P3 報告

1. Title and Author

- Assignment number - **p3**
- Name - **潘煒翔**
- Student number - **112950009**
- Email - **panweikyle.sc12@nycu.edu.tw**

---

2. Description of the Problem

在一個 $m \times n$ 的迷宮中（每格為 `'0'`、`'x'`、`s`、`t`），從唯一的起點 `s` 尋找到唯一終點 `t` 的**最短路徑**。輸入以每列非空元素的稀疏格式表示（每列列出 `col token` 的對並以 `0` 結尾），輸出需列出路徑座標序列，並附上路徑標示。

---

3. Design Overview

目標

- 解析作業指定的輸入格式（稀疏列表示）。
- 建立 `c1 \times c2` 的網格表示，並提供互動式的 `graph()` 建圖工具（可放 `x`、`s`、`t`）。
- 使用 **BFS**（Breadth-First Search）在無權網格上求得最短路徑。
- 輸出路徑座標序列與矩陣標註版本。

架構

1. **選項 1 — 手動輸入迷宮 (graph)：** 讓使用者逐列輸入 `x`、`s`、`t` 位置，建立迷宮。
2. **選項 2 — 自訂隨機迷宮 (`generateCustom()`)：** 根據使用者指定的尺寸與障礙比例自動生成隨機迷宮。
3. **選項 3 — 效能測試 (`performanceTest()`)：** 自動進行多組尺寸與障礙比例的效能測試，統計運算時間與是否能找到路徑。

---

4. `graph()` 設計

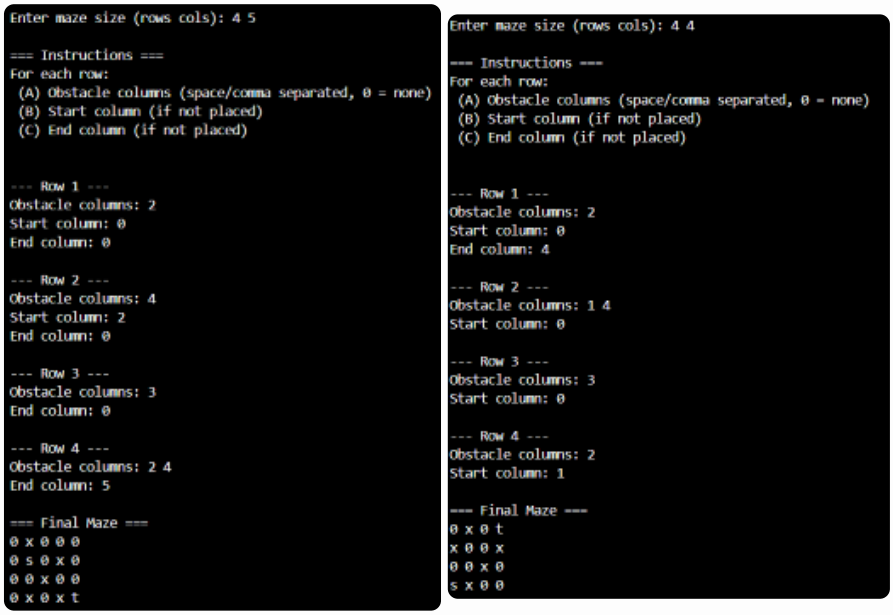
介面

- 先輸入 `c1 c2`（rows cols），先預設一個全 `0` 的迷宮，再對每一行 `row` 做更新
- 對每一列詢問：要放哪些 `x`（多個，空白或逗號分隔，輸入 `0` 表示無），若 `s` 或 `t` 尚未放置則詢問該列是否要放 `s` / `t`（輸入單一欄位或 `0`）
- 每次輸入即寫入 `grid`，最後印出整張圖，並回傳 `(grid, s_pos, t_pos)`

資料結構

- `grid`: 2D array（C++：`vector<vector<char>>`，Python：`list[list[str]]`）
- `s_pos, t_pos`: 1-based `Coord`。

輸出驗證



---

5. BFS 最短路徑

資料結構

- `queue<Coord>`：BFS 佇列
- `visited`：集合或 2D boolean 陣列
- `parent`：用以回溯的映射（或 2D Coord 陣列）

