Practical Applications Of The Stack Data Structure

By Group No. 5 (Comp-C)

49 Nandini Shetty	53 Harshita Shirsat	57 Shivang Shukla
50 Shishir Shetty	54 Krishna Shrivastav	58 Siddharth Shukla
51 Zidane Shikalgar	55 Aditya Shrivastava	59 Ved Shukla
52 Sarvesh Shinde	56 Krishnam Shukla	60 Arsh Siddiqui

An Innovative Exam (IE) Report
Submitted for the
Subject of
Data Structures using Java
Under the Guidance of
Mrs Soumyamol P.S.
Designation - Asst. Prof.
A. Y. 2025-2026

Table of Contents

Abstract

The stack is a fundamental data structure in computer science, characterised by its Last-In, First-Out (LIFO) principle. While simple in concept, its applications are vast and foundational to many complex software systems. This report explores the implementation and significance of the stack through three distinct, practical problems: the evaluation of mathematical expressions, the implementation of undo-redo functionality in text editors, and the pathfinding logic for maze traversal using Depth-First Search (DFS). For each problem, this report provides a theoretical background, a literature survey, a clear algorithm, a complete code implementation in Java, and a discussion of its methodology, applications, and future scope. The collective analysis of these three tasks demonstrates the versatility and power of the stack in managing nested operations, historical states, and recursive exploration.

Part 1: Expression Evaluator using a Stack

1.1 Introduction

Mathematical expression evaluation is a basic operation in computer science. Although simple expressions can be handled linearly, more complex expressions with multiple operators of different precedence levels (say, multiplication and division over addition and subtraction) and parentheses require a more advanced solution. The Last-In, First-Out (LIFO) nature of the stack data structure provides an elegant and efficient solution to this problem. This report outlines the design and implementation of an elementary calculator that evaluates an infix mathematical expression by converting it into a postfix (Reverse Polish Notation) expression and then calculating the result, all based on stacks.

1.2 Theoretical Background

A stack is a linear data structure that follows the LIFO principle. The primary operations are `push` (add an element to the top) and `pop` (remove an element from the top).

- Infix Notation: The standard mathematical notation where the operator is placed *between* the operands (e.g., `3 + 4`). This is easy for humans to read but complex for computers to parse due to the rules of precedence and associativity.
- Postfix Notation (Reverse Polish Notation - RPN): An unambiguous notation where the operator is placed *after* its operands (e.g., `3 4 +`). Expressions in RPN can be evaluated directly using a single stack, making it computationally efficient.

The core of this project is the Shunting-yard algorithm, developed by Edsger Dijkstra, which is used to convert an infix expression to a postfix expression. It uses a stack to correctly order the operators based on their precedence.

1.3 Literature Survey

The problem of evaluating arithmetic expressions is well known in computer science due to its historical importance in compiler and programming language development. One of the most important breakthroughs in the area is the shunting-yard algorithm by Dijkstra. More recent work has focused on further adapting and improving this algorithm. Some of this work addresses the algorithm's extensions to more complicated functions and operators, as well as its use in creating domain-specific languages. As with most research, there is a considerable amount of literature dealing with the time and space efficiency of stack-based evaluation, and this approach is found to be incredibly efficient in expression parsing. Other works have contrasted stack-based techniques with other methods like expression trees. They have shown that, although expression trees are more flexible in terms of symbolic manipulations, the stack-based methods are preferable in terms of raw numerical efficiency because of the less overhead.

1.4 Algorithm

The process is divided into two main parts:

1. Infix to Postfix Conversion (Shunting-yard Algorithm)
2. Postfix Expression Evaluation

Algorithm 1: Infix to Postfix

1. Initialise an empty stack for operators and an empty list or string for the postfix output.
2. Scan the infix expression from left to right.
3. If the token is an operand (a number), append it to the postfix output.
4. If the token is an operator:
   a. While the operator stack is not empty AND the operator at the top has higher or equal precedence, pop the operator from the stack and append it to the postfix output.
   b. Push the current operator onto the stack.
5. If the token is an opening parenthesis (, push it onto the stack.
6. If the token is a closing parenthesis), pop operators from the stack and append them to the output until an opening parenthesis is found. Discard both parentheses.
7. After scanning the whole expression, pop any remaining operators from the stack and append them to the output.

