ECE 4122/6122

Advanced Programming Techniques

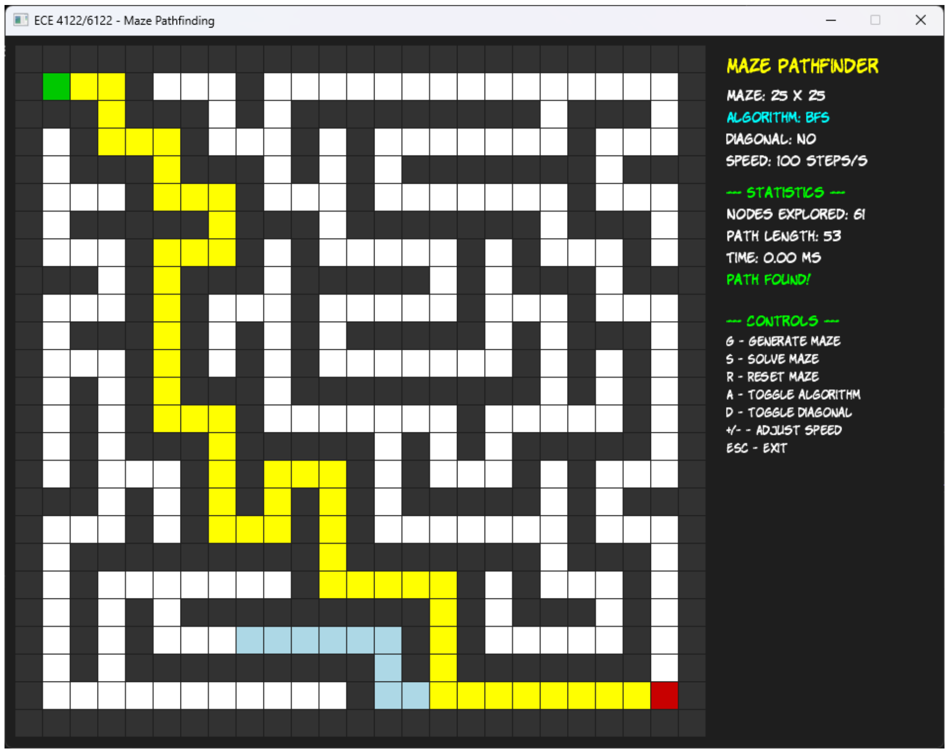
Homework 1: SFML Maze Pathfinding Game

Due Date: February 17, 2026, by midnight.
Submission: Submit via Canvas
Points: 100 points

Overview

In this assignment, you will create a 2D maze game using the SFML (Simple and Fast Multimedia Library) graphics library. The game will feature a randomly generated maze where the player must navigate from a start position to an end position. You will implement a shortest path algorithm that visually demonstrates the optimal route through the maze.

This assignment reinforces concepts in graph algorithms, 2D graphics programming, and software design patterns while giving you hands-on experience with a widely-used multimedia library.



```
MAZE PATHFINDER
MAZE: 25 X 25
ALGORITHM: BFS
DIAGONAL: NO
SPEED: 100 STEPS/S

--- STATISTICS ---
NODES EXPLORED: 61
PATH LENGTH: 53
TIME: 0.00 MS
PATH FOUND!

--- CONTROLS ---
G - GENERATE MAZE
S - SOLVE MAZE
R - RESET MAZE
A - TOGGLE ALGORITHM
D - TOGGLE DIAGONAL
+/- - ADJUST SPEED
ESC - EXIT
```

Learning Objectives

1. Implement graph-based shortest path algorithms (BFS or A*)
2. Use SFML for 2D graphics rendering and event handling
3. Apply maze generation algorithms (recursive backtracking or randomized algorithms)

4. Practice modular C++ code design with proper separation of concerns
5. Handle user input and implement interactive visual feedback

Requirements

Maze Generation (20 points)

1. Implement a maze generation algorithm that creates a random, solvable maze
2. The maze must be at least 25x25 cells (configurable via command line)
3. Walls should be clearly distinguishable from open paths
4. The maze must have exactly one start position (green) and one end position (red)

Pathfinding Algorithm (30 points)

5. Implement either Breadth-First Search (BFS) or A* algorithm to find the shortest path
6. The algorithm must find the optimal (shortest) path from start to end
7. Visualize the algorithm's exploration process (cells visited during search)
8. Highlight the final shortest path in a distinct color (e.g., yellow or blue)

SFML Graphics (25 points)

9. Use SFML 2.5+ to render the maze in a resizable window
10. Display the maze using colored rectangles or sprites for each cell type
11. Implement smooth animation for the pathfinding visualization (adjustable speed)
12. Display relevant information (maze size, path length, algorithm used) as text overlay

User Interaction (15 points)

13. Press 'G' to generate a new random maze
14. Press 'S' to solve the maze and display the shortest path
15. Press 'R' to reset the current maze (clear path visualization)
16. Press 'Escape' to exit the application
17. Use '+' and '-' keys to adjust animation speed
18. Handle window close event properly

Code Quality (10 points)

19. Proper use of classes and object-oriented design
20. Clear separation between maze logic, pathfinding, and rendering
21. Meaningful variable and function names
22. Appropriate comments explaining complex logic
23. No memory leaks; proper resource management

Color Scheme

Use the following color scheme for consistency:

Cell Type	Color	RGB Value
Wall	Dark Gray	`(50, 50, 50)`
Open Path	White	`(255, 255, 255)`
Start	Green	`(0, 200, 0)`

End	Red	`(200, 0, 0)`
Visited	Light Blue	`(173, 216, 230)`
Shortest Path	Yellow	`(255, 255, 0)`

Command Line Arguments

Your program should accept the following optional command line arguments:

```
./Lab2 [width] [height]
```

- `width`: Number of cells horizontally (default: 25, minimum: 10, maximum: 100)
- `height`: Number of cells vertically (default: 25, minimum: 10, maximum: 100)

Provided Sample Code

Sample code files are provided to help you get started:

- `MazeGenerator.h/cpp`: Sample maze generation using recursive backtracking algorithm
- `PathfindingNotes.md`: Detailed notes on BFS and A* algorithms with pseudocode

Note: *You may modify the provided code or write your own implementation from scratch.*

Online Resources

The following open-source C++ projects may be used as references. You may study these implementations but must write your own code. Make sure you include a reference at the top of your files that use these sources.