演算法步驟

1. **初始化**：將起點 `s` 放入佇列（queue），並標記 `visited[s] = true`、`parent` 留空

2. **彈出並擴展**：當 queue 非空時，彈出目前格點 `cur = (r,c)`。

- 依序檢查四個鄰居：`(r-1,c)`, `(r+1,c)`, `(r,c-1)`, `(r,c+1)`。

- **邊界檢查**：若鄰居座標超出 `1..rows` 或 `1..cols` 的範圍，**停止對該鄰居更新**（略過）。

- **障礙檢查**：若鄰居為 `x`（牆/障礙），**停止對該鄰居更新**（略過）

- **造訪檢查**：若鄰居尚未被造訪，則：

  1. 標記 `visited[neighbor] = true`；
  2. 紀錄 `parent[neighbor] = cur`；
  3. 將 `neighbor` 推入 queue 等待後續擴展。

3. **最短性條件**：在擴展過程中，**首次遇到 `t`**（即 `cur == t` 或鄰居為 `t`）時，立刻停止搜尋。由於 BFS 以「層級」擴展，**首次到達 `t` 的路徑即為最短 path**

4. **回溯或無解**：

   - 若搜尋結束 `t` 從未被造訪 → **No path（死路/不可達）**。
   - 否則自 `t` 沿 `parent` 逐格回溯到 `s`，反轉後得到最短路徑座標序列

複雜度

- 時間複雜度：$O(mn)$（每格最多被訪問一次）
- 空間複雜度：$O(mn)$（visited + parent）

6.Random Maze `generateCustom()`

目的

讓使用者可快速產生一個含隨機障礙、起點與終點的迷宮，作為測試輸入

演算法流程

1. 提示使用者輸入三個參數：`rows`、`cols`、`obstacleRatio`（0.0–1.0 之間）。

2. 驗證輸入：

   - 若尺寸無效 → 預設為 **100 × 100**。
   - 若障礙比例無效 → 預設為 **0.25**。

3. 印出確認資訊

4. 呼叫 `generate(rows, cols, obstacleRatio)`：
    - 初始化所有格為 `'0'`。
    - 以 `obstacleRatio` 的機率隨機放置 `'x'`。
    - 隨機選取非 `'x'` 的格子作為 `s`（起點）與 `t`（終點），最多嘗試 1000 次。

5. 回傳一個包含：
    - `grid`（迷宮二維陣列）
    - `start`、`end`（1-based 座標）
    - `hasStart`、`hasEnd`（布林標記）的結構。

輸出與顯示

- 儲存至 `maze.txt`（稀疏格式）。
- 在終端顯示統計資料與最多 20×20 的迷宮預覽。

邊界條件與限制

- 若障礙比例過高，可能在 1000 次嘗試內放不下 `s` 或 `t`。
- `s` 與 `t` 不保證連通。
- 沒有設定固定亂數種子，因此每次結果不同。

---

7. 效能測試 `performanceTest()`

目的

測試隨機迷宮生成與 BFS 搜尋的運算時間

測試設定

- 迷宮尺寸（正方形）：`{100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 5000, 10000}`。
- 障礙比例：`{0.1, 0.2, 0.3}`。

執行流程

對每一組 `(size, obstacleRatio)`：

1. 記錄開始時間

2. 呼叫 `generate(size, size, obstacleRatio)` 產生迷宮

3. 若同時存在 `s` 與 `t`，則呼叫 `findPath()` 以 BFS 尋找最短路徑

4. 記錄結束時間並計算耗時（毫秒）

5. 印出測試結果表：

6. 若任一測試超過 60 秒，則提前結束所有測試

測量內容

- 總時間包含「隨機生成」與「BFS 搜尋」兩部分
- 輸出是否找到路徑（Path Found）

---

8. 測試資料

1. Manual maze input

1.P3 homework example

```
=== Final Maze ===
0 x 0 0 0
0 s 0 x 0
0 0 x 0 0
0 x 0 x t

Path: (2,2)(2,3)(1,3)(1,4)(1,5)(2,5)(3,5)(4,5)

Maze with path:
0 x 2 3 4
0 s 1 x 5
0 0 x 0 6
0 x 0 x t
```