Algorithm 2: Postfix Evaluation

1. Initialise an empty stack for operands.
2. Scan the postfix expression from left to right.
3. If the token is an operand, push it onto the operand stack.
4. If the token is an operator, pop the top two operands from the stack.
5. Perform the operation with these two operands (the second one popped is the first operand).
6. Push the result back onto the operand stack.
7. After scanning the whole expression, the final result is the single value remaining in the stack.

1.5 Code

```
import java.util.Scanner;
import java.util.Stack;
class ExpressionEvaluator
{
 public int evaluate(String exp)
 {
  String postfix = infixToPostfix(exp);
  System.out.println("Postfix Expression: " + postfix);
  return evaluatePostfix(postfix);
 }
 public String infixToPostfix(String expression)
 {
  StringBuilder result = new StringBuilder();
  Stack<Character> stack = new Stack<>();
  for (int i = 0; i < expression.length(); i++)
  {
   char c = expression.charAt(i);
   if (Character.isDigit(c))
   {
    StringBuilder num = new StringBuilder();
    while (i < expression.length() && Character.isDigit(expression.charAt(i)))
    {
     num.append(expression.charAt(i));
     i++;
    }
```

```
      i--;
      result.append(num.toString()).append(" ");
    }
    else if (c == '(')
      stack.push(c);
    else if (c == ')')
    {
      while (!stack.isEmpty() && stack.peek() != '(')
        result.append(stack.pop()).append(" ");
      stack.pop();
    }
    else if (isOperator(c))
    {
      while (!stack.isEmpty() && precedence(c) <= precedence(stack.peek()))
        result.append(stack.pop()).append(" ");
      stack.push(c);
    }
  }
  while (!stack.isEmpty())
    result.append(stack.pop()).append(" ");
  return result.toString().trim();
}
int evaluatePostfix(String expression)
{
  Stack<Integer> stack = new Stack<>();
  Scanner oc = new Scanner(expression);
  while (oc.hasNext())
  {
    if (oc.hasNextInt())
      stack.push(oc.nextInt());
    else
    {
      String o = oc.next();
      if (o.length() == 1 && isOperator(o.charAt(0)))
      {
        int val2 = stack.pop();
        int val1 = stack.pop();
        char op = o.charAt(0);
```

```
        switch (op)
        {
          case '+': stack.push(val1 + val2); break;
          case '-': stack.push(val1 - val2); break;
          case '*': stack.push(val1 * val2); break;
          case '/': stack.push(val1 / val2); break;
        }
      }
    }
  }
  oc.close();
  return stack.pop();
}
boolean isOperator(char ch)
{
  return ch == '+' || ch == '-' || ch == '*' || ch == '/';
}
int precedence(char ch)
{
  switch (ch)
  {
    case '+': case '-': return 1;
    case '*': case '/': return 2;
  }
  return -1;
}
public static void main(String args[])
{
  Scanner sc = new Scanner(System.in);
  System.out.println("Enter a mathematical expression (e.g., 3 + (4 * 2) / (1 - 5)):");
  String exp = sc.nextLine();
  ExpressionEvaluator e = new ExpressionEvaluator();
  try
  {
    int result = e.evaluate(exp);
    System.out.println("Result: " + result);
  }
  catch (Exception ex)
```

```
  {
    System.out.println("Error: Invalid expression or calculation error.");
  }
  sc.close();
 }
}
```

1.6 Output

Enter a mathematical expression (e.g., 3 + (4 * 2) / (1 - 5)):
3+5*2-(12/6)
Postfix Expression: 3 5 2 * + 12 6 / -
Result: 11

1.7 Methodology/Current Trends

The methodology employed is a classic, algorithm-driven approach that has been a standard for decades. However, current trends in this area involve:

- Abstract Syntax Trees (ASTs): For more complex parsing, like in modern compilers, expressions are often converted into a tree structure (an AST) instead of a linear postfix string. This allows for more advanced analysis, optimisation, and code generation.
- Parser Generator Tools: Tools like ANTLR or YACC are used to automatically generate a parser from a formal grammar, which can handle much more complex expression languages with less manual coding.
- Just-In-Time (JIT) Compilation: In high-performance computing, expressions might be compiled into machine code at runtime for maximum evaluation speed.

1.8 Applications

- Compilers and Interpreters: Used to parse and evaluate mathematical expressions in programming languages.
- Scientific and Graphing Calculators: The core logic inside any calculator that respects the order of operations.
- Database Query Engines: Used to parse and execute conditions in `WHERE` clauses.
- Spreadsheet Software: Programs like Excel use this logic to calculate formulas in cells.

