# Звіт до лабораторної 2

Студента групи ТТП-32

Остренка Олександра

## 1. Умова лабораторної:

Розробити та реалізувати алгоритм перевірки: чи допускає скінчений автомат

слова виду w=w0w1 , де w0 – наперед задане (фіксоване) слово, а  w1 – деяке слово, таке

що w допускається скінченим автоматом. На вхід алгоритму подавати w0 .

## 2. Опис виконання:

Ідея виконання: так як скінчений автомат - це структура однозначна, то, отримавши слово w0, можна однозначно перейти від початкового стану до n-ного (n є S, де S - множина станів), таким чином задача зводиться до вирішення наступного питання: Чи можливо дійти від n-ного стану автомату до одного з фінальних станів f (f є F, F - множина фінальних станів)? Щоб відповісти на це питання, автор пропонує скористатися алгоритмом пошуку в ширину " breadth first search algorithm ".

Таким чином, реалізацію лабораторної можна розбити на етапи:

1. *Спочатку треба вирішити, яким чином буде зручніше всього представити автомат, щоб потім було зручно "подорожувати" по ньому, та проводити пошук.*

Автор вирішив представити автомат у вигляді окремого об'єкту ' class FiniteAutomata ', таким чином автомат буде мати наступну структуру:

class FiniteAutomata {

private:

vector<char> inputSymbols; // вхідний алфавіт

vector<int> states; // можливі стани

int initialState; // початковий стан

int currentState; // стан, в якому автомат знаходиться зараз

vector<int> finalStates; // множина фінальних станів

unordered\_map<int, unordered\_map<char, int>> transitionFunction;

public:

bool readAutomataFromFile(const string& filename);

// Method for processing input characters and changing state

bool processInput(char input);

// Function to perform breadth first search

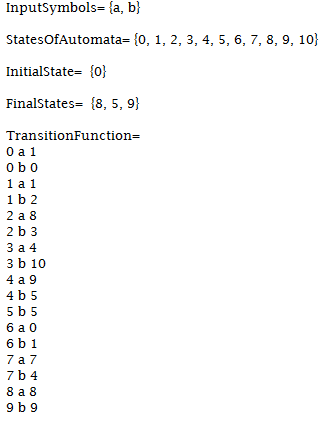
vector<PredecessorInfo> bfsSearch();

int getCurrentState();

};

Єдине, на чому я би зупинився - використання типу unordered\_map в описі функції переходів. Тип " unordered\_map " є контейнерним типом, який зберігає пари ключ-значення, головною його перевагою є легкий пошук значень, що відповідають деякому ключу, а також пошук ключів по відповідним значенням.

1. *Далі треба зрозуміти, яким чином прочитати файл .txt та заповнити автомат.*

Автор ввів наступні правила для опису автомату в текстовому файлі:

Щоб прочитати файл та заповнити автомат, автор використовує функцію

" bool readAutomataFromFile(const string& filename); "

Заповнення " InputSymbols, StatesOfAutomata, InitialState, FinalStates " проходить за одним і тим же алгоритмом:

Знайти ключове слово " key == "InputSymbols=" ", а потім заповнити відповідний вектор

char symbol;

while (iss >> symbol) {

if (isalpha(symbol)) {

this->inputSymbols.push\_back(symbol);

}

}

Так як означення функції переходів виходить за межі одного рядка, а загальна стратегія лежить на тому, що один рядок <--> одне ключове слово <--> множина значень, то алгоритм заповнення функції переходів відрізняється:

Ще до початку зчитування файлу вводимо дві додаткові змінні

" bool insideTransitionFunction = false;

stringstream transitionBuffer;",

перша позначає прапорець "Чи ми почали зчитувати функцію переходів?" , друга ж в свою чергу зберігає усі рядочки в одну стрінгу.

