробки нового функціоналу для веб платформи переважно використовувати “чистий” WASM.

2.4 Створення нового коду застосунків за допомоги мови Rust

Rust відносно нова мова програмування, проте розвивається стрімкими темпами. Rust пропонує багато інноваційних аспектів програмування: швидкодію, порівняну із C та C++, відсутність помилок пов’язаних із доступом до пам’яті, без використання збиральника сміття, легка робота із багатопоточністю, без помилок при паралельному виконанні, сучасний набір інструментів, та наостанок, процвітаючу спільноту розробників. Всі гарантії безпеки Rust виконує завдяки системі під назвою “borrow checker”. Його задача: аналізувати час життя об’єктів та посилань на них, та перевіряти їх валідність під час компіляції. Простою мовою: помилки із

некоректним використанням пам’яті будуть знайдені під час компіляції, і код що їх містить не буде скомпільовано.

Rust досягає такої швидкодії та ергономічності використання завдяки інтеграції із LLVM інструментарієм, та використовуючи концепт “безкоштовних абстракцій”. Також варто зазначити зручну систему збірки та керування залежностями – “cargo”.

2.5 Використання Rust для WebAssembly застосувань

Екосистема Rust навколо підтримки WASM формату чи не найкраща в індустрії.

“rustc” (компілятор мови) використовує інструментарій LLVM для генерації коду. LLVM має еталонну підтримку WASM формату, тому більшість мов, що використовують подібну архітектуру компіляції, автоматично отримують базову підтримку WebAssembly. “rustc” також підтримує різні варіанти компіляції, такі як “wasm32-unknown-unknown” (чистий WASM код) та “wasm32-unknown-emscripten” (разом із підтримкою емуляції можливостей операційної системи).

Низькорівневість мови Rust, та відсутність збирача сміття (GC) роблять мову найбільш придатною до застосування в WASM форматі, на одному рівні з C та C++.

Реєстр “crates.io” налічує величезну кількість сумісних із WASM форматом пакетів (також відомих як “crates”). Варто зазначити такі бібліотеки, як:

- “wasm-bindgen”, що дозволяє генерувати зв’язуючий код між платформами;
- “js-sys”, що дозволяє використовувати примітиви та об’єкти зі світу JS;
- “web-sys”, що надає доступ до всіх інтерфейсів взаємодії із веб платформою;
- “stdweb”, що дозволяє інтегрувати JavaScript зв’язуючий код одразу в самому Rust коді;
- “serde-wasm-bindgen”, що дозволяє серіалізувати та десеріалізувати структури Rust до JS об’єктів;

- “wasm-pack”, що дозволяє легко сформувати готовий вихідний JavaScript пакет для використання як для веб, так і для “Node.js” застосунків.

2.6 Висновки можливостей розширення веб платформи

WebAssembly дозволяє використовувати нові можливості не передбачені веб платформою по замовченню. Ми можемо використовувати проекти, що не були початково розраховані для веб платформи, і таким чином отримати найкраще із двох світів: швидкодію та неосяжні можливості додатків, та легкість поширення, портативність, і незалежність веб платформи.

Поращена швидкодія – це несумнівно великий плюс для застосувань, що потребують максимальної ефективності, такі як засоби машинного навчання, обробка аудіо з низькою затримкою, та інтенсивна комп’ютерна графіка.

Для розробки під WASM краще всього використовувати відносно низькорівневі мови, що не мають GC систем, на кшталт C та C++. Мова Rust є доволі сильним вибором для подібних систем через свою низькорівневість, гарантіями безпеки, та екосистемою інструментарію для підтримки WebAssembly.

РОЗДІЛ 3: Порівняння різних моделей виконання

3.1 Визначення методики порівнянь

Для дослідження поведінки різних методик виконання коду на веб платформі, та аналізу можливостей, що надає WASM, було розроблено прогресивний веб додаток (PWA) – “wasm-web-checkers”[14]. Функціонал додатку доволі простий: надати можливість гри у шашки із можливістю прорахунку ходів комп’ютерним гравцем, із вибором різних алгоритмів, кількості прорахунку наперед та вибором імплементації між SWIPL, JS та Rust. Окрім прорахунку стану грального дошки, додаток має прораховувати оптимальні ходи для комп’ютерного гравця. Саме алгоритми пошуку найкращого ходу будуть використані для порівняння швидкодії кожної із реалізацій, тому варто розуміти влаштування цих алгоритмів.

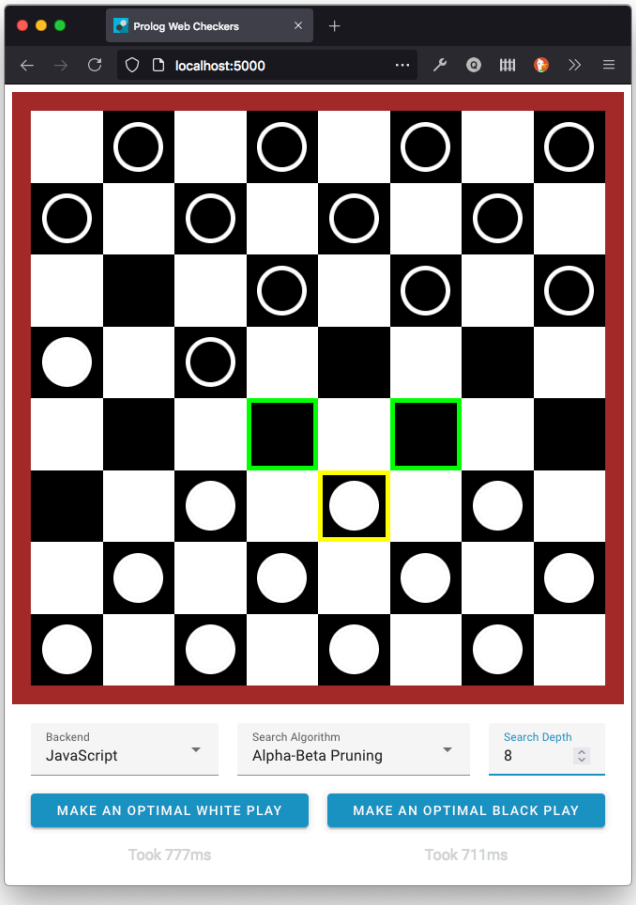


Рисунок 3.1 – Інтерфейс додатку

3.2 Опис роботи алгоритмів

3.2.1 Алгоритм “Minimax”

Алгоритм “Minimax” є типовим для знаходження оптимальних рішень в умовах “ігор із нульовою сумою” (тобто де для того, щоб один гравець переміг, інший має програти).[15]

Гравці діляться на максимізуючого та мінімізуючого. Завдання кожного гравця максимізувати, чи мінімізувати поточну вартість дошки відповідно. Кожній конфігурації дошки надається вартість: позитивне число означає більш вагому позицію максимізуючого, негативне – перевагу мінімізуючого. Від вибору евристики оцінки дошки напряму залежить якість ходів комп’ютерного гравця.

“Minimax” є алгоритмом пошуку в дереві, отже для прорахунку найкращого ходу на N ходів вперед, нам потрібно обчислити кожну можливу пермутацію дошки. Кількість прорахованих рішень зростає експоненціально до N.

Алгоритм працює покроково, вираховуючи найкращий хід одного гравця, в залежності обрахування найкращого ходу суперника, і так рекурсивно до самих листків дерева.

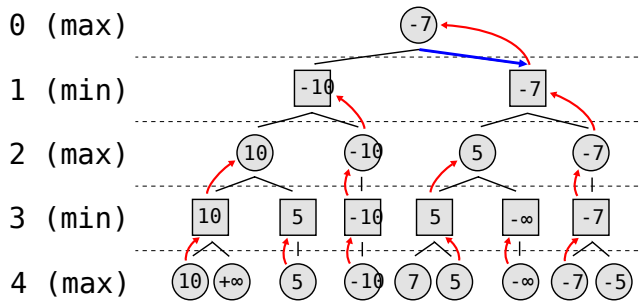


Рисунок 3.2 – Опис роботи алгоритму “Minimax”[15]

3.1.1 Опис роботи алгоритму “Alpha-Beta pruning”

Алгоритм “Alpha-Beta pruning” є покращенням алгоритму “Minimax”. Основна ідея полягає в тому, щоб зменшити кількість обчислених станів гри “обрізаючи” гілки, що однозначно не дадуть кращої відповіді[16].

Це досягається завдяки триманню обліку найкращого ходу для максимізуючого, та найкращого ходу для мінімізуючого гравця. Таким чином якщо гравець вже має кращу альтернативу, обчислену раніше, то йому не матиме сенсу обраховувати гілки, що однозначно не дадуть кращого ходу. Це дозволяє значно зменшити ступінь експоненціального росту складності алгоритму, проте не змінює тенденцію росту його складність вцілому.

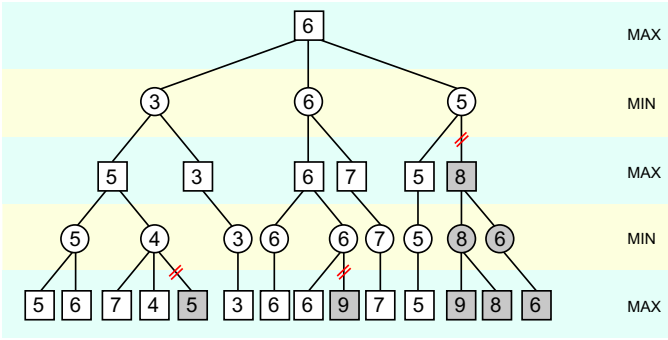


Рисунок 3.3 – Опис роботи алгоритму “Alpha-Beta pruning”[15]

3.3 Особливості моделей виконання

3.3.1 Rust скомпільований до WASM платформи

Rust мова розроблена із урахуванням Ahead-of-time (AOT) компіляції. Rust є представником процедурних, структурованих, імперативних, мов на кшталт C чи C++. Хоч Rust і має запозичення із більш декларативних, функціональних мов родини “Scheme”, модель виконання все рівно лишається імперативною.