1.9 Future Scope

The current implementation can be extended in several ways:

- Handling Functions: Add support for mathematical functions like `sin()`, `cos()`, and `log()`.
- Variable Support: Allow the expression to contain variables and evaluate them by providing their values.

- Error Handling: Implement robust error handling for invalid expressions, such as mismatched parentheses or invalid operators.
- Floating-Point Numbers: Extend the logic to handle decimal numbers instead of just integers.

Part 2: Undo-Redo Functionality in a Text Editor

2.1 Introduction

The undo-redo feature is a common feature found in most software applications, including text editors and professional tools for graphic design. This feature allows users to disengage and re-engage with actions, enabling users to experiment freely. This feature also helps users reduce mistakes and regret. This feature is important in software applications to support an effective user experience. In most applications, the effective implementation of the undo-redo feature is done using a stack data structure. An efficient and user-friendly system for governing user actions is achieved by utilising two stacks, one for undo and one for redo operations.

2.2 Theoretical Background

The standard model for implementing undo-redo functionality relies on two stacks: an `undoStack` and a `redoStack`.

- `undoStack`: This stack stores the history of actions performed by the user. Every time a new action is taken (e.g., typing a character, applying a format), the state or command is pushed onto the `undoStack`.
- `redoStack`: This stack stores actions that have been undone. It is initially empty.

The logic operates as follows:

- Action: When a user acts, it is pushed onto the `undoStack`. The `redoStack` is cleared, as a new action invalidates the previous "redo" history.
- Undo: To undo, the most recent action is popped from the `undoStack` and pushed onto the `redoStack`. The application state is then reverted to the state before that action.
- Redo: To redo, an action is popped from the `redoStack` and pushed back onto the `undoStack`. The application state is then moved forward to reflect that action.

This two-stack approach provides a robust and computationally inexpensive way to manage a linear history of operations.

2.3 Literature Survey

The principles of HCI and software architectural solutions provide the backbone for the undo-redo feature. Earlier studies have concentrated on cognitive aspects concerning forgiving interfaces. The command pattern, documented in the software engineering canon, serves as the object-oriented basis for the implementation of undo-redo in software. It captures a request or action in an object, thus enabling the use of different clients with different requests and the possibility of supporting undoable operations. In HCI, history features have been studied concerning usability, as with linear undo (our model) and non-linear or selective undo, where users operate in a history of actions. The most recent studies in the design of collaborative systems focus on the difficulty of implementing undo-redo in multi-user settings where different users' actions have been blended.

2.4 Algorithm

Let `undoStack` and `redoStack` be two stacks. Let `currentState` be the current text.

Algorithm for a new action (e.g., typing text):

1. Push the `currentState` onto the `undoStack`.
2. Update `currentState` with the new text.
3. Clear the `redoStack`.

Algorithm for Undo:

1. If undoStack is not empty:
   a. Push the currentState onto the redoStack.
   b. Pop the previous state from undoStack and make it the currentState.
2. Else, do nothing.

Algorithm for Redo:

1. If redoStack is not empty:
   a. Push the currentState onto the undoStack.
   b. Pop the next state from redoStack and make it the currentState.
2. Else, do nothing.

2.5 Code

```
import java.util.Scanner;
import java.util.Stack;
class TextEditor
```

```
{
String current;
Stack<String> undoStack;
Stack<String> redoStack;
public TextEditor()
{
  current = "";
  undoStack = new Stack<>();
  redoStack = new Stack<>();
}
public void type(String s)
{
  undoStack.push(current);
  current = s;
  redoStack.clear();
  System.out.println("Current Text: \"" + current + "\"");
}
public void undo()
{
  if (!undoStack.isEmpty())
  {
    redoStack.push(current);
    current = undoStack.pop();
    System.out.println("UNDO. Current Text: \"" + current + "\"");
  }
  else
    System.out.println("Nothing to undo.");
}
public void redo()
{
  if (!redoStack.isEmpty())
  {
    undoStack.push(current);
    current = redoStack.pop();
    System.out.println("REDO. Current Text: \"" + current + "\"");
  }
  else
    System.out.println("Nothing to redo.");
```

```
}
public static void main(String args[])
{
  Scanner sc = new Scanner(System.in);
  TextEditor editor = new TextEditor();
  while (true)
  {
    System.out.println("\n--- Text Editor Menu ---\n1. Type (replace current text)\n2. Undo\n3.
Redo\n4. Exit");
    System.out.print("Enter your choice: ");
    int c = sc.nextInt();
    sc.nextLine();
    switch (c)
    {
      case 1:
        System.out.print("Enter the new text: ");
        String text = sc.nextLine();
        editor.type(text);
        break;
      case 2:
        editor.undo();
        break;
      case 3:
        editor.redo();
        break;
      case 4:
        System.out.println("Exiting editor.");
        sc.close();
        return;
      default:
        System.out.println("Invalid choice. Please try again.");
    }
  }
}
}
```