Сам алгоритм працює наступним чином: якщо ми дійшли до визначення функції переходів, тобто " key == "TransitionFunction=" ", то ми ставимо прапорець та чистимо буфер " insideTransitionFunction = true;

transitionBuffer.str(""); "

Далі, поки прапорець буде true, ми записуємо усі рядочки в буфер " transitionBuffer << line << " "; ", а коли файл закінчився " file.peek() == EOF ", ми в циклі читаємо той буфер та заповнюємо функцію перетворень:

while (transitionBuffer >> fromState >> symbol >> toState) {

this->transitionFunction[fromState][symbol] = toState;

}

1. *Коли файл розпарсено, то є сенс розробити функцію, в яку ми будемо закидати символ алфавіту, а вона буде переміщати нас в наступний стан.*

Таку функцію автор назвав " bool processInput(char input); ", вона повертає True, якщо перехід у наступнийстан був успішний, і False - якщо безуспішний.

У цій функції, спочатку, робимо кілька перевірок, а саме: чи є такий символ взагалі в нашому алфавіті, і чи знаходячись у цьому стані, чи є перехід по цьому символу (тобто чи існує така пара в функції переходу). Після перевірки ми змінюємо поточний стан автомату " this->currentState = this->transitionFunction[currentState][input]; ", та виводимо в консоль візуалізацію переходу.

1. *Нарешті останній і найцікавіший етап - "Чи можна дійти від поточного стану до якогось з фінальних?"*

Як було зазначено раніше, автор використав алгоритм пошуку в ширину, для знаходження можливого фінального стану, але цікавіше наступне - як відтворити ланцюжок переходів?

Відповідна функція має назву " vector<PredecessorInfo> bfsSearch(); ", повертає vector.empty(), якщо шляху від n до F не існує, і повертає vector, якщо такий шлях є.

Здебільшого функція реалізована як стандартний пошук в ширину, тобто є вектор станів, які ми пройшли і які виявилися не фінальними " vector<int> visitedStates; ", є черга станів, які ми проходимо, тобто до яких ми маємо доступ завдяки n-ному стану та переходами з нього " queue<int> statesQueue; ", але також є авторська змінна

" unordered\_map<int, PredecessorInfo> predecessors; ", що зберігає усіх "попередників", через яких ми пройшли в пошуках результуючого стану (попередником називаємо трійку: стан, у який перейшли; попередній стан; символ, який прочитали. В даному випадку, ключем є саме стан, в який перейшли). Далі йде знову стандартна реалізація алгоритму: пробігаємось по черзі, перевіряємо кожний стан у ній на "фінальність", після чого дістаємо зі стану усі можливі стани, в які можна через нього попасти

" for (const auto& transition : this->transitionFunction[currentState]) ", перевіряємо, чи є вони серед станів, що ми вже відвідали

" char inputSymbol = transition.first;

int nextState = transition.second;

if (!contains(visitedStates, nextState)) ", відповідно якщо в цьому стані ми ще не були, то закидуємо його в чергу " statesQueue.push(nextState);", а заодно і поповнюємо змінну "попередників" " predecessors[nextState] = { currentState, inputSymbol };".

Після того, як відпрацював алгоритм пошуку в ширину, залишилось відтворити ланцюжок переходів. Для цього ми й використаємо "predecessors", пройшовши "задом-наперед" від фінального стану до того, з якого ми починали пошук

" while (finaleState != startState) {

path.push\_back(predecessors[finaleState]);

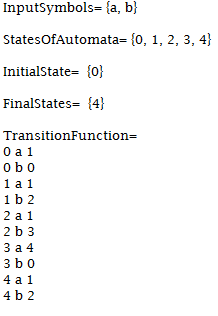
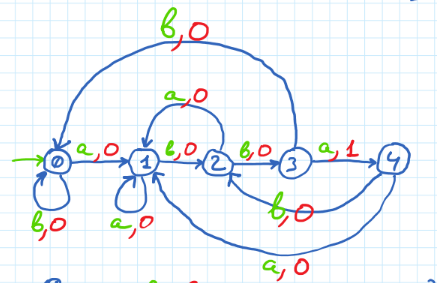
finaleState = predecessors[finaleState].prevState;

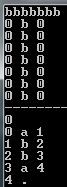
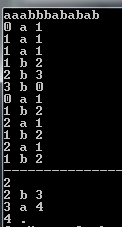
}", далі залишається перевернути ланцюжок " reverse(path.begin(), path.end());" і все, задача виконана.