Ця модель повністю співпадає реалізації WASM у браузерах. Зазвичай вони використовують більш ефективну AOT компіляцію для отримання найкращої

швидкодії. Це гарантує найкращу можливу утилізацію обчислювальних ресурсів, після процесу компіляції до рідного набору інструкцій процесора.

Для полегшення обчислень у нагоді стають ліниві обчислення можливих станів гри. Для реалізації подібних завдань, у Rust принято використовувати чудовий та швидкодійний механізм ітераторів. Хоча реалізація лінивості завдяки ним потребує трохи більше коду ніж подібні реалізації за допомоги вбудованої лінивості, чи генераторів. Варто також зазначити існування експериментальної версії генераторів у “nightly” версії Rust компілятору.

3.3.2 Prolog під керівництвом SWI-Prolog у WebAssembly

Хоч і сам SWI-Prolog керується точно такою самою моделлю імперативних обчислень за допомоги AOT компіляції, сама мова Prolog, що цей проект реалізує, користується зовсім іншими правилами виконання.

Мова Prolog слідує декларативній, логічній моделі обчислень, більш схожа на автоматичне доведення теорем, ніж на традиційну модель обчислень. Виконання програми слідує визначеним логічним правилам – предикатам. Проте самі предикати використовують інтерпретацію для обчислень. Це надає гнучкості для виконання будь-якого сценарію наданого під час роботи, та проте значно поступається у швидкодії та використанні пам’яті для значних навантажень.

Мова Prolog в своїй натурі повністю лінива. Обчислення списків результатів, а також знаходження всіх розв’язків предикатів не будуть обчислені, допоки вони нам не знадобляться у обчисленнях.

3.3.3 JavaScript у сучасній веб платформі

Схожим чином до Prolog, JS дуже гнучка та динамічна мова. В будь-який момент кожна змінна, чи аргумент може отримати інше значення. В такому динамічному середовищі здається неможливо оптимізувати код до такої ж міри, як у відповідних статично компільованих мов.

Хоч і JavaScript розроблявся для використання в інтерпритаторах, нині JS має перевагу “домашньої платформи” у вигляді Just-In-Time (JIT) компіляторів. Кожен сучасний JavaScript двигун, будь то V8 у Google Chrome чи JavaScriptCore у WebKit Safari, мають імплементацію хоча б одного оптимізуючого компілятору.

Як і сама мова, JIT системи мають бути динамічними, і підвладні змінам. Принцип дії подібних систем є в тому, що перед оптимізацією, інтерпритатор робить спостереження того, як саме використовується та, чи інша ділянка коду. Зібравши достатньо інформації, JIT компілятор оптимізує код, роблячи припущення на те, що ця ділянка буде використовуватись так саме, як і використовувалась до цього. Тобто, якщо ділянка працює лише з числами, JIT компілятор оптимізує її на використання лише чисел, що значно пришвидшує виконання, і трансформує модель обчислень із інтерпретованої, до такої, що є скомпільованою до цільової архітектури процесора. Проте є і застереження, якщо система помітить використання оптимізованої ділянки з іншими типами даних, ділянку буде деоптимізовано знову до рівня інтерпретації. Через це, хоч і JS зазвичай швидка система, цю швидкість доволі легко втратити.

Щодо реалізації лінивих обчислень, стандартом “ECMAScript 6” передбачена зручна можливість створення генераторів для призупинки та продовження обчислень. Працювати із ними дуже зручно, а підтримка зі сторони вендорів дозволяє отримати приріст у швидкодії порівняно із ручною імітацією генераторів.

3.4 Результати порівнянь

Для цілісного аналізу було проведено ґрунтовне дослідження швидкодії наведених алгоритмів[16]. Для виконання тестування було використано три різні, сучасні, лідируючі браузери, та їх двигуни виконання відповідно: “SpiderMonkey” під керівництвом “Firefox 89.0b11”, “V8” під керівництвом “Google Chrome 90.0.4430.212”, та “JavaScriptCore” під керівництвом “Safari 14.0.3”.

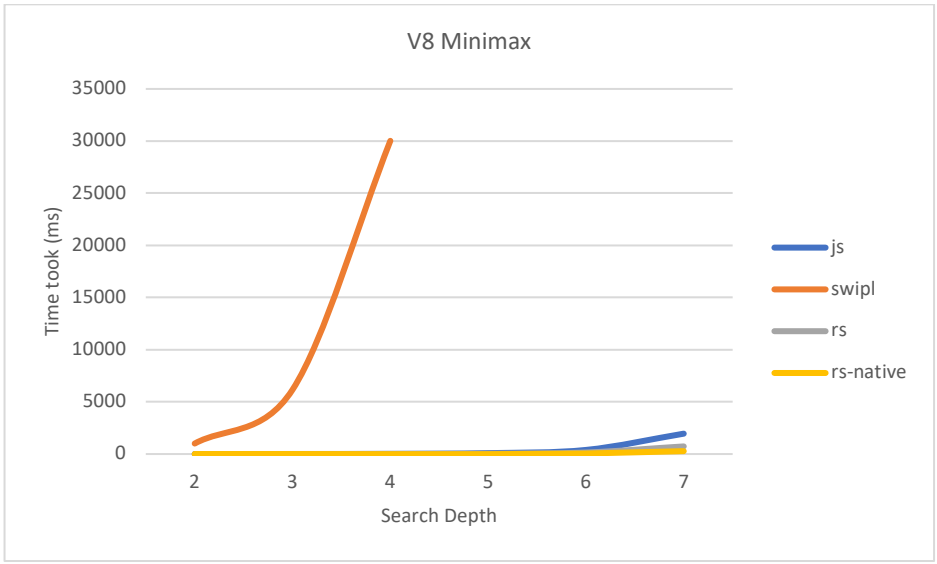


Рисунок 3.4 – Графік залежності часу виконання “minimax” алгоритму від глибини пошуку під керівництвом двигуна V8

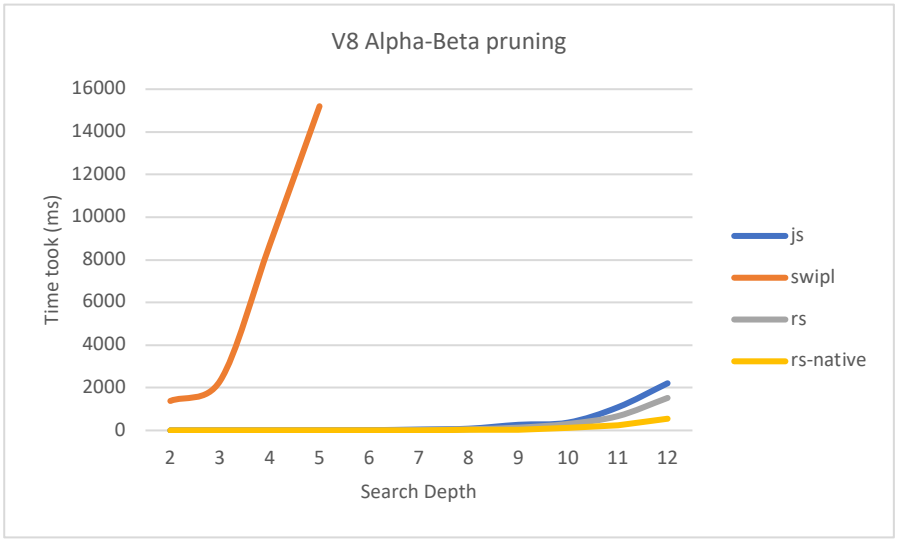


Рисунок 3.4 – Графік залежності часу виконання “alphabeta” алгоритму від глибини пошуку під керівництвом двигуна V8

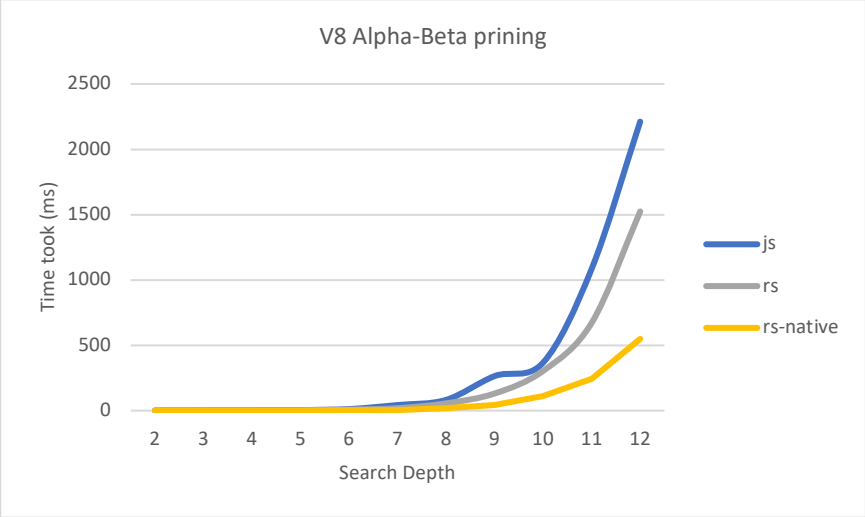


Рисунок 3.4 – Графік залежності часу виконання “alphabeta” алгоритму від глибини пошуку під керівництвом двигуна V8, без урахування SWIPL реалізації

Позначення:

js – Виконання реалізації на JavaScript

rs – Виконання скомпільованої під WASM Rust реалізації

swipl –Скомпільований під WASM SWI-Prolog реалізація

rs-native – Скомпільована до x86-64 Rust реалізація

Додатки А-Ж містять у собі більше інформації, щодо результатів проведених досліджень.