2.No path example

```
=== Final Maze ===
0 x 0 x 0
0 0 s x x
0 x 0 x t
0 0 x 0 0
No path found.
```

3.Single path

```
=== Final Maze ===
0 x 0 t
x 0 0 x
0 0 x 0
s x 0 0

Path: (4,1)(3,1)(3,2)(2,2)(2,3)(1,3)(1,4)

Maze with path:
0 x 5 t
x 3 4 x
1 2 x 0
s x 0 0
```

4.Multiple path

```
=== Final Maze ===
0 0 x 0 x
x 0 0 0 x
0 s 0 x 0
0 0 0 0 t
x 0 x 0 0

Path: (3,2)(4,2)(4,3)(4,4)(4,5)

Maze with path:
0 0 x 0 x
x 0 0 0 x
0 s 0 x 0
0 1 2 3 t
x 0 x 0 0
```

2. Custom random maze

1.10 by 10

- input:

```
Choice: 2
Enter maze dimensions:
Rows: 10
Columns: 10
Obstacle ratio (0.0 - 1.0): 0.3
Generating 10x10 maze with 30% obstacles...
Maze saved to: maze.txt

=== Maze Statistics ===
Size: 10 x 10
Start: (10, 3)
End: (6, 9)
Path length: 0

=== Complete Maze ===
x 0 0 0 x x 0 0 x 0
x 0 0 0 x 0 0 x 0 0
0 0 0 0 0 x 0 0 0 x
0 0 x 0 0 0 x 0 0 0
x 0 0 0 0 0 0 x x 0
0 0 0 0 x 0 0 0 t 0
0 0 0 x 0 x x x x 0
0 0 x 0 x 0 0 x 0 0
0 x 0 0 x 0 0 0 0 0
0 0 s 0 0 0 x 0 0 0
```

- maze:

```
10 10
1 x 5 x 6 x 9 x 0
1 x 5 x 8 x 0
6 x 10 x 0
3 x 7 x 0
1 x 8 x 9 x 0
5 x 9 t 0
4 x 6 x 7 x 8 x 9 x 0
3 x 5 x 8 x 0
2 x 5 x 0
3 s 7 x 0
```

- path:

```
(10,3)(10,4)(10,5)(10,6)(9,6)(9,7)(9,8)(9,9)(8,9)(8,10)(7,10)(6,10)(6,9)
```