## 3. Тестування.

Тестування автор проводить на двох різних автоматах.

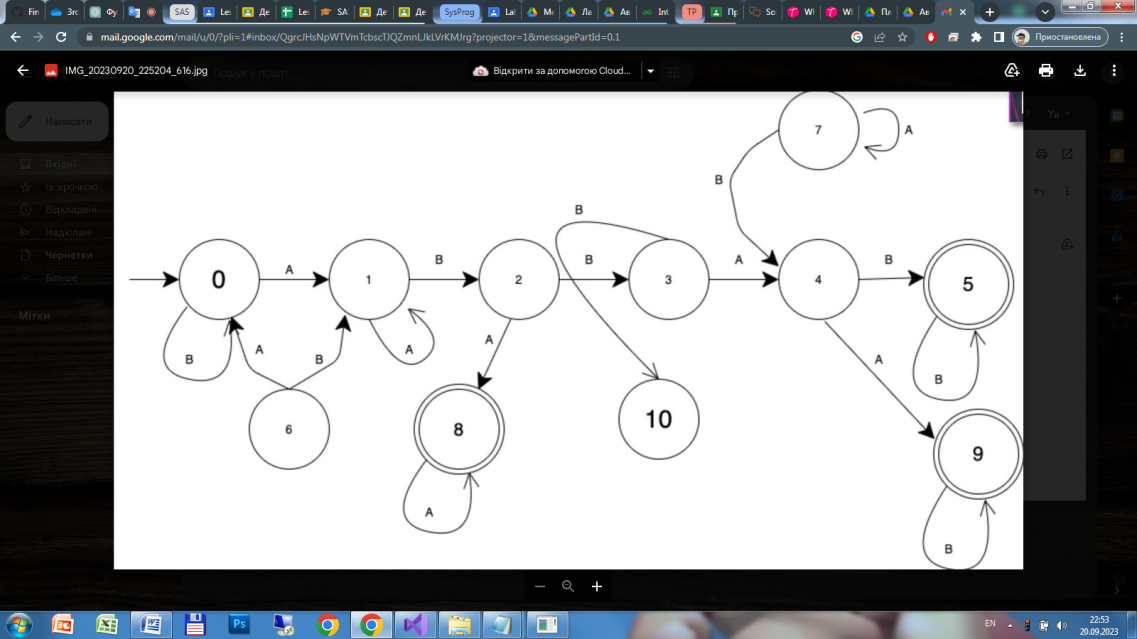
