TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

DSA PROJECT REPORT ON 3 × 3 NUMBERS PUZZLE SOLVER

SUBMITTED BY	SUBMITTED TO
Ashim Sapkota (078BEI006) **Batsal Bhusal** (078BEI009) **Birat Thapa** (078BEI012) **Bikash Pokhrel** (078BEI010)	Department of Electronics and Computer Engineering **Signature:** ____________________
Submission Date: 2080-11-27	

TABLE OF CONTENTS

ABSTRACT

The project report presents an implementation of pa solver for the classic 3 × 3 Numbers sliding puzzle using the Breadth-First Search (BFS) algorithm in C++. The sliding puzzle consists of a 3 × 3 grid with eight numbered tiles and one empty space, allowing tiles to be moved horizontally or vertically into the empty space. The objective of the project is to implement an algorithm that can find the shortest sequence of moves to solve any given initial configuration of the puzzle. The report discusses the design and implementation of the solver, focusing on the BFS algorithm's application, the data structures used, the approach taken to solve the puzzle and the handling of various edge cases. Furthermore, it analyzes the time and space complexity of the proposed solution, presents experimental results demonstrating the solver's performance on a range of initial configurations and discusses potential improvements and future work.

ACKNOWLEDGEMENTS

First of all, We would like to express our gratitude to our teacher, Prakash Chandra Prasad, for his constant support and guidance throughout the semester. The lectures, assignments and guidelines have been of great help for us to successfully complete our project. We are also grateful to our lab teachers, Anku Jaiswal and Bibha Sthapit, for their assistance and feedback throughout the practical sessions.

We would also like to thank Institute of Engineering, Pulchowk campus and Department of Electronics and Computer Engineering for providing us an opportunity to work on this project. Our friends and seniors have also been incredibly helpful in figuring out this whole ”teamwork for a project” scenario by sharing their knowledge and critique, this helped in shaping the outline and rectifying the elements of our project. We feel immensely thankful to be surrounded by such supportive, kind and obliging people. Our family has also played a huge role by providing us the support we needed-both emotional and mental-without which finishing the project would have been impossible. We would like to sincerely thank everyone who has directly or indirectly contributed to the project.

INTRODUCTION

Overview

The 3 * 3 Numbers Puzzle, also known as the 8-puzzle, is a classic sliding puzzle that has fascinated puzzle enthusiasts for decades. The puzzle consists of a 3 * 3 grid with eight numbered tiles and one empty space, allowing tiles to be moved horizontally or vertically into the empty space. The objective of the puzzle is to arrange the tiles in numerical order, with the empty space in the bottom-right corner, by sliding the tiles into the empty space.

Objectives

The project aims to develop a solver for the 3 * 3 Numbers Puzzle using the Breadth-First Search (BFS) algorithm in C++. The BFS algorithm is chosen for its ability to find the shortest path to the solution, ensuring that the solver can solve any given initial configuration of the puzzle. The solver will not only provide the solution but also visualize the steps taken to reach the solution, enhancing the user's understanding of the algorithm's workings.

This project is undertaken as part of the Data Structure and Algorithm course, with the objective of applying the concepts learned in class to a practical problem. By implementing the BFS algorithm for the 3 * 3 Numbers Puzzle solver, the project aims to deepen the understanding of graph traversal algorithms and their applications in solving real-world problems.

Solvability Of 3 × 3 Puzzle

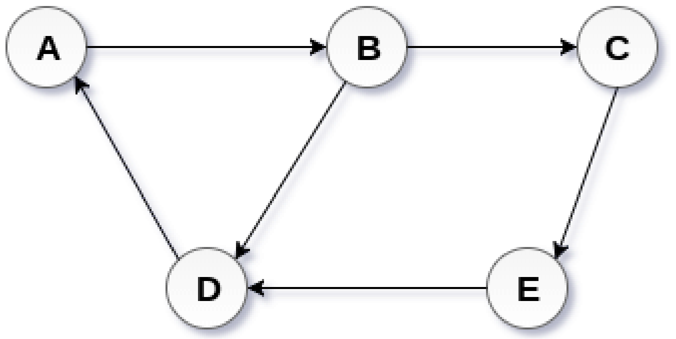
The solvability of a sliding puzzle, including the 3 × 3 Numbers Puzzle, is determined by analyzing the permutation of tiles in the initial configuration. For an odd-ordered puzzle, such as the 3 × 3 variant, the solvability condition is based on the concept of the inversion number. The inversion number is the count of pairs of tiles in the wrong order, considering the goal state. If the inversion number is even, the puzzle is solvable; otherwise, it is unsolvable. This property arises from the fact that in an odd-ordered puzzle, certain permutations of tiles are unreachable through legal moves, which involve sliding a tile into the vacant position. By checking the inversion number of the initial configuration, the solver can determine whether a solution exists before attempting to find it, thereby optimizing the solving process and avoiding unnecessary computations for unsolvable puzzles.