2. 100 by 100

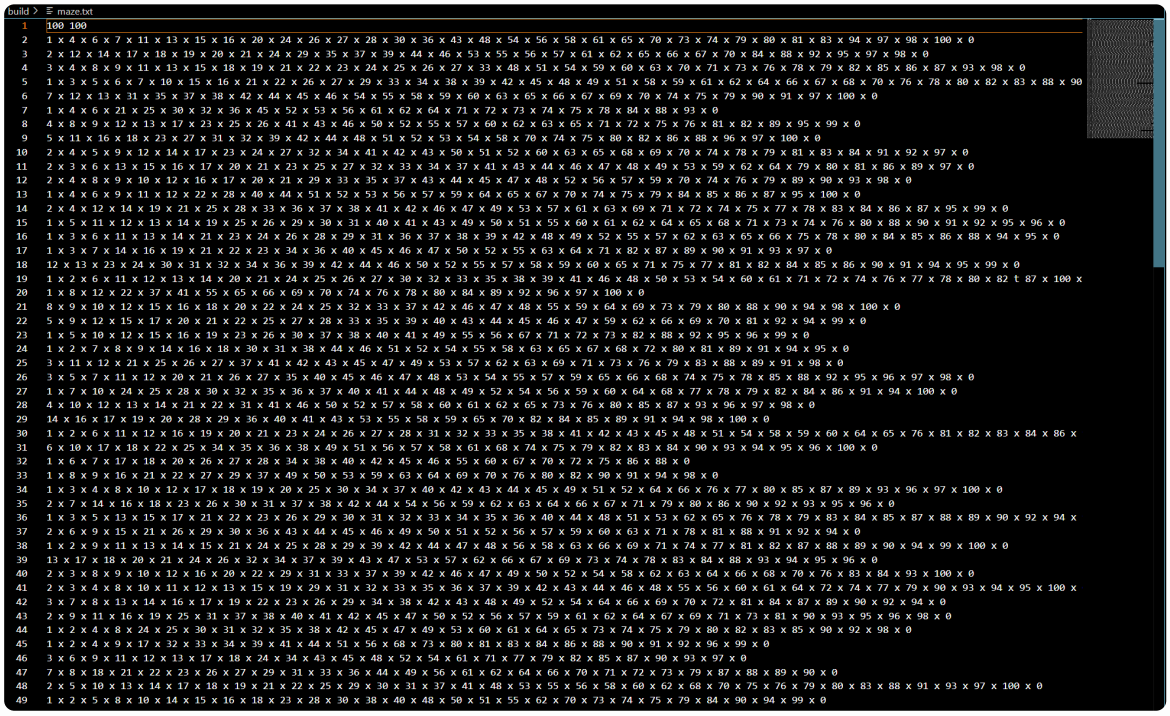
- input:

```
Choice: 2
Enter maze dimensions:
Rows: 100
Columns: 100
Obstacle ratio (0.0 - 1.0): 0.3
Generating 100x100 maze with 30% obstacles...
Maze saved to: maze.txt

=== Maze Statistics ===
Size: 100 x 100
Start: (73, 73)
End: (18, 82)
Path length: 0

=== Maze Preview (first 20x20) ===
x 0 0 x 0 x x 0 0 0 x 0 x 0 x x 0 0 0 x
0 x 0 0 0 0 0 0 0 0 0 x 0 x 0 0 x x x x
0 0 x x 0 0 0 x x 0 x 0 x 0 x 0 0 x x 0
x 0 x 0 x x x 0 0 x 0 0 0 0 x x 0 0 0 0
0 0 0 0 0 0 x 0 0 0 0 x x 0 0 0 0 0 0 0
x 0 0 x 0 x 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 x 0 0 0 x x 0 0 x x 0 0 0 x 0 0 0
0 0 0 0 x 0 0 0 0 0 x 0 0 0 0 x 0 x 0 0
0 x 0 x x 0 0 0 x 0 0 x 0 x 0 0 x 0 0 0
0 x x 0 0 x 0 0 0 0 0 0 x 0 x x x 0 0 x
0 x 0 x 0 0 0 x x x 0 x 0 0 0 x x 0 0 x
x 0 0 x 0 x 0 0 x 0 x x 0 0 0 0 0 0 0 0
0 x 0 x 0 0 0 0 0 0 0 x 0 x 0 0 0 0 x 0
x 0 0 0 x 0 0 0 0 0 x x x x 0 0 0 0 x 0
x 0 x 0 0 x 0 0 0 0 x 0 x x 0 0 0 0 0 0
x 0 x 0 0 0 x 0 0 0 0 0 0 x 0 x 0 0 x 0
0 0 0 0 0 0 0 0 0 0 0 x x 0 0 0 0 0 0 0
x x 0 0 0 x 0 0 0 0 x x x x 0 0 0 0 0 x
x 0 0 0 0 0 0 x 0 0 0 x 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 x x x 0 x 0 0 x x 0 x 0 x
... (Full maze saved to file)
```

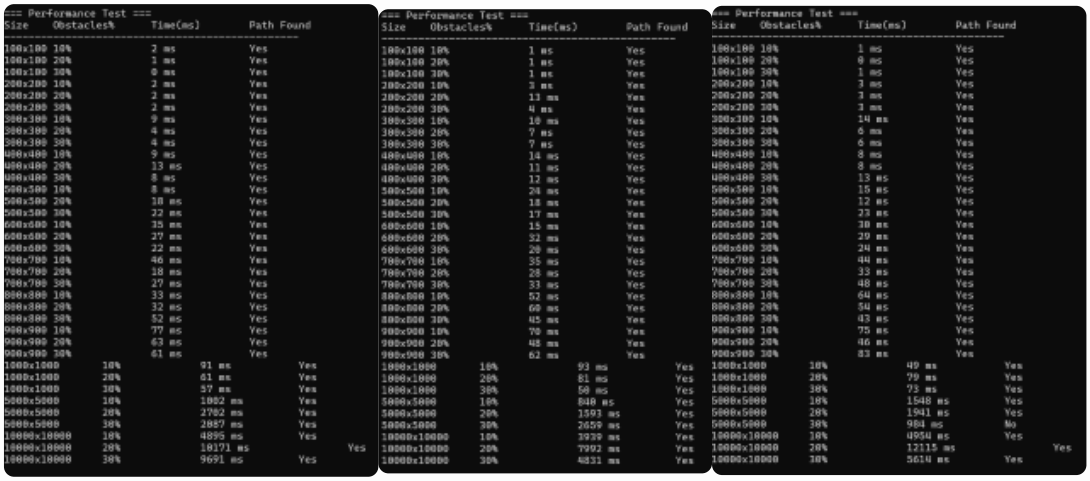
- part of maze:

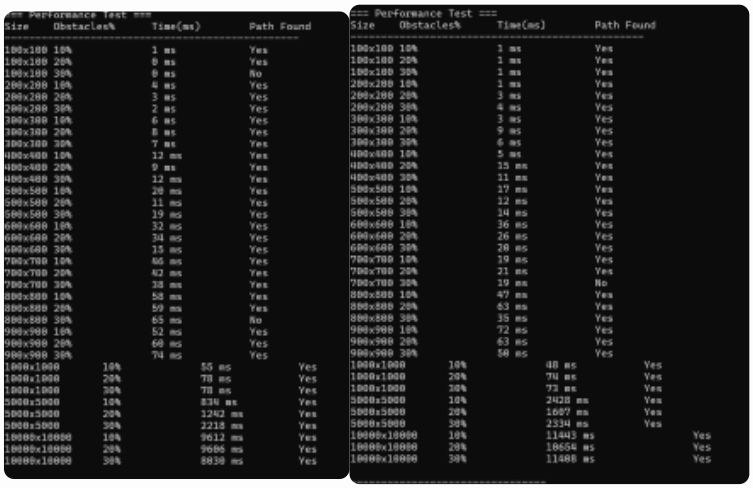


- path:

`(73,73)(73,74)(73,75)(73,76)(73,77)(73,78)(72,78)(71,78)(70,78)(70,79)(69,79)(69,80)(68,80)(67,80)(66,80)(65,80)(65,79)(65,78)(64,78)(63,78)(62,78)(62,79)(61,79)(60,79)(59,79)(59,78)`

3. Performance test





9. 迷宮最短路徑演算法

根據第 8 點 Performance test 的輸出結果 50 筆資料進行分析，提供統計解讀、線性回歸與輸出驗證

95% 信賴區間計算公式

```
標準誤 (SE) = σ / √n
邊際誤差 (ME) = t_{α/2, df} × SE
信賴區間 = μ ± ME
```

其中：

- `σ` = 樣本標準差
- `n` = 樣本大小 (50)
- `t_{0.025, 49}` ≈ 2.01 (t 分布臨界值)
- `μ` = 樣本平均值

變異係數 (Coefficient of Variation)

```
CV = (σ / μ) × 100%
```

線性回歸模型：

```
T = k × (m × n)
k = 平均( T_observed / (m × n) )
```

輸出解讀

- **平均值**：64.53 ms 是 50 次測試的典型執行時間。
- **信賴區間**：真實平均時間有 95% 機率落在 `[59.15 ms, 69.91 ms]`。
- **變異係數**：29.3% 表示中等變異程度。
- **正態性**：數據在 `2σ` 內符合正態分布，信賴區間推論具參考價值。

時間複雜度驗證輸出

觀察：

```
100x100   (10,000 cells)     → 0.69 ms
1000x1000 (1,000,000 cells)  → 64.53 ms
```