2.6 Output

--- Text Editor Menu ---

1. Type (replace current text)
2. Undo
3. Redo
4. Exit
Enter your choice: 1
Enter the new text: Hello World
Current Text: "Hello World"

--- Text Editor Menu ---
1. Type (replace current text)
2. Undo
3. Redo
4. Exit
Enter your choice: 1
Enter the new text: Hello Java
Current Text: "Hello Java"

--- Text Editor Menu ---
1. Type (replace current text)
2. Undo
3. Redo
4. Exit
Enter your choice: 2
UNDO. Current Text: "Hello World"

--- Text Editor Menu ---
1. Type (replace current text)
2. Undo
3. Redo
4. Exit
Enter your choice: 2
UNDO. Current Text: ""

--- Text Editor Menu ---
1. Type (replace current text)
2. Undo
3. Redo
4. Exit

```
Enter your choice: 3
REDO. Current Text: "Hello World"

--- Text Editor Menu ---
1. Type (replace current text)
2. Undo
3. Redo
4. Exit
Enter your choice: 3
REDO. Current Text: "Hello Java"

--- Text Editor Menu ---
1. Type (replace current text)
2. Undo
3. Redo
4. Exit
Enter your choice: 4
Exiting editor.
```

2.7 Methodology/Current Trends

The two-stack model is a well-known example of a methodology. A modern, object-oriented strategy is to use the Command design pattern. Within this pattern, each action (such as typing, deleting, or formatting) is encapsulated in its class, which implements the shared command interface containing the methods execute() and unexecute(). The undoStack in this instance holds command objects, which is superior to storing raw state, as it provides greater flexibility and is more memory efficient in complicated applications.

Inspired by version control systems like Git, we are also examining non-linear history management. This would let users rewind their history, make alterations, and then create new “branches” of their work instead of the linear overwrite model.

2.8 Applications

- Word Processors: Microsoft Word, Google Docs.
- Integrated Development Environments (IDEs): IntelliJ IDEA, VS Code, Eclipse.
- Graphic Design Software: Adobe Photoshop, Figma (for every change, like moving an object, changing a colour, etc.).
- Database Management Systems: To roll back transactions.

2.9 Future Scope

- Selective Undo: Allowing users to undo a specific action from the middle of the history without affecting subsequent actions.
- Batch Operations: Grouping multiple actions into a single undoable transaction.
- Persistent History: Saving the undo/redo history so it persists even after the application is closed and reopened.
- Collaborative Undo: Implementing a model that works for multiple users editing the same document simultaneously.

Part 3: Stack-based Depth-First Traversal of a Maze

3.1 Introduction

Maze solving is an example of a problem in computer science, particularly one involving the application of graph traversal algorithms. A maze is a grid of cells, which can be viewed as a graph where the cells form the vertices and open adjacent cells form the edges. The objective is to navigate from a given starting cell to an exit cell. Depth-First Search (DFS) is one of the most basic algorithms that can be used to find a solution to this problem. Though DFS is commonly taught as a recursive algorithm, it can be just as beneficial to implement it iteratively with an explicit stack, especially in terms of memory for deeply nested mazes. The purpose of this report is to document the implementation of an iterative, stack-based DFS to find a solution to a maze.

3.2 Theoretical Background

Depth-First Search (DFS) is a graph traversal algorithm that explores as far as possible along each branch before backtracking. The algorithm starts at a root node and explores each branch to its deepest point before moving to the next branch.

The LIFO (Last-In, First-Out) nature of a stack makes it the perfect data structure for implementing DFS iteratively:

1. When a new path (a neighbour cell) is discovered, it is pushed onto the stack.
2. The algorithm always explores from the most recently discovered path, which is the one at the top of the stack.
3. When a path leads to a dead end, that path is popped from the stack, and the algorithm automatically "backtracks" to the previous cell to explore its other neighbours.

This process continues until the destination is found or the stack becomes empty, which would mean no path exists.