Одразу можемо зазначити один тренд, що вибивається із решти: SWIPL реалізація. Варто зазначити, що оптимізація коду логічної парадигми значно відрізняється від звичного мені процедурного імперативного коду. Тому варто мати на увазі лімітований досвід автора у вирішенні проблем швидкодії мови Prolog.

Інший аспект, що відверто дивує, це наскільки близька швидкодія JS та WASM Rust коду. На графіку 3.4 зображений найкращий випадок, що було зафіксовано у двигуні V8. Інші браузери показують схожі тенденції, проте не настільки близькі.

І останнє, що можна підчерпнути із отриманих даних, те що, хоч і WASM версія застосунку доволі швидка, та щоб досягти швидкодії нативної платформи, ще варто попрацювати. Різні браузери проте ведуть себе по-різному. І варто зазначити їх відносні швидкодії у різних конфігураціях. Більше даних про різні характеристики швидкодії, порівняно із різними браузерами, зазначено у додатку Ж.

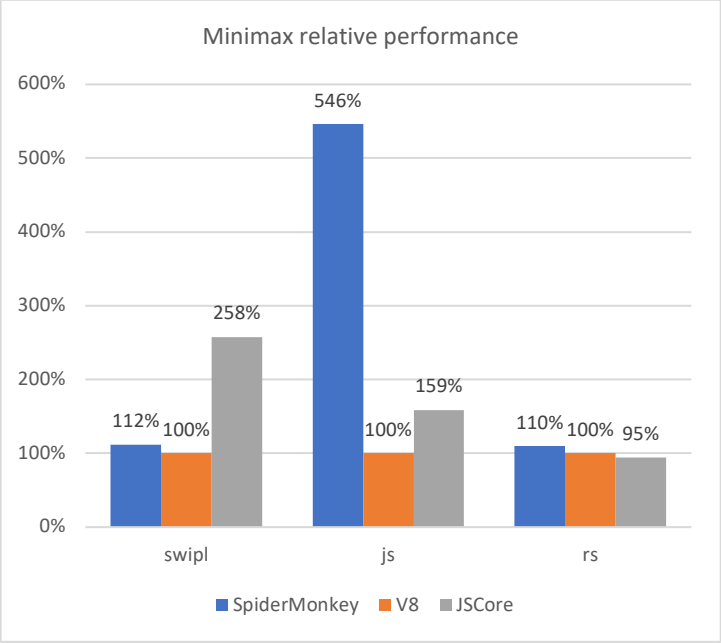


Рисунок 3.5 – Відносне порівняння швидкодії різних двигунів браузерів алгоритму “minimax”. Швидкодія Chrome V8 взята за 100%

Проте у роботі JS середовищі також є особливість. Використовуючи наведені у рисунку 3.6 дані, ми бачимо, що перші декілька інвокацій алгоритму мають значно гірші показники ніж наступні виклики тієї ж самої функції.

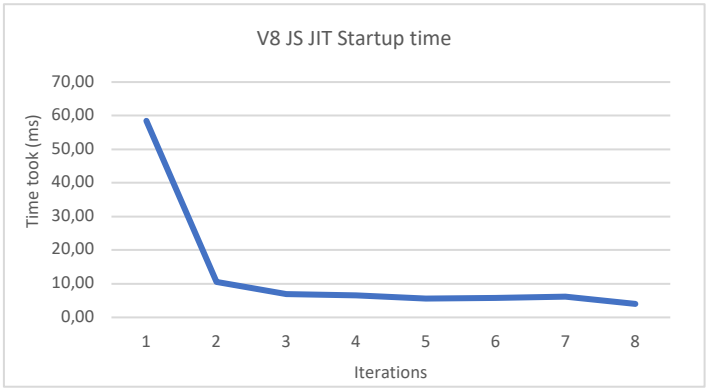


Рисунок 3.6 – Час затрачений на перші виклики алгоритму “minmax” у Chrome V8 використовуючи JS реалізацію

Варто зазначити, що код, що використовує WASM середовище також має невеличкий “час запуску”, проте не такий сильний як при JS виконанні.

3.5 Висновки порівнянь

Маючи данні, та попереднє дослідження аспектів виконання різних платформ можна зробити наступні висновки:

1. Швидкодія Prolog у середовищі WASM далеко не ідеальна. Дійсно, зазначений раніше аспект погіршеної швидкодії інтерпретованих мов себе показав. Не додає надії також модель виконання SWI-Prolog. Хоча і досвіду налагоджування швидкодії Prolog середовищ в автора немає, та проте SWI-Prolog не надає особливих засобів її покращення.
2. За роки вдосконалення, двигуни JS досягли неймовірних висот швидкодії. Новації у сфері оптимізації динамічних мов, та JIT компіляції роблять JavaScript дуже потужною мовою для використання в середовищах підвищеної продуктивності.
3. WebAssembly дає значний, а головне очікуваний приріст у швидкодії. Незважаючи на покращені аспекти швидкості, порівняно із JS, набагато більшу роль дає очікуваність швидкодії. На відміну від JIT компіляції динамічних мов, швидкодія WASM доволі стабільна та передбачувана між різними імплементаціями. До нестабільності JIT компіляції також варто враховувати час на “розгін” двигунів, адже перед тим, як оптимізувати динамічне середовище JS, потрібно зібрати данні про його використання.
4. WASM хоч і швидке середовище, та проте веб платформі ще є куди рости. Включення до стандарту можливостей SIMD[10] та потоків[12] можуть значно наблизити виконання коду на веб платформі до рівнів нативного коду.

РОЗДІЛ 4: Особливості розробки Прогресивних Веб Застосувань (PWA)

4.1 Ознайомлення із додатком

Під час досліджень щодо нових можливостей веб застосувань, увагу прикула технологія WASM, та проте саме об'єднання із сучасними можливостями веб платформи, дає змогу продемонструвати розвиток веб застосувань.

Як і зазначалось раніше, в цілях досліджень цих можливостей було створено додаток "wasm-web-checkers". Та окрім досліджень WASM платформи додаток інкорпорує низку технологій націлених на покращення користувацького досвіду. Розбір саме цих технологій та можливостей буде представлено у цьому розділі.

4.2 Короткий опис архітектури додатку

Додаток в своєму створені інкорпорував достатньо велику кількість мов та парадигм, будучи як частиною дослідженнь WASM платформи, так і задля реалізації функціоналу інтерфейсу. В розробці найбільш визначне місце посідає мова TypeScript (TS). Завдяки неї реалізовано інтерфейс, комунікацію із WASM модулями, та один із трьох доступних "двигунів" обробки ігрового стану. Переваги TS понад JS доволі суттєві. Включення гнучкої та потужної системи типів дозволяє просуватись більш впевнено під час розробки.

Через підтримку багатьох різних "бекендів" для обробки логіки додатку було узагальнено інтерфейс взаємодії між інтерфейсом та важкою для обрахування логікою додатку. Сама логіка працює поза межами головного потоку у веб-воркерах. Це підвищує респонсивність додатку на взаємодію із користувачем, а також дозволяє обірвати довгі синхронні обчислення. Обробка стану та відмалювання зроблено завдяки веб-компонентам

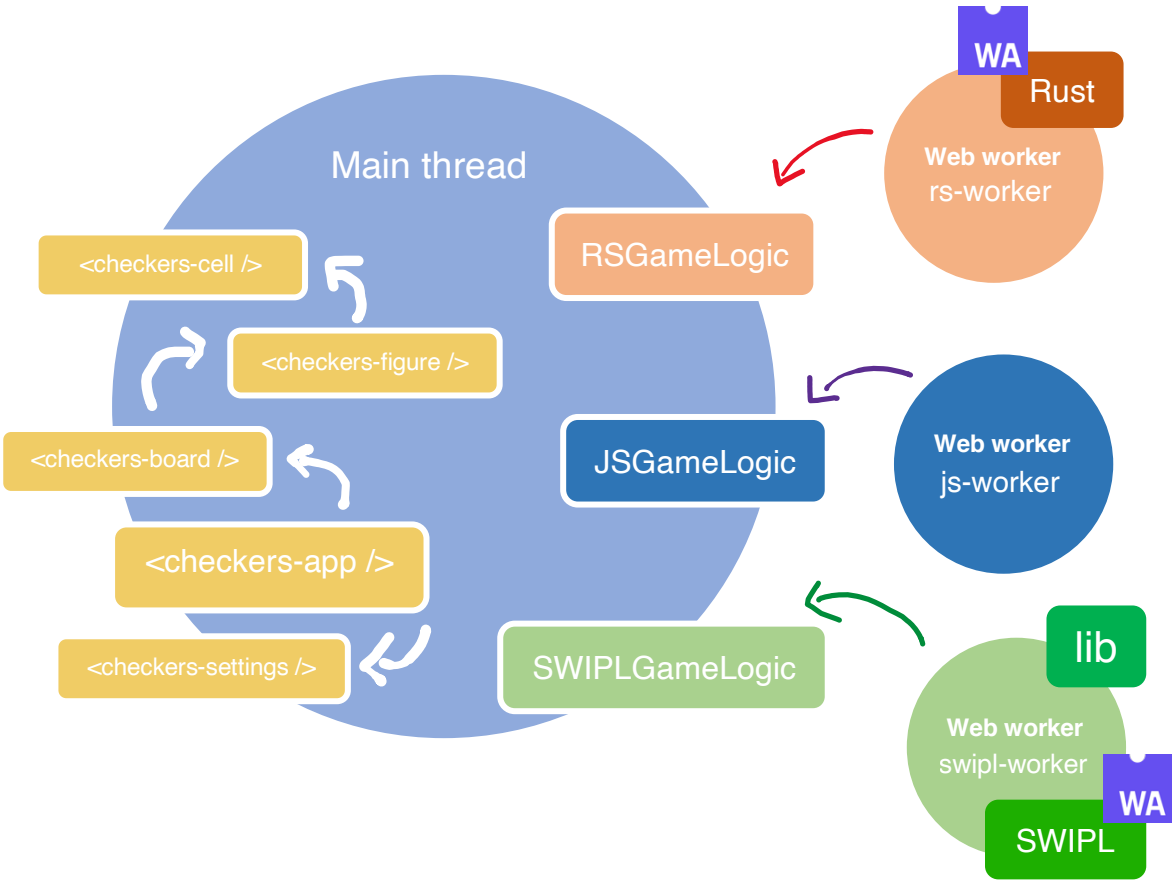


Рисунок 4.1 – Схематичне зображення архітектури

4.2.1 Використання веб компонентів

Для відмальовки інтерфейсу було використано нову можливість веб-платформи – “веб-компоненти”. Тим хто працював із сучасними веб фреймворками це може здатись дещо схожим, та проте існують значні відмінності.