比例：

```
64.53 / 0.69 ≈ 93.5 倍
1,000,000 / 10,000 = 100 倍
```

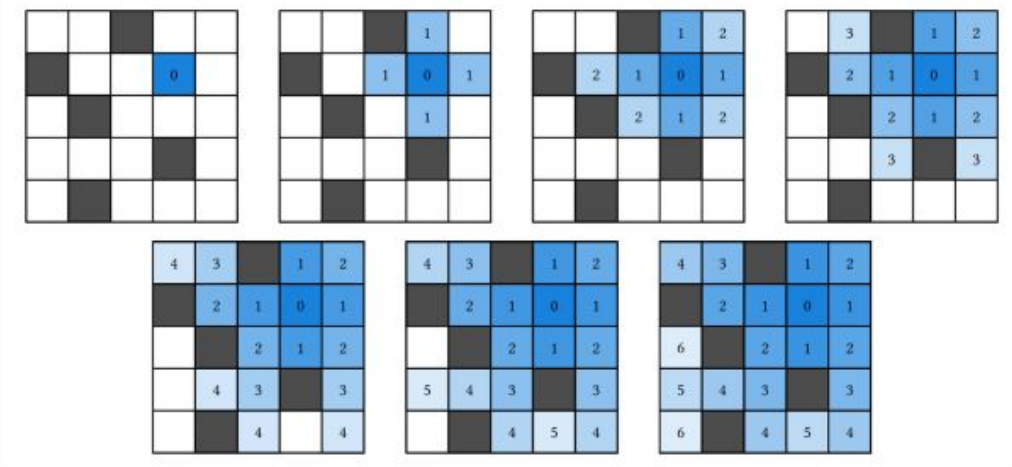
> 時間與單元格數近似線性，符合 `O(mn)` 複雜度。

10. 心得與反思

這次作業從「手動輸入迷宮」到「隨機生成與效能分析」，一開始我先從最基本的全 0 圖出發，思考「怎麼讓使用者方便地輸入障礙與起終點」。

這部分看似簡單，但要讓輸入介面不容易出錯、同時能對應多行輸入，必須先把 grid 的結構設計好。我後來用二維陣列搭配座標結構 (Coord) 來儲存每一格的狀態，這讓後面 BFS 處理時變得很直覺，在設計路徑搜尋的時候，我先從 BFS 偽代碼 入手，理解「同層更新」與「同步擴展」的概念，重點就是「一圈一圈擴展」：從起點開始，每次更新四周的格點，遇到障礙或超出範圍就跳過，當

第一次走到終點 t 時，代表這條路一定是最短的。



並用 queue 來實作，並搭配 parent 陣列記錄每個格點的前一個位置，最後用「逆向回溯」還原整條最短路徑。這段過程讓我對 BFS 的結構與邏輯更熟悉。

若要能「自動產生測資」，做了 generateCustom()，我讓使用者輸入尺寸和障礙比例，然後利用亂數在矩陣中撒點，這裡用的是均勻分布，，更貼近自然迷宮的結構，並在效能測試與信賴區間中，運用了 performanceTest() ，設計了不同尺寸與障礙比例的組合，紀錄每次的執行時間。

為了檢查結果是否合理，我套用了統計學的 信賴區間（Confidence Interval） 計算，並計算變異係數（CV）來觀察穩定度。
實驗結果顯示，時間幾乎與迷宮大小成線性關係（約 O(mn)），這與理論複雜度一致。

在側資中，在測試多組資料後發現，BFS 在障礙多的迷宮中會提早結束（因為死路多），在稀疏迷宮中則會走完整張圖，說明了時間複雜度雖固定為 O(mn)，但實際平均耗時會隨障礙密度變化。

所以在延伸部分，我從 BFS 迷宮想到組合數學中的
多米諾鋪磚理論（Kasteleyn, Temperley, Fisher）與計算複雜度中的 哈密頓路徑問題
BFS 雖然能在多項式時間內找到「最短路」，但如果要找「訪問所有空格一次的路徑」（即哈密頓路徑），那就是 NP 完全問題，說明迷宮問題在找路同時，「路徑的性質」與「可計算性」帶障礙的網格中，最短路可「多項式求解」，但完美覆蓋或全局最優路徑卻屬於更高複雜度的範疇。