Vectors

The vector is a fundamental data structure in the C++ Standard Template Library (STL). It is a dynamic array that can change its size at runtime, providing efficient storage and manipulation of sequential data elements. Vectors offer several advantages over traditional arrays, including automatic memory management, dynamic resizing, and a rich set of member functions for common operations. They provide constant-time random access to their elements, making it efficient to retrieve or modify the value of a tile at a specific position. They also offer various member functions for common operations, such as push_back for adding elements, pop_back for removing elements, and size for retrieving the number of elements.

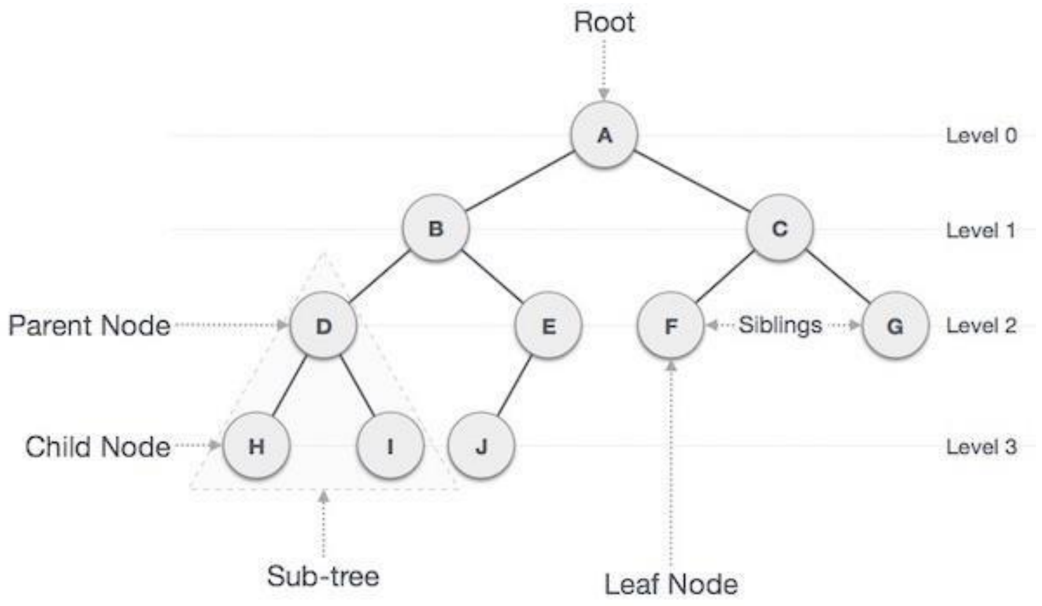
Graphs

Graph data structures are widely used to represent and model pairwise relationships between objects. A graph consists of a finite set of vertices or nodes and a collection of edges that connect pairs of vertices. Graphs can be classified as undirected, where the edges do not have a specific direction, or directed, where the edges have an associated direction from one vertex to another. Graphs can be implemented using various representations, such as adjacency matrices, adjacency lists, or edge lists. The choice of representation depends on factors like graph density, memory constraints, and the specific operations to be performed. Graphs find applications in numerous domains, including network analysis, path planning, circuit design, and social network analysis, making them an essential concept in computer science and algorithm design.



Tree

Trees are hierarchical data structures that represent hierarchical relationships between elements. A tree consists of nodes connected by edges, with one distinguished node called the root. Each node can have zero or more child nodes, and each child node has exactly one parent node, except for the root, which has no parent. Trees are recursive data structures, meaning that each subtree is itself a tree. Trees can be classified into various types, such as binary trees, where each node has at most two children, or general trees, where nodes can have an arbitrary number of children. Trees are commonly implemented using node objects containing data and pointers to child nodes, with efficient operations like insertion, deletion, and traversal. The time complexity of these operations depends on the specific tree type and the balancing properties of the tree. Trees are widely used in computer science for representing hierarchical data, such as file systems, XML/HTML documents, decision trees, and expression trees in compilers.



Breadth First Search Traversal

Breadth-First Search (BFS) is an algorithm for traversing or searching a graph data structure. It explores all vertices at the current depth level before moving on to vertices at the next depth level. The algorithm starts at a given source vertex and systematically visits all the neighboring vertices before visiting any of their neighbors. This approach ensures that the shortest path from the source to any other vertex is found, provided that no edges have negative weights. The time complexity of the BFS algorithm is O(V+E), where V is the number of vertices and E is the number of edges in the graph. This is because, in the worst case, all vertices and edges need to be processed. The space complexity is O(V), as the algorithm requires a queue to store vertices and a visited set or array to track visited vertices.

