Project 3
Data Structures and Algorithms

# Report

**Team Name:** Group 189, Grease Police
**Team Members:** James Shen, Hunter Brinkman
**GitHub URL:**
**Video Link:**

**Project Title:** Going the Distance (Maze Pun)

**Problem:** Investigate the difference between A* and BFS implementations for maze-solving.

**Motivation:** There are many ways to solve mazes and we want to see how we can apply mazes learned in this class in a real-life puzzle such as mazes. Using the raw breadth-first-search algorithm learned during the trees and graphs unit, and implementing a C++ version of the A* pseudocode algorithm found on (https://en.wikipedia.org/wiki/A*_search_algorithm), we thought this project would be a great introduction to the deep treasure trove of advanced pathfinding algorithms. A* is just the beginning.

**Features:** We have a console-based step-by-step interactive stdout display of the individual steps of the A* and BFS algorithms. This feature is invaluable for learning the difference between various pathfinding algorithms, as visualization makes code easier to follow.

**Data:** The input is a cartesian plane in matrix of cells format, with 0 meaning the cell is passable, 1 meaning there is a wall, making the cell impassable. The matrix of cells can be represented by a 2D vector in C++. The grid sizes are 20x20, as 20x20 is a convenient size to fit into stdout.

**Tools:** We will be coding using C++. The IDE of choice is Notepad++ and Visual Studio.

**Visuals:** Visuals will be created using stdout.

**Strategy:** We had a few ideas regarding maze-solving:

i. Breadth-first search (BFS)
ii. Depth-first search (DFS)
iii. Depth-limited Depth-first-search (DLDFS)
iv. Topological dead-end filling
v. A*

We narrowed our choices down to BFS and A*, as BFS is the simplest, and A* is the most advanced of the choices.

**Division of Responsibility and Roles:**

James will be the primary coder. Hunter will be the tertiary coder.

Project 3
Data Structures and Algorithms

**Analysis**

We made a few changes after the initial submission of the proposal. We initially aimed to use the BRIDGES API for graphics, but figured that stdout would be more interactive, and simpler to use for the user.

We added a new feature, called wallFactor, to customize the makeup of the initial grid (to increase or decrease the number of walls in the randomly-generated grids).

Time Complexity Investigations:

i. Function reverse(path): O(n) worst case time, with n = path length.

ii. Function reconstruct_path(goal): O(n) worst case time, with n = path length. We retrospectively find the previous cell from which we came from, and build the path in reverse order.

iii. Function h(): O(1). This is the heuristic helper function for A*, providing the minimum possible distance from current node to the goal. This function performs a constant number of abs(), subtracts, and additions, each of which is O(1).

iv. Function getKey(x, y): O(1). This converts the {x,y} coordinates into a single long long which is easily hashable as a key in an unordered_set.

v. Function BFS(grid): O(m*n), where m is #rows, and n is #cols. Worst case BFS finds/does not find a path in a long, winding grid, searching through every single cell in the grid.

vi. Function A*(grid): O(|E|*log(|V|)). The logarithmic time complexity comes from the minheap used in the algorithm. A* is essentially Dijkstra's with an added heuristic function.

vii. Function generateGrid(): O(m^2 * n^2). To generate the grid, a random permutation of each of the m * n coordinates in the grid is generated, and the first K cells (K = wallFactor * m * n) are chosen as the coordinates of the walls. The random permutation is generated by successively deleting coordinates from the list of coordinates, each delete takes O(m*n) time, and we delete m*n times.

**Reflection**

As a group, the overall experience of the project was a resoundingly positive one. The maze topic was highly engaging. The challenges were in the implementation of the A* algorithm. Even though we used the pseudocode as a guideline, understanding the code and debugging our C++ implementation was tricky (the pseudocode was slightly vague and abstracted).

Project 3
Data Structures and Algorithms

If we redid the project, we would investigate other variables, such as grid size, the effect of wallFactor on the probability that a valid path exists from start (1, 1) to finish (m-2, n-2), and more pathfinding algorithms.

Each of us learned how to program interactively in a stdout environment. Moreover, we learned how to work collaboratively, in a timely manner, via incremental changes. Specifically, my partner resolved the BFS implementation bug where the same cell is pushed twice to the next iteration of the BFS queue in the same iteration. Each cell should only appear in the queue a SINGLE TIME.