Веб-компоненти – це набір нових можливостей веб платформи, що дозволяє створювати набір HTML опису, JS поведінки, та CSS стилів, що підв’язані до одного нового компоненту і не забруднюють оточення. Веб-компоненти складаються із трьох частин:

1. Ізоляція HTML та CSS правил завдяки “Shadow DOM”[17]
2. Реєстрація та створення нових компонентів завдяки JS[18]
3. Контроль існуючих та можливість створення власних подій

Проте написання логіки керування веб-компонентів потребує доволі достатню кількість імперативних створень розмітки, та прорахунку зміни стану. Для полегшення написання веб-компонентів було використано пакети “lit-html”[19] та “lit-element”[20].

4.2.2 Використання WebWorker-ів

Виходячи із досліджень швидкодії, обробка алгоритму ходу гравця може займати значну кількість часу. Це є проблемою для веб платформи, адже будь-яке синхронне обчислення, що займає більше 16 мілісекунд (у випадку оновлення сторінки 60 разів в секунду), викличе незадовільний досвід користування додатком. JS та веб-платформа оперує в однопоточному режимі виконання. Головний потік виконання займається як обчисленням сценаріїв, так і відмальовкою, та обрахунком користувацьких взаємодій. Будь-яка затримка браузера, змусить користувача чекати на бодай яку-небуть відповідь на його взаємодію (як наприклад натиск на кнопку чи прокрутка сторінки).

Для вирішення цієї проблеми веб-платформа надає “Web Worker”-ів. “Web Worker”, це окремий, незалежний контекст виконання JS коду. Вони працюють в окремому потоці, що робить їх ідеальним засобом для масивних обчислень.

Як було сказано, контексти виконання повністю ізольовані. Єдиний спосіб комунікації між потоками, це передача повідомлень:

```
const worker = new Worker("./worker.js")
worker.addEventListener("message", ev => {
    console.log(ev.data)
})
worker.postMessage("do work")

// worker.js
self.addEventListener("message", ev => {
    if (ev.data === "do work") {
        // do computationally intensive work
        self.postMessage("work done!")
    }
})
```

Рисунок 4.2 – Фрагмент коду взаємодії із WebWorker-ами

Проте часто така модель комунікації не є зручною. Часто виникає проблема, коли нам потрібно кооперувати різні виклики та відповіді. Для вирішення цієї проблеми було використано бібліотеку “comlink”[21]. Завдяки ній можливо використовувати звичну для JS розробників модель асинхронного програмування.

```
import { wrap } from "comlink"

const worker = wrap(new Worker("./worker.js"))
console.log(await worker.doWork())

// worker.js
import { expose } from "comlink"

expose({
    doWork() {
        // do computationally intensive work
        return "work done!"
    }
})
```

Рисунок 4.3 – Фрагмент коду взаємодії із WebWorker-ами, з використанням “comlink”

4.3 Прогресивні Веб Застосунки

Термін “Progressive Web App” (PWA) інкорпорує за собою набір певних технологій для прогресивного покращення користувацького досвіду. Основною задачею цього стандарту є зменшення відчутної різниці між можливим функціоналом веб та нативних застосунків.

За умови використання технологій та виконання умов PWA, можна отримати додаток, що функціонує, і виглядає як нативний застосунок. Його можна встановити, як додаток на робочий стіл. Він надає можливість роботи в офлайн режимі. Отримує доступ до нових можливостей пристрою та операційної системи, таких як завантаження в фоні, можливість отримувати файли із інших додатків, тощо.

Маючи дозвіл на встановлення, додаток отримує, окрім іконки на робочому столі, розширення власних можливостей. Наприклад, браузер дає більші квоти на використання вбудованого сховища даних. Чи дозволяє використовувати додаток,

як ціль отримання файлів та даних із інших додатків встановлених на пристрої користувача[24].

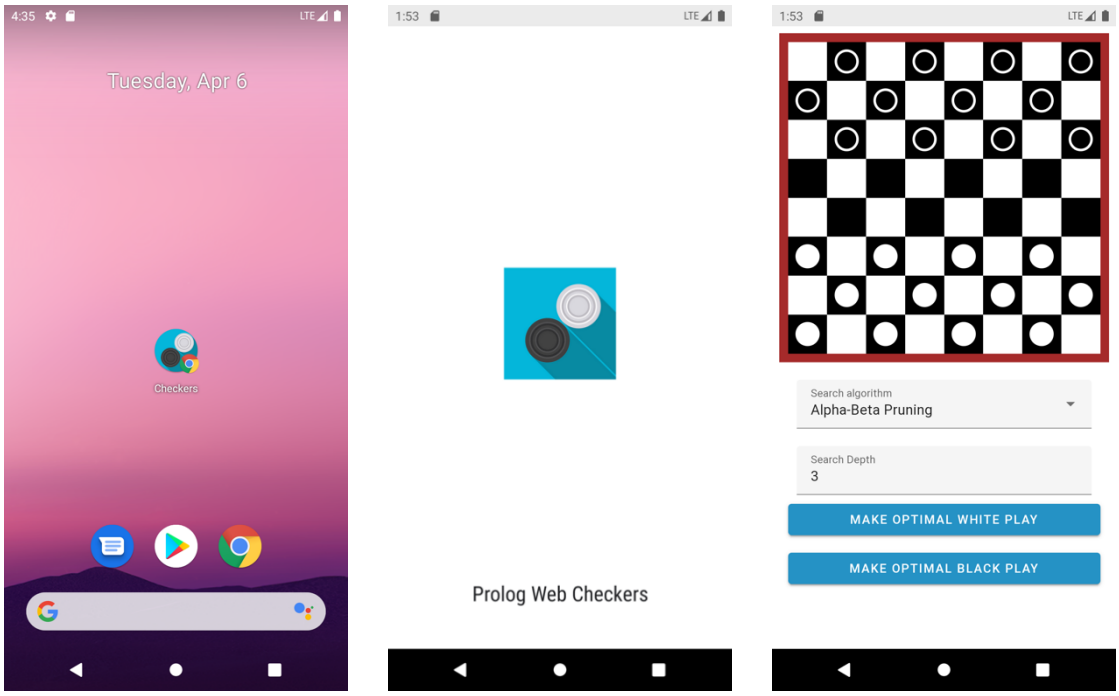


Рисунок 4.4 – Встановлений прогресивний веб застосунок на Android девайс

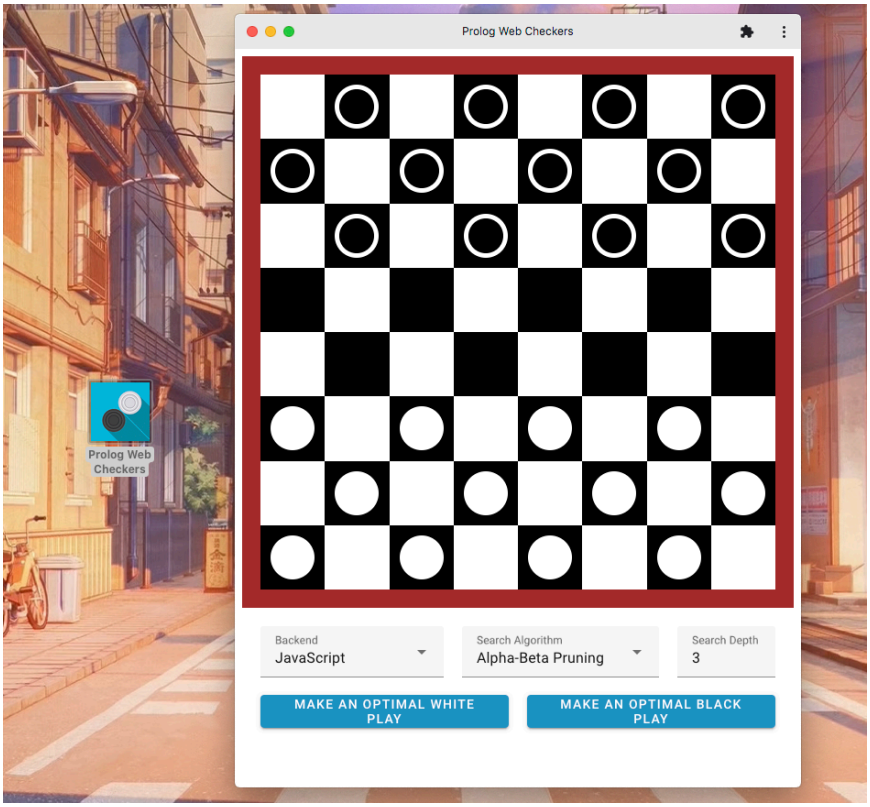


Рисунок 4.5 – Встановлений прогресивний веб застосунок на настільний комп’ютер під керуванням MacOS

4.3.1 Web Manifest

Web Manifest собою представляє JSON документ стандартизованого вигляду[22]. В ньому описано такі властивості як: назва, скорочена назва, режим відображення, колір фону, колір панелі, набір іконок, тощо. Ці параметри дозволяють користувачу інсталювати додаток на власний девайс, надаючи йому параметрів відображення описані, у Web Manifest файлі.