3.3 Literature Survey

The algorithms for solving mazes have applications in computer science and are employed in robotics. The idea of solving a maze by following a single route through it leads to algorithms such as Trémaux's algorithm, which is a variant of DFS. The foundational literature of computer science considers DFS as one of the main methods for traversing a graph. Scientific publications compare DFS to another traversal algorithm, Breadth-First Search (BFS). While BFS is guaranteed to find the shortest path, DFS is usually more economical on memory in large, sprawling graphs, where it only needs to retain the current path in memory. More sophisticated pathfinding algorithms like Dijkstra's and A* are popular in contemporary research, for instance in robotics and video game development, as these algorithms are designed to determine the shortest route in a complicated weighted graph. For the more straightforward task of finding a valid route in a graph, however, the elegance and effectiveness of stack-based DFS is hard to beat.

3.4 Algorithm

Let maze be a 2D grid where 0 is a path and 1 is a wall.

Let start and end be the coordinates of the start and end points.

Algorithm: Stack-based DFS for Maze Traversal

1. Create a `visited` 2D boolean array, initialised to `false`.
2. Create an empty stack of `Cell` objects (a `Cell` stores its x, y coordinates).
3. Push the `start` cell onto the stack.
4. While the stack is not empty:
   a. Pop a cell from the stack. Let its coordinates be (x, y).
   b. If (x, y) is the end cell, a path has been found. Return true.
   c. If (x, y) has not been visited:
   i. Mark (x, y) as visited.
   ii. Check its neighbours (up, down, left, right).
   iii. For each valid, unvisited neighbour (i.e., within bounds, not a wall, and not visited), push it onto the stack.
5. If the loop finishes and the end is not reached, the stack is empty, meaning no path exists. Return `false`.

3.5 Code

```
import java.util.Stack;
class MazeSolver
{
 static class Cell
 {
  int x, y;
  Cell(int a, int b)
  {
   x = a;
   y = b;
  }
 }
 public boolean solveMaze(int maze[][], Cell start, Cell end)
 {
  if (maze == null || maze.length == 0) return false;
  int rows = maze.length;
  int cols = maze[0].length;
  boolean[][] visited = new boolean[rows][cols];
  Stack<Cell> stack = new Stack<>();
  stack.push(start);
  while (!stack.isEmpty())
  {
   Cell current = stack.pop();
   int x = current.x;
   int y = current.y;
   if (x == end.x && y == end.y)
    return true;
   if (visited[x][y])
    continue;
   visited[x][y] = true;
   System.out.println("Visiting: (" + x + ", " + y + ")");
   if (isValid(x, y + 1, rows, cols, maze, visited))
    stack.push(new Cell(x, y + 1));
   if (isValid(x, y - 1, rows, cols, maze, visited))
    stack.push(new Cell(x, y - 1));
   if (isValid(x + 1, y, rows, cols, maze, visited))
```

```
      stack.push(new Cell(x + 1, y));
    if (isValid(x - 1, y, rows, cols, maze, visited))
      stack.push(new Cell(x - 1, y));
  }
  return false;
}
private boolean isValid(int x, int y, int rows, int cols, int[][] maze, boolean[][] visited)
{
  return x >= 0 && x < rows && y >= 0 && y < cols && maze[x][y] == 0 && !visited[x][y];
}

public static void main(String args[])
{
  MazeSolver s = new MazeSolver();
  int[][] maze = {
          {0, 1, 0, 0, 0},
          {0, 1, 0, 1, 0},
          {0, 0, 0, 1, 0},
          {0, 1, 1, 1, 0},
          {0, 0, 0, 0, 0}
                };
  Cell start = new Cell(0, 0);
  Cell end = new Cell(4, 4);
  System.out.println("Attempting to solve the maze...");
  boolean check = s.solveMaze(maze, start, end);
  if (check)
    System.out.println("\nPath found from start (0,0) to end (4,4)!");
  else
    System.out.println("\nNo path found.");
}
}
```

3.6 Output

Attempting to solve the maze...
Visiting: (0, 0)
Visiting: (1, 0)
Visiting: (2, 0)
Visiting: (3, 0)

Visiting: (4, 0)
Visiting: (4, 1)
Visiting: (4, 2)
Visiting: (4, 3)

Path found from start (0,0) to end (4,4)!

3.7 Methodology/Current Trends

The methodology used is an iterative Depth-First Search. This approach is functionally equivalent to a recursive DFS but avoids the potential for a `StackOverflowError` in very large or deep mazes by using the heap memory (for the `Stack` object) instead of the call stack.