Source Code:

```
#include <bits/stdc++.h>

#define ROWS 20
#define COLS 20
#define WALL_FACTOR 0.10

// include SFML for visualization?

using namespace std;


// do A* for one step, do BFS for one step, see which one finishes first

vector<vector<int>> reverse(vector<vector<int>>& path){
        vector<vector<int>> rev;
        for(int i = path.size() - 1; i >= 0; --i){
                rev.push_back(path[i]);
        }
        return rev;
}

vector<vector<int>> reconstruct_path(vector<vector<vector<int>>>& cameFrom, vector<int>
current){
   vector<vector<int>> total_path = {current};
   while(cameFrom[current[0]][current[1]].size() > 0){
                current = cameFrom[current[0]][current[1]];
                total_path.push_back(current);
        }
        total_path = reverse(total_path);
   return total_path;
}
// h is the heuristic function. h(n) estimates the cost to reach goal from node n.

int h(vector<int> node, vector<vector<int>>& grid){ // Manhattan distance
```

```
        int target_x = (int)grid.size() - 2;
        int target_y = (int)grid[0].size() - 2;
        int x = node[0];
        int y = node[1];
        return abs(x - target_x) + abs(y - target_y);
}

int getKey(int x, int y){
        return x * pow(10, (int)floor(log10(y)) + 1) + y;
}

// add a searched bool field to A_Star so we can print out which cells have been
searched/unsearched, like in the BFS
vector<vector<int>> A_Star(vector<int> start, vector<int> goal, vector<vector<int>>& grid){

        int start_x = start[0];
        int start_y = start[1];
        int goal_x = goal[0];
        int goal_y = goal[1];
        if(grid[start_x][start_y] == 1 || grid[goal_x][goal_y] == 1) return {}; // if wall at start or
finish locations
        int m = grid.size();
        int n = grid[0].size();
   // For node n, cameFrom[n] is the node immediately preceding it on the cheapest path from the
start
   // to n currently known.
   //cameFrom := an empty map
        vector<vector<vector<int>>> cameFrom (m, vector<vector<int>> (n, vector<int>() ) );

   // For node n, gScore[n] is the cost of the cheapest path from start to n currently known.
        //gScore := map with default value of Infinity
        //gScore[start] := 0
        vector<vector<int>> gScore (m, vector<int> (n, INT_MAX ) );
   gScore[start_x][start_y] = 0;

   // For node n, fScore[n] := gScore[n] + h(n). fScore[n] represents our current best guess as to
   // how cheap a path could be from start to finish if it goes through n.
   //fScore := map with default value of Infinity
   //fScore[start] := h(start)
        vector<vector<int>> fScore (m, vector<int> (n, INT_MAX ) );
        fScore[start_x][start_y] = h(start, grid);

         // The set of discovered nodes that may need to be (re-)expanded.
   // Initially, only the start node is known.
   // This is usually implemented as a min-heap or priority queue rather than a hash-set.
        // openSet := {start}
```

```
priority_queue<vector<int>, vector<vector<int>>, greater<vector<int>>> pq;
        unordered_set<int> openSet; // shadows pq
        pq.push({fScore[start_x][start_y], start_x, start_y});
        openSet.insert(getKey(start_x, start_y));

        vector<vector<int>> dirs = {{-1, 0}, {0, 1}, {1, 0}, {0, -1}};

    while(pq.size() > 0){
      // This operation can occur in O(Log(N)) time if openSet is a min-heap or a priority queue
      //current := the node in openSet having the lowest fScore[] value
                int x = pq.top()[1];
                int y = pq.top()[2];
                vector<int> current = {x, y};
      //if current = goal
      //    return reconstruct_path(cameFrom, current)
                if(current == goal)
                        return reconstruct_path(cameFrom, current);
      pq.pop();
                openSet.erase(getKey(x, y));
      //for each neighbor of current
                for(vector<int>& dir : dirs){
        // d(current,neighbor) is the weight of the edge from current to neighbor
        // tentative_gScore is the distance from start to the neighbor through current
                        // unweighted graph, each edge has edge length 1
        //tentative_gScore := gScore[current] + d(current, neighbor)
                        int nx = x + dir[0];
                        int ny = y + dir[1]; // neighbor x, neighbor y
                        if(grid[nx][ny] == 1) continue; // wall
                        vector<int> neighbor = {nx, ny};
                        int tentative_gScore = gScore[x][y] + 1;
        //if tentative_gScore < gScore[neighbor]