11. 附錄

```cpp
#include <bits/stdc++.h>
#include <random>
#include <fstream>
using namespace std;

struct Pos {

    int r, c;
    bool operator==(const Pos& o) const { return r == o.r && c == o.c; }

};

struct Maze {

    vector<vector<char>> grid;
    Pos start{-1,-1}, end{-1,-1};
    bool hasStart = false, hasEnd = false;

};

Maze generate(int rows, int cols, double obstacleRatio = 0.2) {
    Maze maze;
    maze.grid.assign(rows, vector<char>(cols, '0'));

    random_device rd;
    mt19937 gen(rd());
    uniform_real_distribution<double> dis(0.0, 1.0);

    // Generate obstacles
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            if (dis(gen) < obstacleRatio) {
                maze.grid[i][j] = 'x';
            }
        }
    }

    uniform_int_distribution<int> rowDist(0, rows-1);
    uniform_int_distribution<int> colDist(0, cols-1);

    // Select start position
    int attempts = 0;
    while (attempts < 1000) {
        int r = rowDist(gen), c = colDist(gen);
        if (maze.grid[r][c] == '0') {
            maze.grid[r][c] = 's';
            maze.start = {r+1, c+1};
            maze.hasStart = true;
            break;
        }
        attempts++;
    }

    // Select end position
    attempts = 0;
    while (attempts < 1000) {
        int r = rowDist(gen), c = colDist(gen);
        if (maze.grid[r][c] == '0' && !(r+1 == maze.start.r && c+1 == maze.start.c)) {
            maze.grid[r][c] = 'e';
            maze.end = {r+1, c+1};
            maze.hasEnd = true;
            break;
        }
        attempts++;
    }

    return maze;
}
```

```cpp
Maze generateCustom() {
    int rows, cols;
    double obstacleRatio;

    cout << "Enter maze dimensions:" << endl;
    cout << "Rows: "; cin >> rows;
    cout << "Columns: "; cin >> cols;
    cout << "Obstacle ratio (0.0 - 1.0): "; cin >> obstacleRatio;

    // Clear input buffer
    cin.ignore(numeric_limits<streamsize>::max(), '\n');

    if (rows <= 0 || cols <= 0) {
        cout << "Invalid dimensions. Using default 100x100." << endl;
        rows = 100; cols = 100;
    }

    if (obstacleRatio < 0.0 || obstacleRatio > 1.0) {
        cout << "Invalid obstacle ratio. Using default 0.25." << endl;
        obstacleRatio = 0.25;
    }

    cout << "Generating " << rows << "x" << cols << " maze with "
         << (obstacleRatio * 100) << "% obstacles..." << endl;

    return generate(rows, cols, obstacleRatio);
}
```

```cpp
void saveToFile(const vector<vector<char>>& grid, const string& filename) {
    ofstream file(filename);
    if (!file.is_open()) {
        cout << "Cannot create file: " << filename << endl;
        return;
    }

    int rows = grid.size(), cols = grid[0].size();
    file << rows << " " << cols << endl;

    for (int i = 0; i < rows; ++i) {
        bool hasNonZero = false;
        for (int j = 0; j < cols; ++j) {
            if (grid[i][j] != '0') {
                if (hasNonZero) file << " ";
                file << (j + 1) << " " << grid[i][j];
                hasNonZero = true;
            }
        }
        if (!hasNonZero) {
            file << "0";
        } else {
            file << " 0";
        }
        file << endl;
    }
    file.close();
    cout << "Maze saved to: " << filename << endl;
}
```

```cpp
void savePathToFile(const vector<Pos>& path, const string& filename) {
    ofstream file(filename);
    if (!file.is_open()) {
        cout << "Cannot create file: " << filename << endl;
        return;
    }

    for (const auto& pos : path) {
        file << "(" << pos.r << "," << pos.c << ")";
    }
    file << endl;
    file.close();
    cout << "Path saved to: " << filename << endl;
}

// tool
string trim(string s) {
    size_t start = s.find_first_not_of(" \t"), end = s.find_last_not_of(" \t");
    return start == string::npos ? "" : s.substr(start, end - start + 1);
}

vector<string> split(string s) {
    replace(s.begin(), s.end(), ',', ' ');
    stringstream ss(s);
    vector<string> tokens;
    for (string token; ss >> token;) tokens.push_back(token);
    return tokens;
}
```

```
pair<int,int> getSize() {
    while (true) {
        cout << "Enter maze size (rows cols): " << flush;
        string line; getline(cin, line);
        auto tokens = split(trim(line));