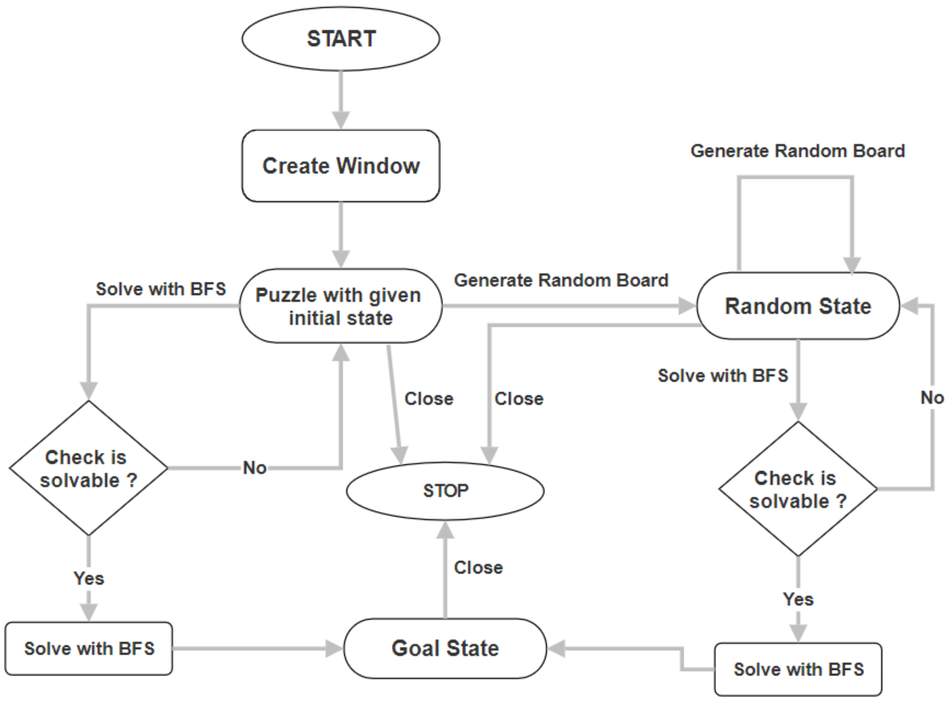
METHODOLOGY

SOURCE FILES

Source Files	Definitions
`main.cpp`	It is the entry point of the program handles the creation of the SFML window, rendering the GUI components, event handling for user interactions, and showing the solving process using the BFS algorithm. It integrates the puzzle solver logic with the graphical user interface.
`utilities.h`	Contains utility functions like shuffleVector and inv_num used for generating random initial states and checking if a puzzle configuration is solvable.
`sfmlHelpers.h`	Includes helper functions and classes for the SFML GUI implementation, such as creating buttons, rendering text, displaying progress bars, and controlling animation timing.

IMPLEMENTATION OF GUI

Flow Chart of User Interface



The graphical user interface (GUI) for the project is implemented using the SFML (Simple and Fast Multimedia Library) library, which provides a set of multimedia APIs for creating multimedia applications, including graphics rendering, events tracking and user input handling. The main components of the GUI are the main window, buttons, puzzle board rendering, and event handling.

Main Window and Rendering

The main window is created using the `sf::RenderWindow` class, with a width of 800 pixels and a height of 800 pixels. The window title is set to "Sliding Puzzle". The `drawBoard` function is responsible for rendering the puzzle board on the window. It iterates over the tiles and draws each tile as a rectangle with the corresponding number displayed inside using the `sf::RectangleShape` and `drawText` functions. The empty cell (represented by 0) is left blank.

Buttons

The GUI includes two buttons:

Generate Random Board: This button is created using the `sfButton` class, which is a custom class that encapsulates the functionality of creating and drawing buttons on the window. When the button is clicked, the `shuffleVector` function is called to generate a random initial state by shuffling the goal state. The window is then cleared, and the new initial state is rendered on the board using the `drawBoard` function.

Solve with BFS: This button is also created using the `sfButton` class. When clicked, the program checks if the initial state is solvable using the `isSolvable` function, which calculates the inversion number of the initial state. If the puzzle is solvable, the BFS algorithm is executed by calling the `BFS` function to find the minimum set of moves to solve the puzzle.

Solving Process Animation

During the solving process, the program animates the moves by swapping the tiles on the board. This is done by iterating over the sequence of moves returned by the BFS algorithm. For each move, the following steps are performed:

- The index of the empty cell (represented by 0) is found using `std::distance` and `std::find.`
- Depending on the move (Left, Right, Up, Down), the corresponding tiles are swapped using `std::swap`.
- The window is cleared, and the updated board is rendered using the `drawBoard` function.