4.3.2 Service Worker

Service Worker[23] (SW) працює аналогічно до раніше описаного “Web Worker”-а. Проте він наділений особливими можливостями. По-перше, Service Worker лише один на всі можливі сторінки веб браузеру. Отже якщо у вас відкрито декілька сторінок одного й того самого додатку, вони мають спільного, єдиного Service Worker-а. По-друге, SW вміє перехоплювати запити підконтрольних сторінок. А по-третє, SW отримує доступ до нових можливостей, що не існують в контексті головного додатку.

Можливість SW контролювати запити сторінки дозволяє робити дуже потужні речі. Поєднуючи із Cache API[25], Service Worker може кешувати файли потрібні для роботи додатку, локально на пристрої. Це дозволяє, як працювати в офлайн режимі, так і модифікувати запити в реальному часі даними із кешу, серверу, та можливо, згенерованими власноруч.

4 Висновки щодо нових можливостей Прогресивних Веб Додатків

Під терміном PWA зібралось багато практик та технологій, що дозволяють додатку відчуватись більш “нативним”. Одна із ключових можливостей – це встановлення на робочий стіл пристрою користувача. Отримавши дозвіл на встановлення, додаток отримує додатково нові можливості, як наприклад, більше доступної пам’яті для збереження власних даних, та можливість отримувати файли і дані від сторонніх додатків.

Найголовнішим інструментом у досягнені покращення користувацького досвіду є Service Worker. Завдяки йому браузер дає нам можливість контролювати сторінку та її запити. Тепер ми можемо оброблювати та зберігати потрібні нам файли для роботи додатку. Таким чином ми отримуємо можливість працювати навіть при відсутності інтернет підключення.

Щодо використання PWA можливостей у розробленому додатку автор детальніше розповідав у його минулій доповіді[26]

Висновки

Технологія WebAssembly є доволі перспективним напрямком розвитку веб платформи. Це новий бінарний стандарт, що дозволяє використовувати код із будь-якої компільованої до WASM мови. Це надає можливість створювати нові види додатків, що раніше не були розраховані для використання у веб браузері, використовувати бібліотеки написані на інших мовах, чи покращити швидкодію та отримати більше контролю над характеристиками виконання коду. Найкраще для використання у WebAssembly підходять мови без рантайму, такі як C, C++ чи Rust.

Прикладом таких застосунків може стати дизайн програмне забезпечення “Figma”[27], додаток для компресії зображень “Squoosh”[28], порт гри “Diablo” до веб платформи[30], сучасні ігри на “Unity”та “Unreal Engine”, тощо. Все це можливості, які раніше було дуже складно додати до веб платформи.

WebAssembly також набирає обертів для використання у серверних середовищах. Завдяки ізольованості та безпеці виконання, WASM модулі легко поширювати та розгортати на серверному залізі.

Розробка рішень для веб платформи із застосуванням WASM щодня стає більш доступною. Портування існуючих додатків завдяки інструментарію “Emscripten” стає доволі легкоздійсненним. Проте в разі розробки нового програмного забезпечення, та адаптації бібліотек, краще скористатись підтримкою сучасних мов компіляції до “чистого” WASM, адже сама бібліотека “Emscripten” додає достатньо багато коду для використання у веб застосунках.

При розробці нових рішень для WASM платформи, вибір автора падає на мову Rust. Вона швидка, зручна, сучасна, має великий обсяг бібліотек та гарну підтримку для WASM платформи. Можна використовувати так само і мови C / C++; інструментарій навколо цих мов також розвивається щодня.

Проте WASM це далеко не єдине вдосконалення веб-платформи, що ми маємо на даний момент. Такі аспекти як веб-компоненти, Web Worker-и та Service

Worker надають додаткових можливостей, раніше недоступних для веб-платформи. У своєму дослідженні я виявив надзвичайну корисність Web Worker-ів, через свою можливість уникати блокування головного потоку важкими синхронними обчисленнями. Варто також приділити увагу Service Worker-ам. Ця нова особливість веб-платформи надає нам можливість контролювати трафік, що надходить із веб сторінок. Цей контроль надає нам незалежність від мережі. Тепер наш додаток може працювати без підключення до інтернету, а також швидко надавати свій функціонал в умовах нестабільного з'єднання. Поєднання цих особливостей, разом із можливістю встановлення на девайс користувача, робить додаток більш схожим на нативний. Користувач отримує такий функціонал і вигляд, що і очікує від додатків на своєму пристрої.

Разом із WebAssembly прогресивні веб застосунки мають значну перевагу над класичними веб додатками. Вони отримують доступ до швидкодії та додаткового інструментарію завдяки WASM, що раніше був доступний лише для розробників нативних додатків. Застосунки тепер працюють у офлайн режимі і мають можливість використовувати новий функціонал платформи. Все це не залишаючи переваги веб-платформи.

Список літератури

[1] WebAssembly – Wikipedia [Електронний ресурс] 9.05.2021.
https://en.wikipedia.org/wiki/WebAssembly

[2] Understanding WebAssembly text format - WebAssembly | MDN [Електронний ресурс] 10.05.2021.
https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format#webassembly_memory

[3] Introduction — WebAssembly 1.1 (Draft 2021-05-13)
[Електронний ресурс] 13.05.2021.
https://webassembly.github.io/spec/core/intro/index.html

[4] interface-types/Explainer.md at master · WebAssembly/interface-types
[Електронний ресурс] 10.05.2021.
https://github.com/WebAssembly/interface-types/blob/master/proposals/interface-types/Explainer.md

[5] Instructions — WebAssembly 1.1 (Draft 2021-05-13)
[Електронний ресурс] 15.05.2021.
https://webassembly.github.io/spec/core/syntax/instructions.html

[6] appcypher/awesome-wasm-runtimes: A list of webassemby runtimes [Електронний ресурс] 10.05.2021.
https://github.com/appcypher/awesome-wasm-runtimes

[7] WebAssembly meets Kubernetes with Krustlet - Microsoft Open Source Blog
[Електронний ресурс] 15.05.2021. Автор Ralph Squillace.
https://cloudblogs.microsoft.com/opensource/2020/04/07/announcing-krustlet-kubernetes-rust-kubelet-webassembly-wasm/

[8] Solomon Hykes: «If WASM+WASI existed in 2008, we wouldn't have needed to created Docker. That's how important it is. Webassembly on the server is the future of computing. A standardized system interface was the missing link. Let's hope WASI is up to the task!» / Twitter
[Електронний ресурс] 10.05.2021. Автор Solomon Hykes.
https://twitter.com/solomonstre/status/1111004913222324225

[9] Made with WebAssembly
[Електронний ресурс] 15.05.2021.
https://madewithwebassembly.com

[10] Fast, parallel applications with WebAssembly SIMD · V8
[Електронний ресурс] 15.05.2021. Автори Deepti Gandluri, Thomas Lively, Ingvar Stepanyan.
https://v8.dev/features/simd

[11] World Wide Web – Wikipedia
[Електронний ресурс] 14.05.2021.
https://en.wikipedia.org/wiki/World_Wide_Web

[12] WebAssembly Specifications
[Електронний ресурс] 14.05.2021.
https://webassembly.github.io/threads/

[13] SWI-Prolog – Manual
[Електронний ресурс] 15.05.2021.
https://www.swi-prolog.org/pldoc/man?section=foreign

[14] Malien/wasm-web-checkers: Trying to run swipl in the browser using wasm, and making it easier to use swipl C FLI from JS/TS. That's all while trying to create a working client-side game of checkers. Now with Rust and JS implementations for comparison
[Електронний ресурс] 14.05.2021. Автор Петрик Ярослав.
https://github.com/Malien/wasm-web-checkers

[15] Minimax – Wikipedia
[Електронний ресурс] 11.05.2021.
https://en.wikipedia.org/wiki/Minimax

[16] wasm-web-checkers/WASM checker benchmark results.xlsx at master · Malien/wasm-web-checkers
[Електронний ресурс] 12.05.2021. Автор Петрик Ярослав.
https://github.com/Malien/wasm-web-checkers/blob/master/WASM%20checker%20benchmark%20results.xlsx

[17] Using shadow DOM - Web Components | MDN
[Електронний ресурс] 15.05.2021.
https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM

[18] Using custom elements - Web Components | MDN
[Електронний ресурс] 15.05.2021.
https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_custom_elements

[19] lit-html [Електронний ресурс] 15.05.2021.
https://lit-html.polymer-project.org

[20] lit-element [Електронний ресурс] 15.05.2021.
https://lit-element.polymer-project.org

[21] comlink [Електронний ресурс] 15.05.2021
https://www.npmjs.com/package/comlink

[22] Web app manifests | MDN [Електронний ресурс] 15.05.2021
https://developer.mozilla.org/en-US/docs/Web/Manifest

[23] Service Worker API - Web APIs | MDN
[Електронний ресурс] 15.05.2021.
https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API

[24] Receiving shared data with the Web Share Target API
[Електронний ресурс] 15.05.2021.
https://web.dev/web-share-target/