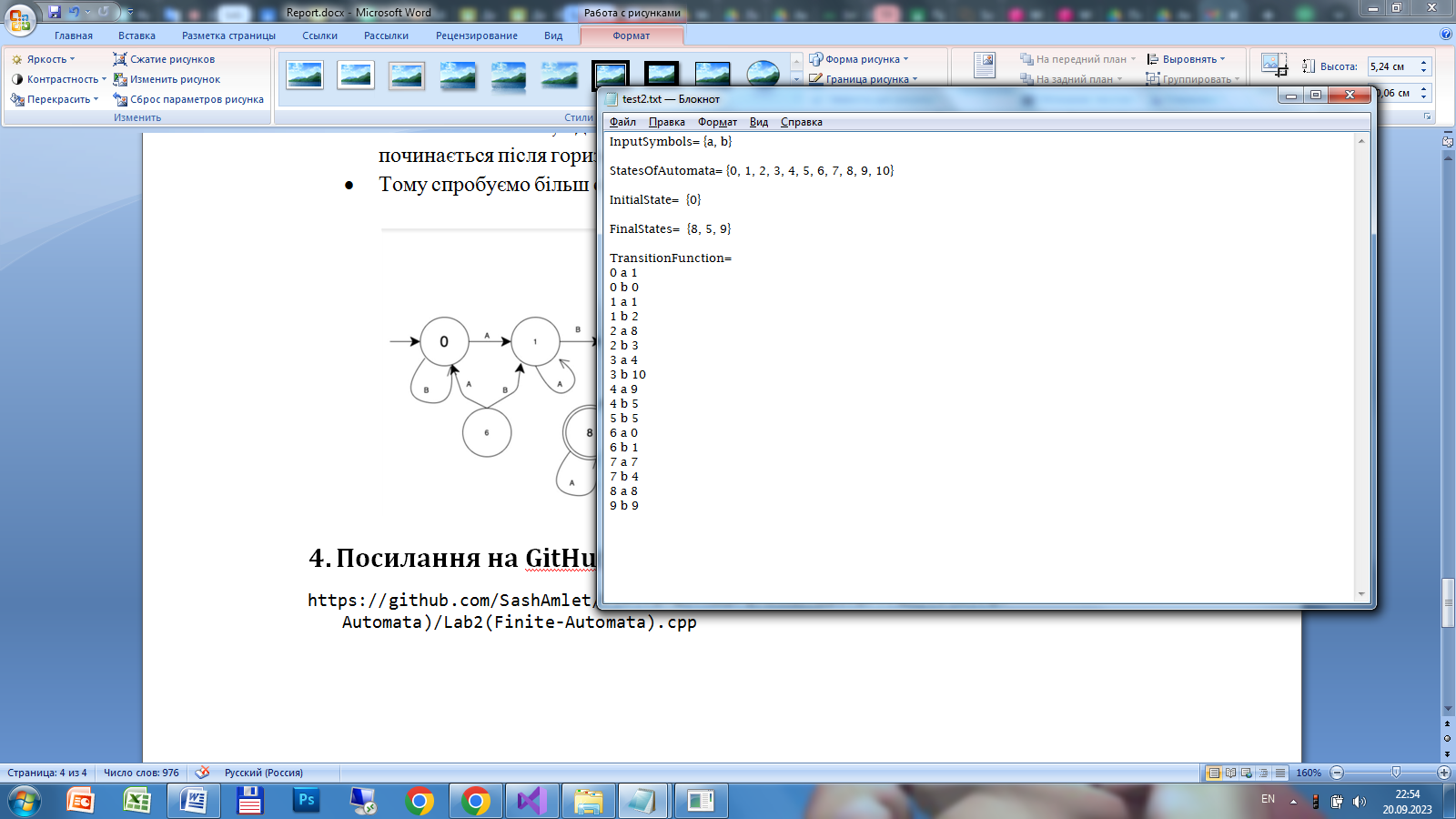
* Спочатку спробуємо на автоматі, який пропускає лише ті слова, що закінчуються на послідовність "abba".