Maze Generation Libraries

- **mazegen** - Header-only C++ maze generation library `https://github.com/aleksandrbazhin/mazegen`
- **Maze-Generator-and-Solver** - Complete SFML maze project with recursive backtracking and A* `https://github.com/fuyalasmit/Maze-Generator-and-Solver`
- **maze-generator (fawzeus)** - Multiple algorithms: DFS, Hunt-and-Kill, Prim's, Shift Origin `https://github.com/fawzeus/maze-generator`

Shortest Path Algorithm Implementations

- **JeremyDsilva/MazeSolver** - Generic maze solver with DFS, BFS, Dijkstra, and A* `https://github.com/JeremyDsilva/MazeSolver`
- **PathFinding_Cpp** - SFML visualizer with Dijkstra and A* using minHeap `https://github.com/satwikShresth/PathFinding_Cpp`
- **JDSherbert/A-Star-Pathfinding** - Clean A* implementation with multiple heuristics `https://github.com/JDSherbert/A-Star-Pathfinding`
- **BFS_Simulator** - SFML BFS visualization for 2D grids `https://github.com/Bishesh-Khanal/BFS_Simulator`
- **pathfinding-visualizer** - Simple SFML Dijkstra/A* implementation `https://github.com/Parimal7/pathfinding-visualizer`

Educational Resources

- **Red Blob Games - A* Tutorial** - Excellent interactive explanation
  `https://www.redblobgames.com/pathfinding/a-star/introduction.html`
- **Bob Nystrom - Rooms and Mazes** - Dungeon generation algorithm explanation
  `https://journal.stuffwithstuff.com/2014/12/21/rooms-and-mazes/`

Submission Requirements

Submit a single zip file named `LastName_FirstName_HW1.zip` containing:

- All source files (.cpp, .h)
- CMakeLists.txt for building the project
- README.md with build instructions and any additional notes
- Do NOT include build directories, executables, or SFML libraries

Grading Rubric

Component	Points	Percentage
Maze Generation	20	20%
Pathfinding Algorithm	30	30%
SFML Graphics & Visualization	25	25%
User Interaction	15	15%
Code Quality & Documentation	10	10%
Total	**100**	**100%**

ECE 6122 Additional Requirements

Graduate students (ECE 6122) must also implement:

- Both BFS and A* algorithms with ability to switch between them (Press 'A' to toggle)
- Display comparison statistics: nodes explored, path length, computation time

Tips and Hints

- Start by getting SFML installed and rendering a simple grid before implementing maze generation
- Test your maze generation separately before integrating with graphics
- Use std::queue for BFS and std::priority_queue for A*
- Consider using sf::Clock for animation timing
- The Manhattan distance heuristic works well for A* in grid-based mazes
- Refer to SFML documentation: https://www.sfml-dev.org/documentation/

Academic Integrity

This is an individual assignment. You may discuss concepts with classmates but all code must be your own. Using AI tools (ChatGPT, Copilot, etc.) to generate substantial portions of code is not permitted. Cite any external resources you reference.

Violations will result in a zero for the assignment and potential referral to the Office of Student Integrity.

LATE POLICY

Element	**Percentage Deduction**	**Details**
Late Deduction Function	score - 0.5*H	H = number of hours (ceiling function) passed deadline

Appendix A: Coding Standards

Indentation:

When using *if/for/while* statements, make sure you indent 4 spaces for the content inside those. Also make sure that you use spaces to make the code more readable.
For example:

```
for (int i; i < 10; i++)
{
    j = j + i;
}
```

If you have nested statements, you should use multiple indentions. Each { should be on its own line (like the *for* loop) If you have *else* or *else if* statements after your *if* statement, they should be on their own line.

```
for (int i; i < 10; i++)
{
   if (i < 5)
   {
       counter++;
       k -= i;
   }
   else
   {
       k +=1;
   }
   j += i;
}
```

Camel Case:

This naming convention has the first letter of the variable be lower case, and the first letter in each new word be capitalized (e.g. firstSecondThird). This applies for functions and member functions as well! The main exception to this is class names, where the first letter should also be capitalized.

Variable and Function Names:

Your variable and function names should be clear about what that variable or function is. Do not use one letter variables, but use abbreviations when it is appropriate (for example: “imag" instead of “imaginary”). The more descriptive your variable and function names are, the more readable your code will be. This is the idea behind self-documenting code.

File Headers:
Every file should have the following header at the top

```
/*
Author: <your name>
Class: ECE4122 or ECE6122
Last Date Modified: <date>

Description:

What is the purpose of this file?

*/
```

Code Comments:

1. Every function must have a comment section describing the purpose of the function, the input and output parameters, the return value (if any).
2. Every class must have a comment section to describe the purpose of the class.
3. Comments need to be placed inside of functions/loops to assist in the understanding of the flow of the code.