[25] Cache - Web APIs | MDN [Електронний ресурс] 15.05.2021.
https://developer.mozilla.org/en-US/docs/Web/API/Cache

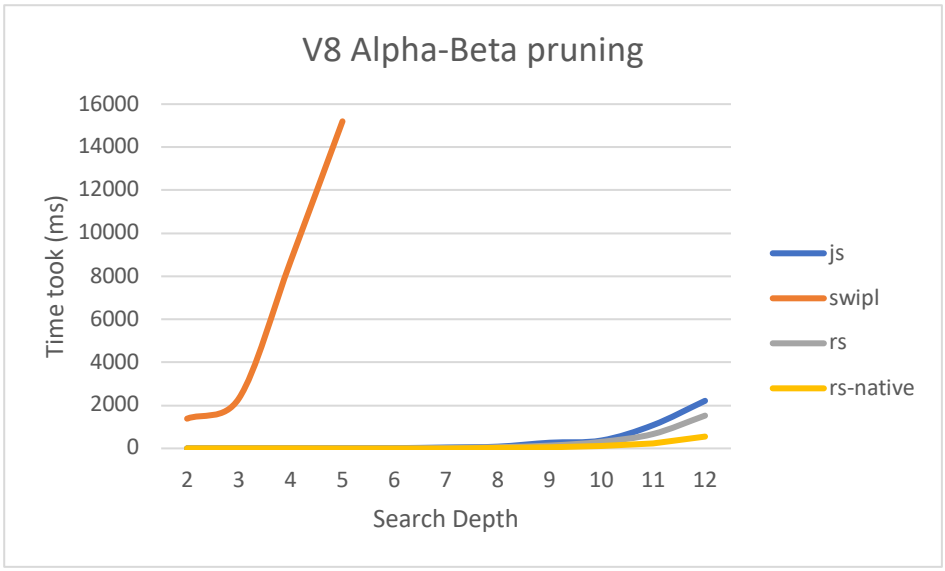
[26] Building PWAs in SWI-Prolog
[Електронний ресурс] 7.04.2021. Автори Петрик Ярослав, Ющенко Юрій.
https://www.youtube.com/watch?v=dF5w-wIoX-A

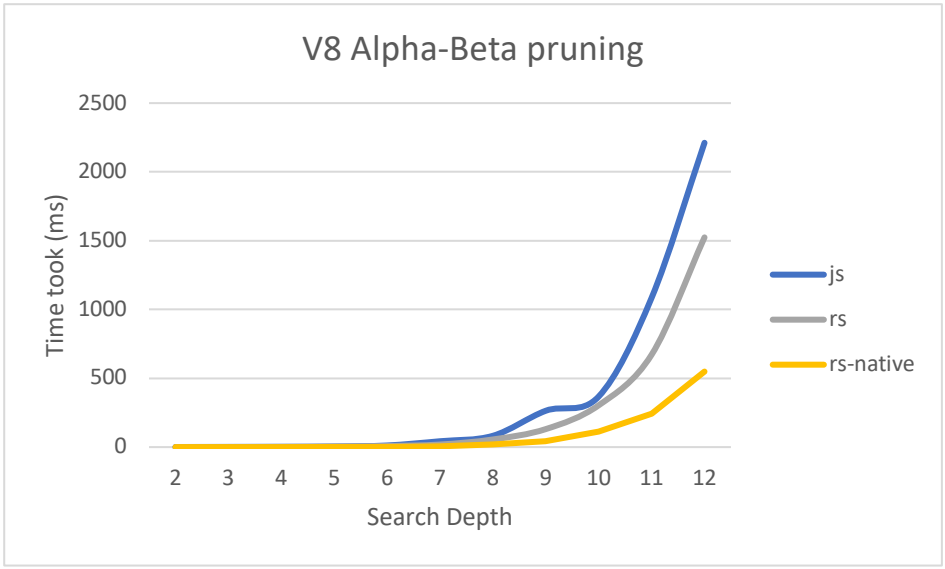
[27] Figma is powered by WebAssembly [Електронний ресурс] 08.06.2017
https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/

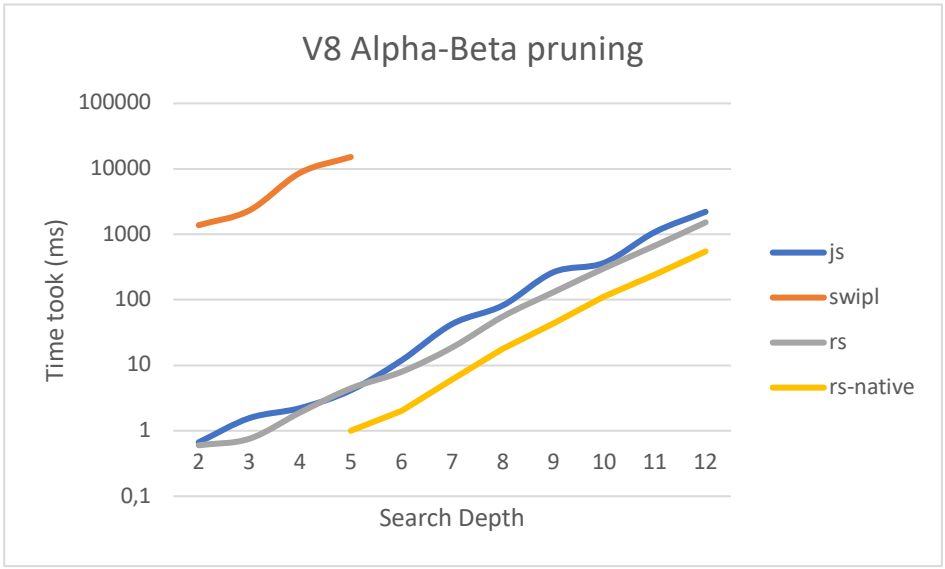
[28] Complex JS-heavy Web Apps, Avoiding the Slow (Chrome Dev Summit 2018)
[Електронний ресурс] 13.10.2018. Автори Mariko Kosaka, Jake Archibald,
Google Chrome Deleopers
https://www.youtube.com/watch?v=ipNW6lJHVEs&t=1s

[29] d07RiV/diabloweb: Diablo 1 for web browsers
[Електронний ресурс] 15.05.2021
https://github.com/d07RiV/diabloweb

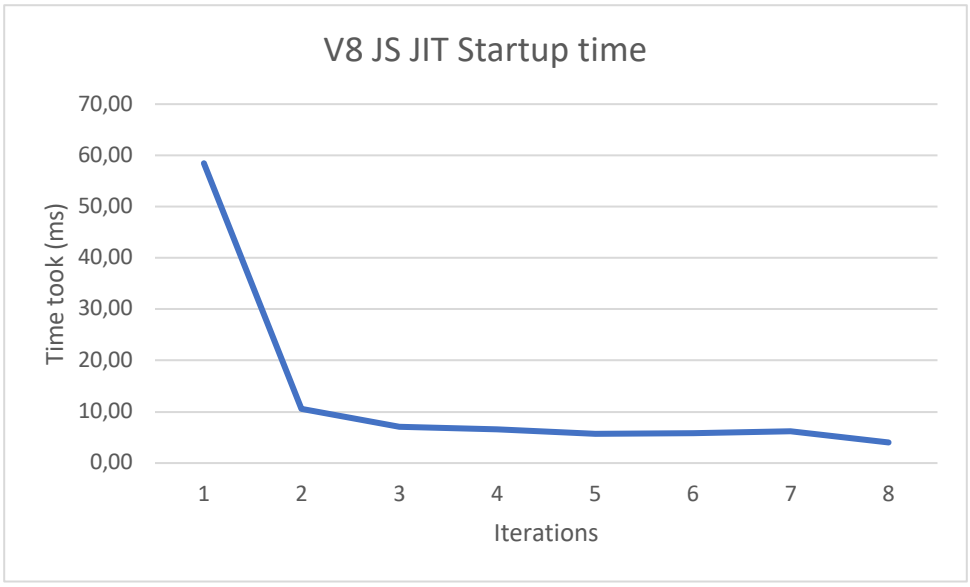
Додаток А: Результати виконання Alpha-Beta pruning алгоритму в середовищі V8