- A progress bar is displayed using the `showProgress` function, which draws a rectangular bar representing the progress of the solving process.
- Text messages are displayed on the window to show the current move being performed and the progress (e.g., "5/20 moves").
- The window is displayed using the `window.display()` function.
- A delay is introduced using the delay function to slow down the animation for better visibility.

Solution Display

After the puzzle is solved, the final state of the board is displayed along with a message indicating the number of moves required and the computational time taken to solve the puzzle.

Event Handling

The program enters an event loop where it continuously checks for user input events, such as window closing or mouse clicks. This is done using the `sf::Event` class and the `window.pollEvent` function. The event handling logic is as follows:

- If the user closes the window, the program exits.
- If the user clicks the "Generate Random Board" button, a new initial state is generated and rendered on the board.
- If the user clicks the "Solve with BFS" button, the solving process is initiated using the BFS algorithm, and the animation and progress tracking logic is executed.

CODE: Board Generation

```
//Screen rendering

void drawBoard(sf::RenderWindow& window) {
    sf::Font font;
    if (!font.loadFromFile("arial.ttf")) {
        std::cerr << "Font file not found!" << std::endl;
    }

    //Draw the title
    Vector2f position(windowWidth / 2 - 170, 20);
    Color color = Color::Black;
    drawText(window, "3 × 3 Number Puzzle Solver", font, 30, position, color, Text::Bold);
    //Draw the Generate button
    Vector2f pos1{ windowWidth / 2 - 250,100 };
    Vector2f size1{ 250,40 };
    Color c1(12, 45, 87);
    sfButton generateBtn(pos1, size1, c1, Color::Green, Color::White, "Generate Random
Board", font, 20);
    generateBtn.draw(window);

    //Draw the BFS Solve button
    Vector2f pos2{ windowWidth / 2 - 250,550 };
    Vector2f size2{ 160,40 };
    Color c2(255, 0, 7);
    sfButton BFSBtn(pos2, size2, c2, Color::Green, Color::White, "Solve with BFS", font,
20);
    BFSBtn.draw(window);


    //Draw the tiles on the screen
  for (int i = 0; i < gridSize; i++) {
        for (int j = 0; j < gridSize; j++) {
            int number = initial[i * gridSize + j];
            if (number == 0) continue;
            RectangleShape rect(Vector2f(cellSize, cellSize));
            Color woodenColor(86, 50, 50);
            rect.setPosition(marginX + j * cellSize, marginY + i * cellSize);
            rect.setFillColor(woodenColor);
            rect.setOutlineThickness(2);
            rect.setOutlineColor(Color::Black);
            window.draw(rect);
            Vector2f position(marginX + j * cellSize + cellSize / 3, marginY + i * cellSize
+ cellSize / 3);
            Color color = Color::White;
            drawText(window, to_string(number), font, cellSize / 3, position, color,
Text::Bold);
        }
    }
}
```