        if (tokens.size() != 2) {
            cout << "Please enter two positive integers.\n";
            continue;
        }
        try {
            int r = stoi(tokens[0]), c = stoi(tokens[1]);
            if (r > 0 && c > 0) return {r, c};
        } catch (...) {}
        cout << "Invalid input.\n";
    }
}

// Least path
bool isValid(int r, int c, int R, int C) {
    return r >= 1 && r <= R && c >= 1 && c <= C;
}
```

```
vector<Pos> findPath(const vector<vector<char>>& grid, Pos start, Pos end) {
    int R = grid.size(), C = grid[0].size();
    int dr[] = {-1, 1, 0, 0}, dc[] = {0, 0, -1, 1};

    vector<vector<bool>> visited(R+1, vector<bool>(C+1, false));
    vector<vector<Pos>> parent(R+1, vector<Pos>(C+1, {-1,-1}));
    queue<Pos> q;

    q.push(start);
    visited[start.r][start.c] = true;

    while (!q.empty()) {
        Pos cur = q.front(); q.pop();
        if (cur == end) break;

        for (int i = 0; i < 4; ++i) {
            int nr = cur.r + dr[i], nc = cur.c + dc[i];
            if (isValid(nr, nc, R, C) && grid[nr-1][nc-1] != 'x' && !visited[nr][nc]) {
                visited[nr][nc] = true;
                parent[nr][nc] = cur;
                q.push({nr, nc});
            }
        }
    }

    if (!visited[end.r][end.c]) return {};

    vector<Pos> path;
    for (Pos at = end; !(at == start); at = parent[at.r][at.c])
        path.push_back(at);
    path.push_back(start);
    reverse(path.begin(), path.end());
    return path;
}
```

```
// visualize output
void visualize(const Maze& maze, const vector<Pos>& path = {}) {
    int rows = maze.grid.size(), cols = maze.grid[0].size();

    vector<vector<char>> display = maze.grid;

    for (size_t i = 1; i + 1 < path.size(); ++i) {
        int r = path[i].r - 1, c = path[i].c - 1;
        if (display[r][c] == '0') {
            display[r][c] = '.';
        }
    }

    cout << "\n=== Maze Statistics ===" << endl;
    cout << "Size: " << rows << " x " << cols << endl;
    cout << "Start: (" << maze.start.r << ", " << maze.start.c << ")" << endl;
    cout << "End: (" << maze.end.r << ", " << maze.end.c << ")" << endl;
    cout << "Path length: " << (path.empty() ? 0 : path.size()) << endl;

    // For large mazes, show partial preview
    if (rows > 20 || cols > 20) {
        cout << "\n=== Maze Preview (first 20x20) ===" << endl;
        int previewRows = min(20, rows);
        int previewCols = min(20, cols);

        for (int i = 0; i < previewRows; ++i) {
            for (int j = 0; j < previewCols; ++j) {
                cout << display[i][j] << " ";
            }
            cout << endl;
        }
        cout << "... (Full maze saved to file)" << endl;
    } else {
        cout << "\n=== Complete Maze ===" << endl;
        for (const auto& row : display) {
            for (char cell : row) {
                cout << cell << " ";
            }
            cout << endl;
        }
    }
```

```
void performanceTest() {
    vector<int> sizes = {100, 200, 500, 400, 500, 600, 700, 800, 900, 1000, 5000, 10000};
    vector<double> obstacles = {0.1, 0.2, 0.3};

    cout << "\n=== Performance Test ===" << endl;
    cout << "Size\tObstacles%\tTime(ms)\tPath Found" << endl;
    cout << "-----------------------------------------------" << endl;

    for (int size : sizes) {
        for (double obstacleRatio : obstacles) {
            auto startTime = chrono::high_resolution_clock::now();

            Maze maze = generate(size, size, obstacleRatio);
            vector<Pos> path;
            if (maze.hasStart && maze.hasEnd) {
                path = findPath(maze.grid, maze.start, maze.end);
            }

            auto endTime = chrono::high_resolution_clock::now();
            auto duration = chrono::duration_cast<chrono::milliseconds>(endTime - startTime);

            cout << size << "x" << size << "\t" << (obstacleRatio * 100)
                 << "%\t\t" << duration.count() << " ms\t\t"
                 << (path.empty() ? "No" : "Yes") << endl;

            // Stop if it takes too long
            if (duration.count() > 60000) { // 1 minute
                cout << "Stopping test - taking too long..." << endl;
                return;
            }
        }
    }
}
```