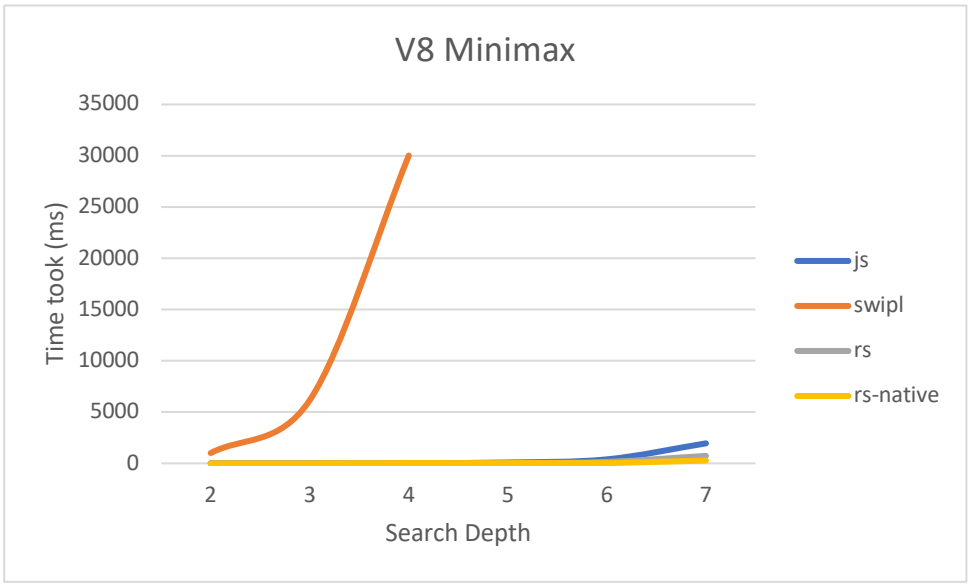


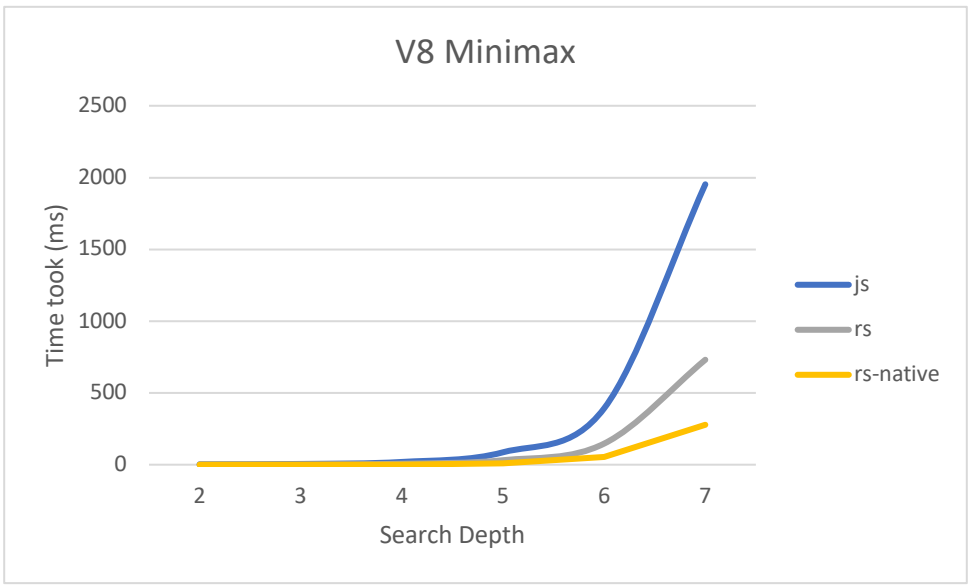




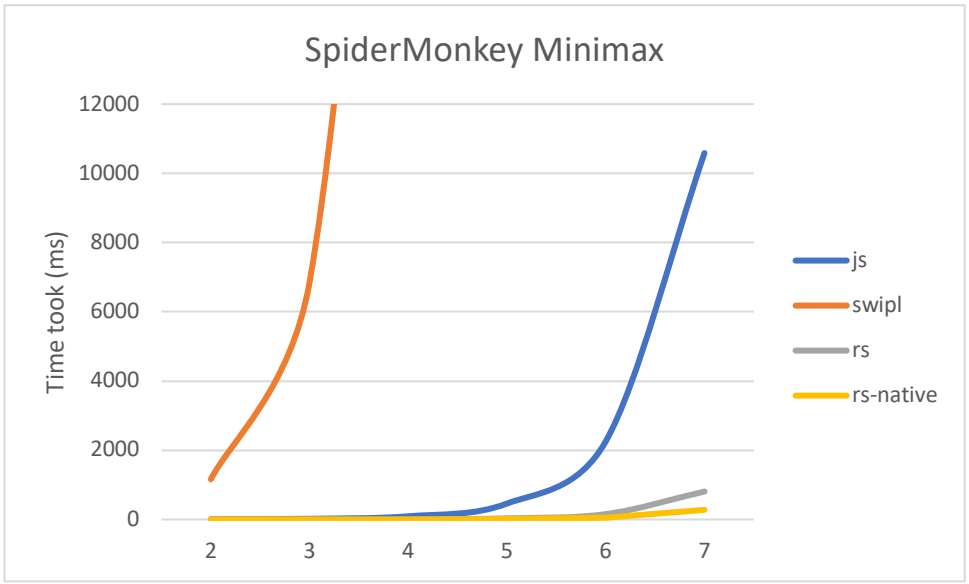
Додаток Б: Результати виконання Minimax алгоритму в середовищі Google V8

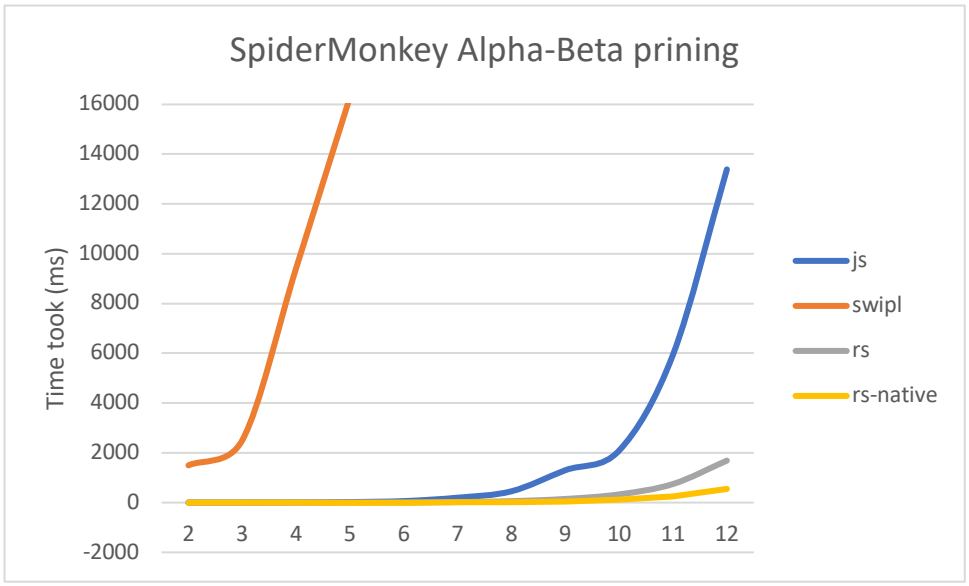




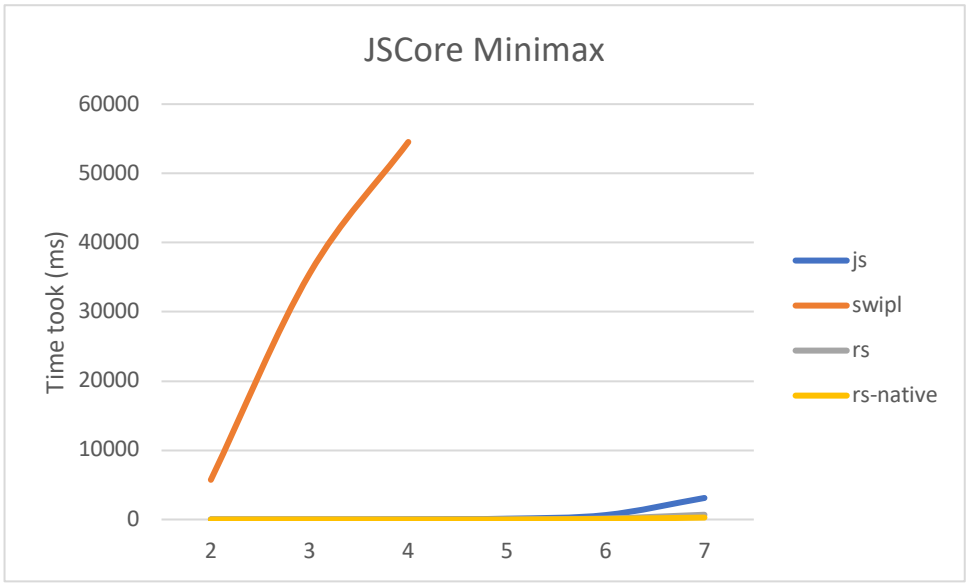


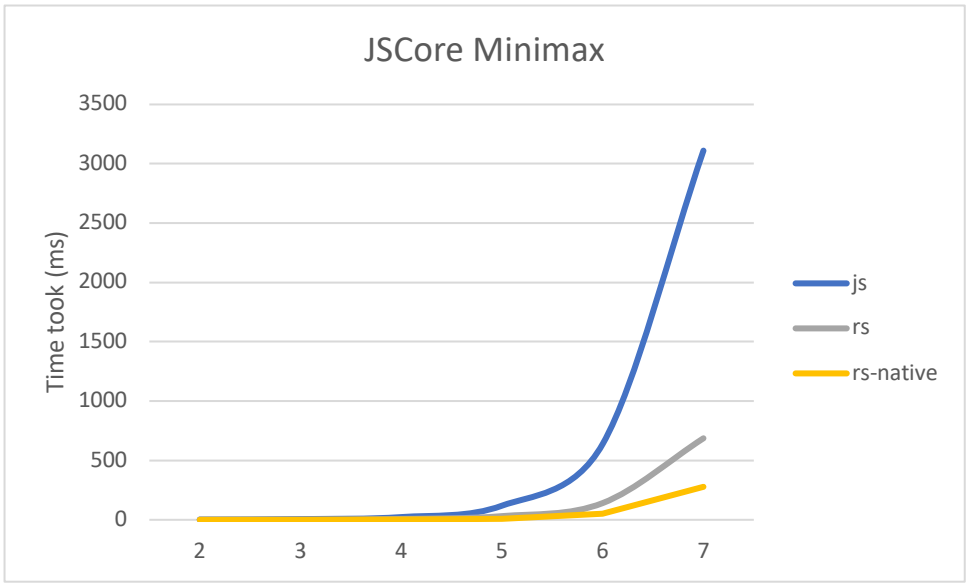
Додаток В: Результати виконання алгоритмів в середовищі Mozilla SpiderMonkey