CODE: Puzzle Solving Animation

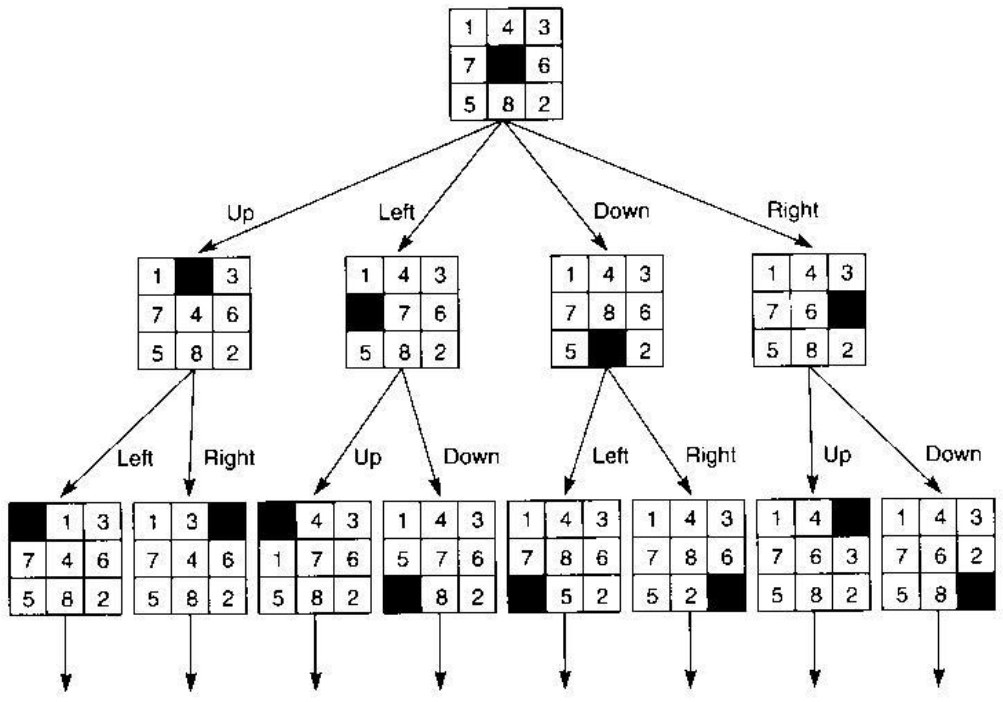
```
//Check if SOLVE with BFS Button is Clicked
if (isBtnClicked(vector<int>{150, 310, 550, 590}, mousePos) && initial != goal) {
//Find the set of moves using BFS
isSolving = true;
if (isSolvable(inv_num(initial))) {
    //Draw isSOlving State
    Vector2f position2(windowWidth / 2 - 250, 620);
    window.clear(sf::Color::White); // Set background color to white
    Color color2(51, 58, 115);
    drawText(window, "Finding the minmum set of moves...", font, 22, position2, color2, 
Text::Bold);
    drawBoard(window);
    window.display();

    //Track the duration of execution
    auto start = chrono::steady_clock::now();
    soln = BFS(initial, gridSize);
    auto end = chrono::steady_clock::now();
    duration = chrono::duration_cast<std::chrono::seconds>(end - start).count();
}
else {
    soln = {};
}
if (!soln.empty()) {
    cout << "Solution found:" << endl;
    for (string& move : soln) {
        cout << move << " ";
    }
    cout << endl;
    for (int i = 0; i < soln.size(); i++) {
        emptyIndex = distance(initial.begin(), find(initial.begin(), initial.end(), 0));
        if (soln[i] == "Left") {
            std::swap(initial[emptyIndex], initial[emptyIndex - 1]);
            emptyIndex = emptyIndex - 1;
        }
        if (soln[i] == "Right") {
            std::swap(initial[emptyIndex], initial[emptyIndex + 1]);
            emptyIndex = emptyIndex + 1;
        }
        if (soln[i] == "Down") {
            std::swap(initial[emptyIndex], initial[emptyIndex + gridSize]);
            emptyIndex = emptyIndex + gridSize;
        }
        if (soln[i] == "Up") {
            std::swap(initial[emptyIndex], initial[emptyIndex - gridSize]);
            emptyIndex = emptyIndex + gridSize;
        }
        window.clear(sf::Color::White); // Set background color to white
        drawBoard(window);
        //Draw isSOlving State
        Vector2f position2(windowWidth / 2 - 250, 620);
        Color color2(51, 58, 115);
        drawText(window, "Solving...", font, 22, position2, color2, Text::Bold);

        //Code to Generate the Progress BAR
        int progress = 300 / soln.size() * (i+1);
        showProgress(window, progress, windowWidth / 2 - 250);
        //Progress Bar Label
        Vector2f position(500, 680);
        Color color = Color::Black;
        drawText(window, to_string(i + 1) + " / " + to_string(soln.size()) + " moves", font,
25, position, color, Text::Bold);

        //Diplay Moves name
        Vector2f position3(windowWidth / 2 - 250, 750);
        drawText(window, "Current Move: " + soln[i], font, 22, position3, color2,
Text::Bold);
        window.display();
        delay(200);
    }
    window.clear(sf::Color::White);
    Vector2f position2(windowWidth / 2 - 250, 620);
    Color color2(51, 58, 115);
    drawText(window, "Solved in " + to_string(soln.size()) + " moves (Computational Time:" +
to_string(duration) + " seconds)", font, 22, position2, color2, Text::Bold);
    drawBoard(window);
    showProgress(window, 300, windowWidth / 2 - 250);
    //Progress Bar Label
    Vector2f position(500, 680);
    Color color = Color::Black;
    drawText(window, to_string(soln.size()) + " / " + to_string(soln.size()) + " moves",
font, 25, position, color, Text::Bold);
    window.display();
    isSolving = false;
}
```

Data Structures and Algorithms

State

Each instance of the State class represents a unique configuration or arrangement of the tiles on the puzzle board. This class serves as a node in the graph, where each node corresponds to a potential state in the solution path. The total number of states in the puzzle is 9! out of which only half of them is solvable.