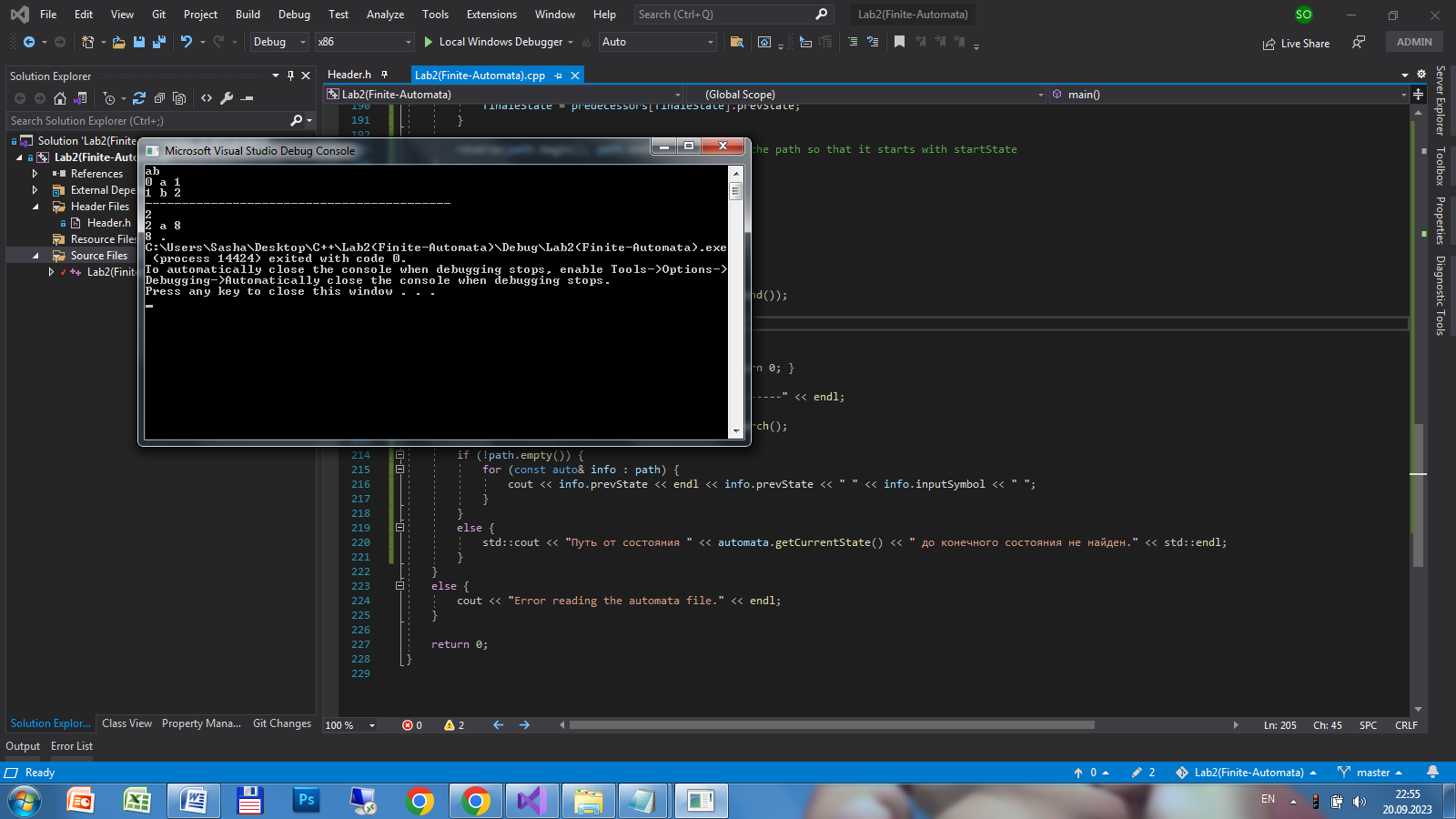
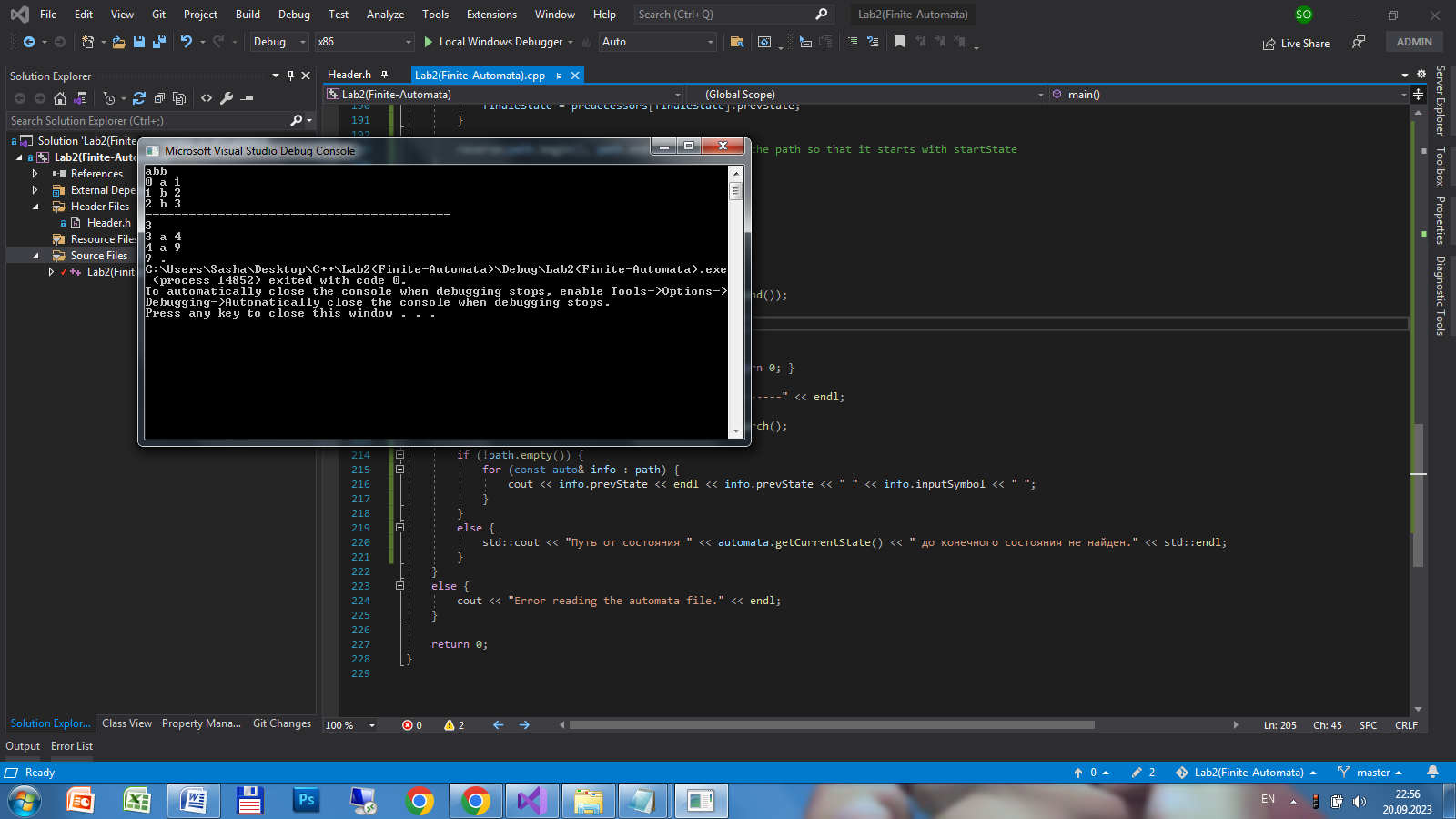
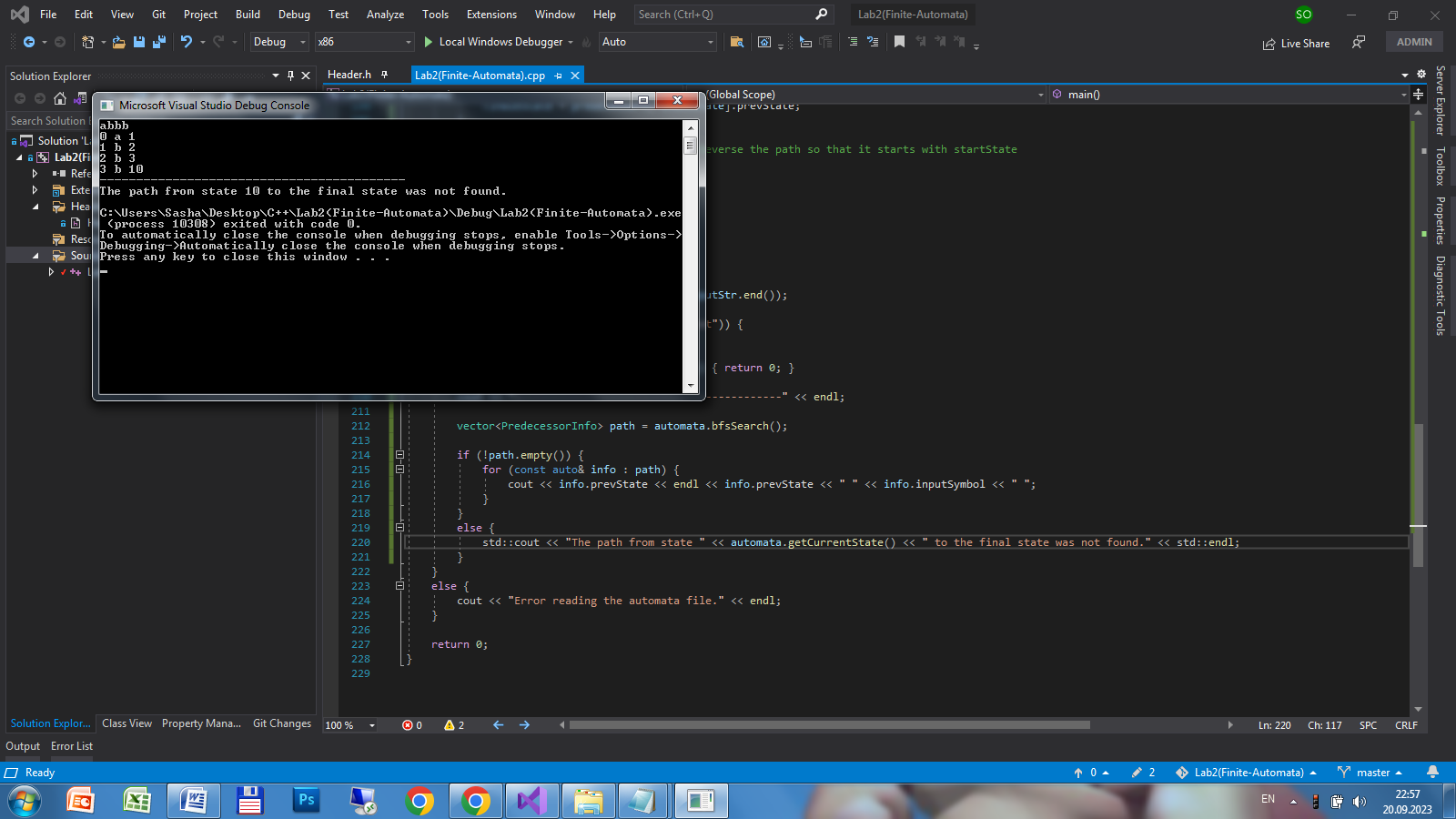




Як можемо бачити, в даному автоматі алгоритм завжди знайде фінальний стан (пошук починається після горизонтальної лінії, усе, що до неї, - перехід по слову w0).

* Тому спробуємо більш складний автомат.

Як можемо бачити, коли вихід є - він його знаходить, а коли немає - то попереджає.

## 4. Посилання на GitHub.

https://github.com/SashAmlet/Finite-Automata/blob/master/Lab2(Finite-Automata)/Lab2(Finite-Automata).cpp