Current trends in pathfinding have largely moved towards more optimal algorithms for specific use cases:

- A* (A-star) Algorithm: This is the industry standard in video games and robotics. It's an informed search algorithm that uses heuristics to find the shortest path much more quickly than uninformed searches like DFS or BFS.
- Bidirectional Search: For problems with a known start and end point, running two searches simultaneously (one from the start, one from the end) can find a path much faster.

3.8 Applications

- Pathfinding: In video games for non-player characters (NPCs) and in robotics for navigation.
- Web Crawling: Search engines use traversal algorithms to discover and index web pages by following hyperlinks.
- Solving Puzzles: Algorithms for solving puzzles like Sudoku or finding a way out of a labyrinth.
- Network Analysis: Finding connected components in a computer or social network.

3.9 Future Scope

- GUI Visualisation: Create a graphical user interface to visually show the maze and the path being explored by the DFS algorithm in real time.
- Shortest Path: Modify the implementation to use a queue (for Breadth-First Search) to find the shortest path instead of just any path.
- Weighted Mazes: Extend the maze to have different "costs" for moving through different cells and implement Dijkstra's or the A* algorithm to find the cheapest path.

Conclusion

The stack has demonstrated its usefulness in three different areas: expression parsing, state management, and graph traversal. It is simple Last-In, First-Out (LIFO) method is suitable for dealing with the nested hierarchical intricacy of mathematical expressions, managing user action history for undo-redo processes, and the backtracking in depth-first search. While there are more sophisticated data structures and algorithms tailored to each problem, the solutions proposed in this report, which utilise a stack, are as simple as they are efficient and provide an essential insight into the principles of computer science.

Acknowledgement

We would like to express our sincere gratitude to our teacher, Soumya Ma'am, for their invaluable guidance and support throughout the course of this work. Their expertise and encouragement were instrumental in the successful completion of this report.

References

1. Aho, A. V., Sethi, R., & Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.
2. Baeza-Yates, R. (1989). A New Algorithm for the Evaluation of Arithmetic Expressions. *SIGPLAN Notices*, 24(1), 27-31.
3. Berlage, T. (1994). A selective undo mechanism for graphical user interfaces based on command objects. *ACM Transactions on Computer-Human Interaction*, 1(3), 269-294.
4. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT press.
5. Dijkstra, E. W. (1961). *Algol 60 translation: An algol 60 translator for the x1 and making a translator for algol 60*. Stichting Mathematisch Centrum.
6. Edwards, W. K., Mynatt, E. D., & Stockton, K. (1995). A scripting language for building multi-user interactive applications. *Proceedings of the 8th annual ACM symposium on User interface and software technology*, 239-248.
7. Even, S. (2011). *Graph algorithms*. Cambridge University Press.
8. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
9. Grune, D., & Jacobs, C. J. H. (2008). *Parsing Techniques: A Practical Guide*. Springer Science & Business Media.
10. Hamblin, C. L. (1962). Translation to and from Polish notation. *The Computer Journal*, 5(3), 210-213.

11. Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2), 100-107.
12. Hopcroft, J., & Tarjan, R. (1973). Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6), 372-378.
13. Knuth, D. E. (1997). *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley.
14. Lucas, É. (1882). *Récréations mathématiques*. Gauthier-Villars.
15. Myers, B. A., & Kosbie, D. S. (1996). Reusable hierarchical command objects. *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, 260-267.
16. Naur, P. (Ed.). (1963). Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 6(1), 1-17.
17. Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), 1053-1058.
18. Prasanna, M., & Venkatesan, R. (2011). A survey on evaluation of arithmetic expressions. *International Journal of Computer Applications*, 21(8), 36-41.
19. Ressel, M., & Nitsche-Ruhland, D. (1996). An object-oriented framework for developing multi-user interactive applications. *Computers & Graphics*, 20(3), 399-407.
20. Russell, S. J., & Norvig, P. (2016). *Artificial intelligence: a modern approach*. Pearson Education Limited.
21. Sedgewick, R., & Wayne, K. (2011). *Algorithms*. Addison-Wesley Professional.
22. Shneiderman, B. (1982). The future of interactive systems and the emergence of direct manipulation. *Behaviour and Information Technology*, 1(3), 237-256.
23. Sun, C. (2000). Undo as a concurrent primitive in multi-user applications. *Proceedings of the Australasian conference on Computer science*, 22, 219-226.
24. Thimbleby, H. (2007). *Press On: A Design Approach for Everyday Devices*. MIT Press.