The following graph illustrates the sequential transformation of the puzzle's configuration from one state to another, including moves and child states,



The State class encapsulates the following information:

Properties

Vector<int> state : It stores the current arrangement of tiles on the puzzle board as a vector. This vector typically represents a flattened version of the 3x3 grid, where each element corresponds to the value of a tile at a specific position. For example, a vector {1, 2, 3, 4, 5, 0, 6, 7, 8} represents the goal configuration of the puzzle, with 0 denoting the vacant position.

String direction: To reconstruct the sequence of moves that led to a particular state, each State object stores the direction of the move that produced it from its parent state. This information is crucial for backtracking the solution path once the goal state is reached.

State* Parent: It is a reference to its parent state, forming a linked structure. The parent state represents the configuration from which the current state was derived by performing a valid move. This reference allows for efficient traversal of the solution path and facilitates the backtracking process.

Methods

bool isGoalState() : This method checks whether the current state of the puzzle board represents the goal state. It compares the arrangement of tiles in the current state with the predefined goal configuration. If the configurations match, it means the puzzle is solved, and the method returns true otherwise false.

vector<string> available_moves(int x, int n): This method determines the valid moves that can be performed from the current state of the puzzle board. It takes two parameters: x representing the position of the vacant tile, and n representing the size of the board (in this case, 3 for a 3x3 puzzle). The method analyzes the position of the vacant tile and generates a vector of strings, where each string represents a valid move (e.g., "up", "down", "left", "right"). The resulting vector contains all possible moves that can be applied to the current state without violating the rules of the puzzle.

vector<State*> expand(int n): The expand method is responsible for generating the child states of the current state by applying the available moves. It takes the board size n as a parameter. The method first calls available_moves to obtain the list of valid moves for the current state. Then, for each valid move, it creates a new State object by applying the move to the current state's configuration. The new State objects are added to a vector, which is returned by the method. These child states represent the next possible configurations that can be reached from the current state.

vector<string> solution(): This method is used to backtrack and reconstruct the sequence of moves that lead from the initial state to the goal state. It operates by traversing the parent-child relationships between states, starting from the goal state and tracing back to the initial state. The method builds a vector of strings, where each string represents a move (e.g., "up", "down", "left", "right"). The moves are added to the vector in reverse order, starting from the move that led to the goal state and working backwards until the initial state is reached. The resulting vector contains the complete sequence of moves required to solve the puzzle from the given initial configuration.

Code:

```
class State {
public:
    string direction;
    vector<int> state;
    State* parent;

    State(vector<int> state, State* parent, string direction)
        : state(state), parent(parent), direction(direction){}

    bool isGoalState() {
        return state == goal;
    }

    vector<string> available_moves(int x, int n) {
        vector<string> moves = { "Left", "Right", "Up", "Down" };
        if (x % n == 0) {
            moves.erase(remove(moves.begin(), moves.end(), "Left"), moves.end());
        }
        if (x % n == n - 1) {
            moves.erase(remove(moves.begin(), moves.end(), "Right"), moves.end());
        }
        if (x - n < 0) {
            moves.erase(remove(moves.begin(), moves.end(), "Up"), moves.end());
        }
        if (x + n > n * n - 1) {
            moves.erase(remove(moves.begin(), moves.end(), "Down"), moves.end());
        }
        return moves;
    }

    vector<State*> expand(int n) {
        int x = distance(state.begin(), find(state.begin(), state.end(), 0));
        vector<string> moves = available_moves(x, n);

        vector<State*> children;
        for (string direction : moves) {
            vector<int> temp = state;
            if (direction == "Left") {
                swap(temp[x], temp[x - 1]);
            }
            else if (direction == "Right") {
                swap(temp[x], temp[x + 1]);
            }
            else if (direction == "Up") {
                swap(temp[x], temp[x - n]);
            }
            else if (direction == "Down") {
                swap(temp[x], temp[x + n]);
            }
            children.push_back(new State(temp, this, direction));
        }
        return children;
    }

    vector<string> solution() {
        vector<string> solution;
        solution.push_back(direction);
        State* path = this;
        while (path->parent != nullptr) {
            path = path->parent;
            solution.push_back(path->direction);
        }
        solution.pop_back();
        reverse(solution.begin(), solution.end());
        return solution;
    }
};
```

BFS Algorithm

BFS algorithm is implemented in the function *vector<string> BFS(vector<int>& given_state, int n)* to find the shortest path from initial to goal state. The function takes a *given_state* vector representing the initial configuration of the puzzle board and the board size *n*. A State object *root* is created to represent the initial state, with *nullptr* as the parent and an empty *move* string. If the initial state is already the goal state, the solution is immediately returned by calling *root->solution()*. A data structure queue *frontier* and an unordered set *explored* are initialized to keep track of the nodes to be explored and the states that have already been explored, respectively.