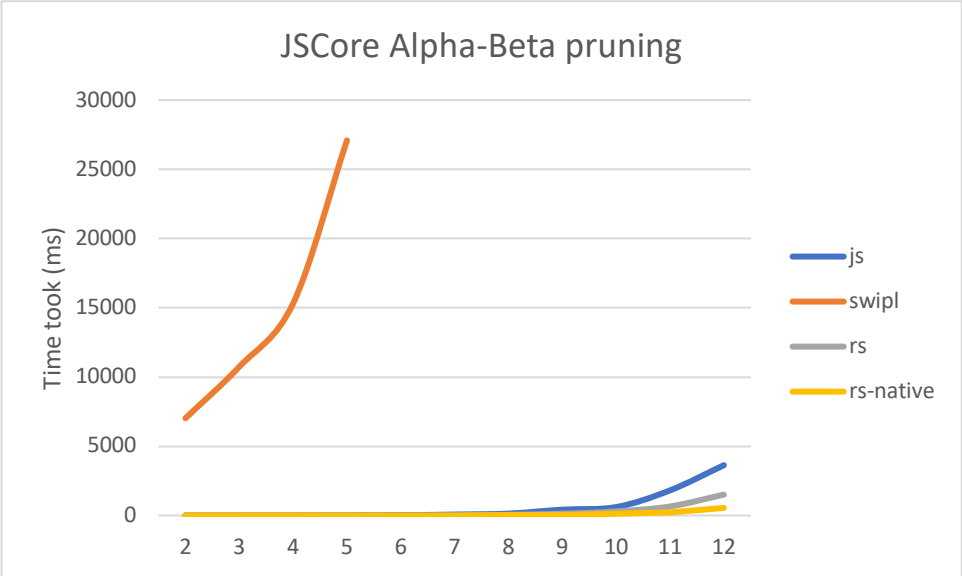


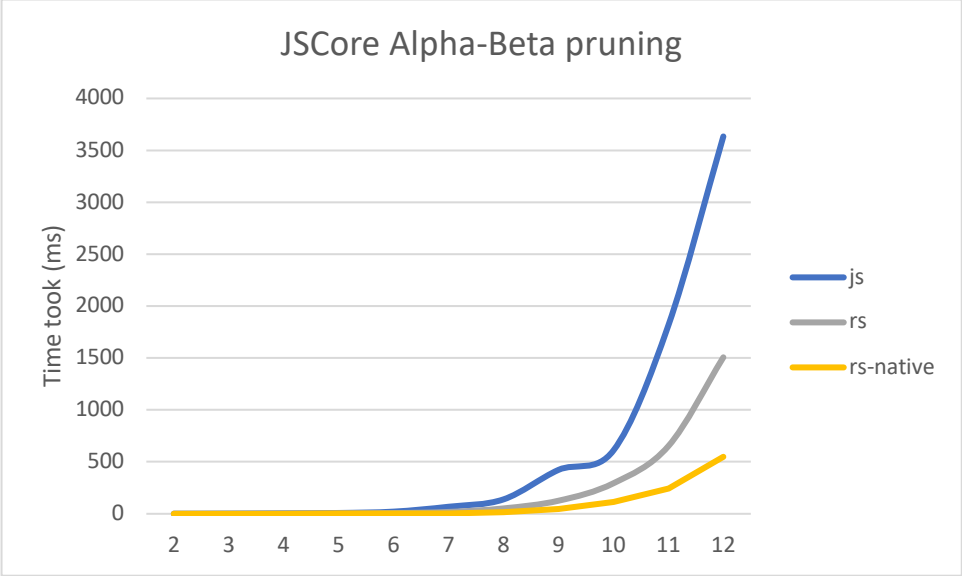
Додаток Г: Результати виконання алгоритму Minimax в середовищі JavaScriptCore



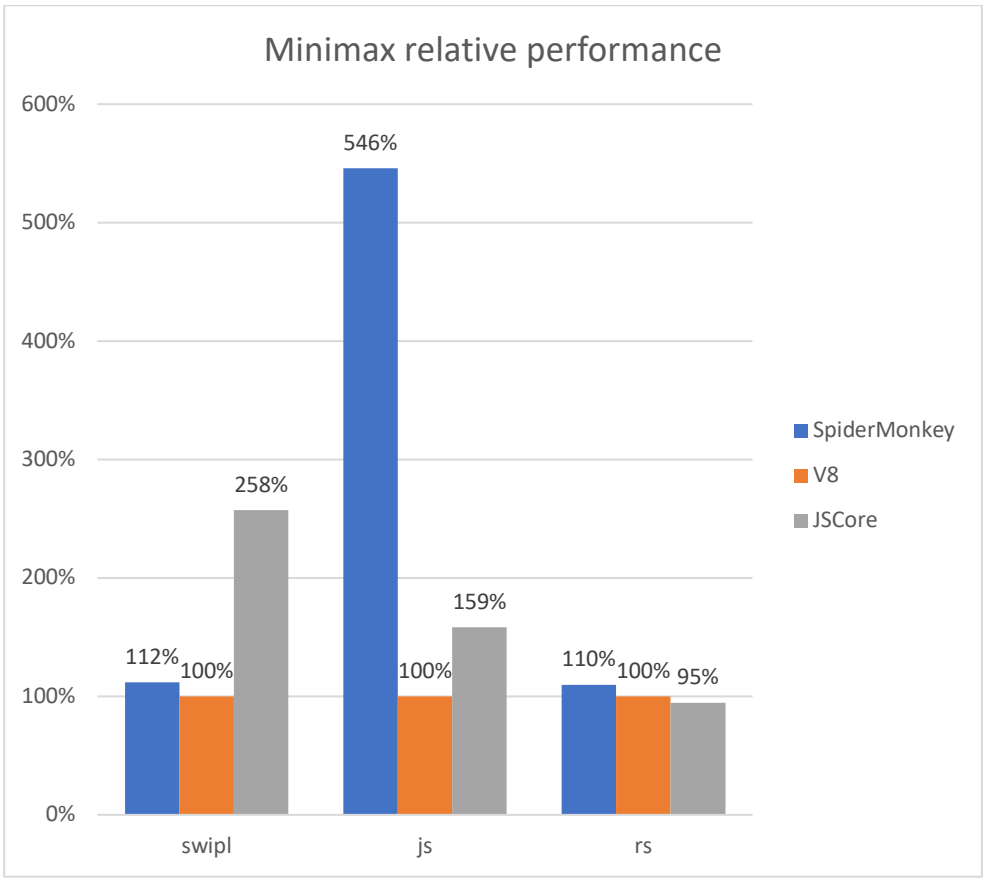


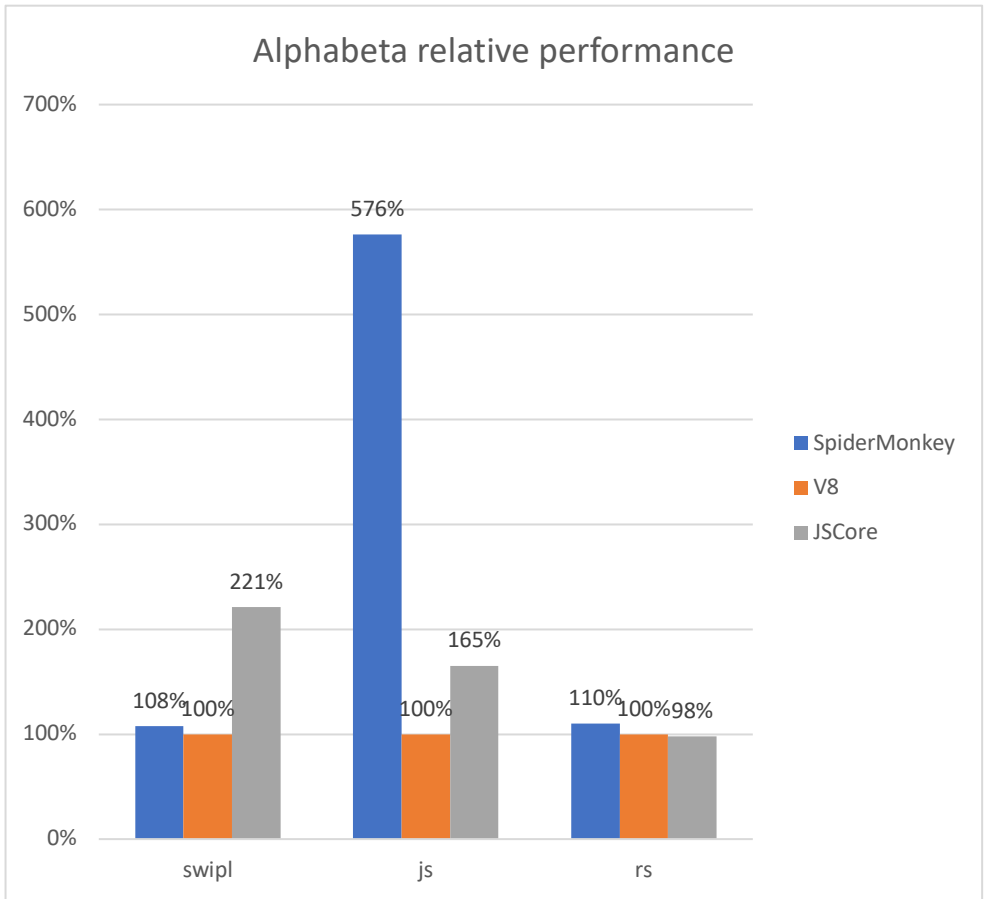
Додаток Ґ: Результати виконання алгоритму Alpha-Beta pruning в середовищі JSCore





Додаток Д: Відносна швидкодія різних браузерних двигунів





Додаток Ж: Порівняння швидкодії брузерних двигунів для різних імплементацій алгоритму Alphabeta