The algorithm starts by enqueuing root node to the frontier. The BFS algorithm starts a loop that continues until the frontier is empty. In each iteration, a node *current_node* is dequeued from the frontier. The state of *current_node* is marked as explored by inserting it into the explored set. The *expand* method of *current_node* is called to generate its child nodes, representing the possible next states obtained by applying valid moves.

For each child node:

a) If the child state has not been explored yet (*explored.find(child->state) == explored.end()*), it is processed further.

b) If the child state is the goal state (*child->isGoalState()*), the solution is reconstructed by calling *child->solution()*, the child node is deleted, and the solution vector is returned.

c) If the child state is not the goal state, it is enqueued to the frontier to be explored later.

If the loop completes without finding the goal state, an empty vector is returned, indicating that no solution was found.

CODE:

```
vector<string> BFS(vector<int>& given_state, int n) {
    State* root = new State(given_state, nullptr, "");
    if (root->isGoalState()) {
        return root->solution();
    }

    queue<State*> frontier;
    frontier.push(root);
    unordered_set<vector<int>, VectorHash> explored;

    while (!frontier.empty()) {
        State* current_node = frontier.front();
        frontier.pop();
        explored.insert(current_node->state);
        vector<State*> children = current_node->expand(n);
        for (State* child : children) {
            if (explored.find(child->state) == explored.end()) {
                if (child->isGoalState()) {
                    vector<string> solution = child->solution();
                    delete child;
                    return solution;
                }
                frontier.push(child);
            }
        }
    }
    return {};
}
```

GLOBAL VARIABLES

Following global variables have been declared in the program:

Variable	Description
`vector <int> initial`	Stores the initial configuration of the puzzle.
`vector <int> goal`	Represents the desired goal configuration of the puzzle.
`const int windowWidth`	Defines the width of the window where the puzzle is displayed.
`const int windowHeight`	Defines the height of the window where the puzzle is displayed.
`const int gridSize`	Defines the size of the puzzle grid
`const int marginX`	Determines the horizontal margin in window
`const int marginY`	Determines vertical margin in window
`const int cellSize`	Determines the size of each cell in grid

ANALYSIS OF TIME COMPLEXITY

The time complexity of an algorithm refers to how the execution time scales with the size of the input. In the case of the 3×3 Number Puzzle Solver, the input size is relatively small and fixed (9 cells). However, the time complexity is primarily determined by the number of possible states that need to be explored to find the solution.

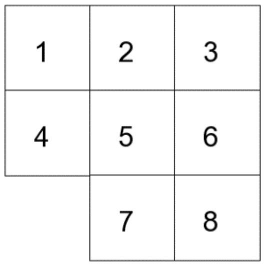
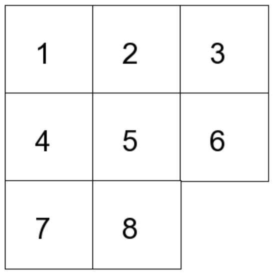
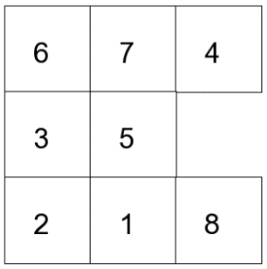
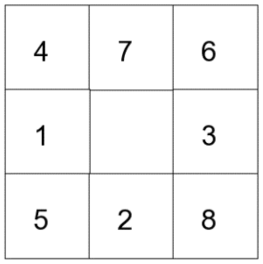
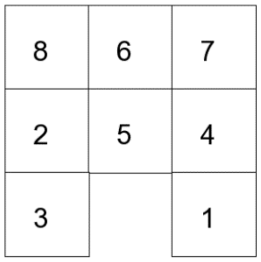
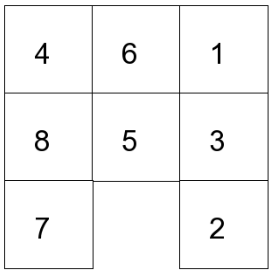
The code uses a Breadth-First Search (BFS) algorithm to explore all possible states of the puzzle until the goal state is reached. The BFS algorithm explores all the neighboring states (generated by moving the empty tile in different directions) level by level, ensuring that the solution found is the shortest possible sequence of moves.The time complexity of the BFS algorithm depends on the number of states that need to be explored, which can vary based on the initial state of the puzzle. In the worst case, when the initial state is farthest from the goal state, the algorithm may need to explore a significant portion of the state space before finding the solution.

Mathematically, the time complexity of the BFS algorithm for the 3×3 Number Puzzle Solver can be expressed as $O(b^d)$ where b is the branching factor (the number of neighboring states for each state), and d is the maximum depth of the search tree (the number of moves required to reach the goal state from the initial state).In the case of the 3×3 puzzle, the branching factor b is not always exactly 4 since some states may have fewer possible moves due to the position of the empty tile (e.g., if the empty tile is in a corner or at the edge of the board). To account for this, we can consider the weighted average of the branching factor given as,

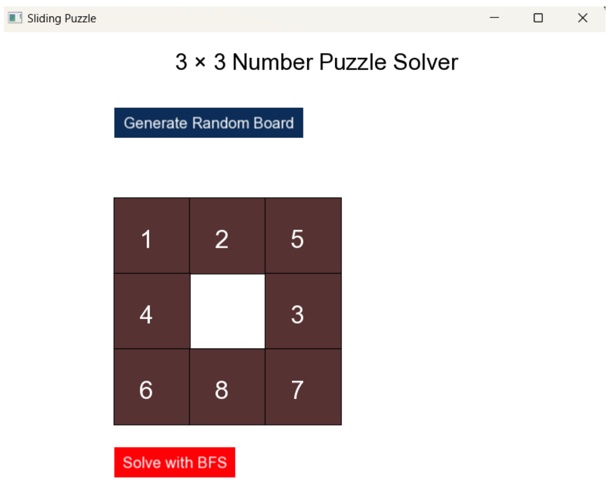
$$b = \frac{4(corners) \times 2 + 4(edges) \times 3 + 1(center) \times 4}{9} = 2.67$$

In the worst case, the puzzle can have a depth d of 31 i.e. requiring maximum of 31 moves to reach the goal state. Therefore, the worst case time complexity of the BFS algorithm for the 3×3 Number Puzzle Solver can be considered as $O(2.67^{31})$. For the average case, the depth d is taken as 20 making the average case time complexity of $O(2.67^{20})$.

STATISTICAL RESULTS

Configuration 1 (Easy)  Number of Moves Required : 2 Computation Time ≈ 0 seconds	**Configuration 2 (Easy)**  Number of Moves Required : 21 Computation Time = 4 seconds	**Configuration 3 (Medium)**  Number of Moves Required : 23 Computation Time = 11 seconds
Configuration 4 (Medium)  Number of Moves Required : 22 Computation Time = 9 seconds	**Configuration 5 (Hard)**  Number of Moves Required : 31 Computation Time = 37 seconds	**Configuration 6 (Not Solvable)**  Not solved due to odd number of inversions

OUTPUT



DISCUSSION

The development of the 3x3 Number Puzzle Solver provided a valuable learning experience, allowing us to gain practical skills in algorithm implementation, data structure utilization, complexity analysis, problem modeling, and graphical user interface (GUI) development. Through the process, we gained insights into the strengths and limitations of the Breadth-First Search (BFS) algorithm, as well as the importance of efficient data structures and optimizations for enhancing performance. Representing the puzzle as a graph and leveraging the SFML library for GUI development taught us how to create interactive and visually appealing user interfaces, enhancing the overall user experience.

While the project was successful we also encountered several limitations and challenges. One of the primary limitations was the exponential growth in time and space complexity as the puzzle size increased or the initial configurations became more scrambled. This limitation is inherent to the Breadth-First Search (BFS) algorithm, which explores all possible states systematically, leading to a rapid increase in computational resources required for larger or more complex puzzles. The visualization and user interface, while functional, lacked advanced features such as animation, customization options, and intuitive controls, which could enhance the overall user experience.

While the current implementation is functional and efficient for the 3x3 puzzle size, there is ample room for improvement. Incorporating heuristic-based search algorithms, such as the A* algorithm, could potentially boost performance, especially for larger puzzle sizes or more complex configurations. Parallelizing the search process across multiple threads or processors could accelerate solution finding, taking advantage of modern hardware capabilities. Furthermore, extending the solver to support larger puzzle sizes, enhancing the visualization and user interface, and integrating alternative algorithms like Iterative Deepening Depth-First Search (IDDFS) or Bidirectional Search could broaden the solver's applicability, improve scalability, and enable diverse solution strategies and performance comparisons.

CONCLUSION

In conclusion, the BFS algorithm is an effective approach for solving the 3x3 Number Puzzle, providing optimal solutions for all tested configurations. The algorithm's time and space complexity make it suitable for this small puzzle size, but it becomes inefficient for larger puzzle sizes due to the exponential growth in the number of states explored. Overall, we successfully implemented a solver for the 3x3 Number Puzzle which provided us valuable insights into its performance and complexity.

REFERENCES

https://www.sfml-dev.org/

https://en.wikipedia.org/wiki/Sliding_puzzle

https://www.youtube.com/watch?v=_CrEYrcImv0