                        if(tentative_gScore < gScore[nx][ny]){
          // This path to neighbor is better than any previous one. Record it!
          //cameFrom[neighbor] := current
          //gScore[neighbor] := tentative_gScore
          //fScore[neighbor] := tentative_gScore + h(neighbor, grid)
          //if neighbor not in openSet
          //    openSet.add(neighbor)
                                cameFrom[nx][ny] = current;
                                gScore[nx][ny] = tentative_gScore;
                                fScore[nx][ny] = tentative_gScore + h(neighbor, grid);
                                if(openSet.count(getKey(nx, ny)) == false){
                                        openSet.insert(getKey(nx, ny));
                                        pq.push({fScore[nx][ny], nx, ny});
                                }
                        }
```

```
                }
        }
    // Open set is empty but goal was never reached
    //return failure
        return {};
}


vector<vector<int>> BFS(vector<vector<int>>& grid){ // start at (1, 1), go until (m - 2, n - 2)

        // edge case: no valid path exists: return empty vector {}
        // check if start cell or finish cell has a wall, impossible in that case

        int m = grid.size();
        int n = grid[0].size();
        if(grid[1][1] == 1 || grid[m - 2][n - 2] == 1) return {};

        vector<vector<vector<int>>> prev (m, vector<vector<int>> (n, vector<int>() ) );
        vector<int> start = {1, 1};
        vector<vector<int>> bfsv = {start}; // bfs vector, "queue"-esque
        vector<vector<int>> searched (m, vector<int> (n, 0));
        searched[1][1] = 1; // mark root starting cell
        for(int i = 0; i < m; ++i){
                for(int j = 0; j < n; ++j){
                        if(grid[i][j] == 1) searched[i][j] = 2; // mark walls as already searched,
walls are 2, searched cell is 1, unsearched 0
                }
        }
        vector<vector<int>> dirs = {{-1, 0}, {0, 1}, {1, 0}, {0, -1}}; // north east south west
        while(bfsv.size() > 0){
                vector<vector<int>> next_bfsv;
                for(vector<int>& p : bfsv){
                        for(vector<int>& dir : dirs){
                                int x = p[0] + dir[0];
                                int y = p[1] + dir[1];
                                if(searched[x][y] == 0){
                                        searched[x][y] = 1;
                                        next_bfsv.push_back({x, y});
                                        prev[x][y] = p;
                                }
                        }
                }
                bfsv = next_bfsv;
        }
        if(searched[m - 2][n - 2] == 0) return {}; // final cell not reachable
        vector<vector<int>> path = {{m - 2, n - 2}};
```

```
        int x = m - 2;
        int y = n - 2;
        while(prev[x][y] != start){
                path.push_back(prev[x][y]);
                x = path.back()[0];
                y = path.back()[1];
        }
        path.push_back(start);
        // is this slower than just initializing new
        // reverse path, right now in finish -> start order, I want start -> finish order
        //int bound = path.size() / 2;
        //for(int i = 0; i < bound; ++i){
        //        vector<int> temp = path[i];
        //        path[i] = path[path.size() - 1 - i];
        //        path[path.size() - 1 - i] = temp;
        //}
        path = reverse(path);
        return path;
}

int myrandom (int i){
        return std::rand()%i;
}

vector<vector<int>> myshuffle(vector<vector<int>> v){
        vector<vector<int>> shuffled;
        srand(time(NULL));
        while(v.size() > 0){
                int gap = rand() % 7;
                for(int i = 0; i < gap; ++i) rand();
                int i = rand() % (int)v.size();
                shuffled.push_back(v[i]);
                v.erase(v.begin() + i);
        }
        return shuffled;
}

vector<vector<int>> generateGrid(int m, int n, double wallFactor){
        vector<vector<int>> grid (m, vector<int> (n, 0)); // 0 means passable, 1 means wall
        // top border
        for(int j = 0; j < n; ++j){
                grid[0][j] = 1;     // top border
                grid[m - 1][j] = 1; // bottom border
        }
        for(int i = 0; i < m; ++i){
                grid[i][0] = 1;    // left border
```

```
                grid[i][n - 1] = 1;//right border
        }
        vector<vector<int>> v;
        for(int i = 1; i < m - 1; ++i){
                for(int j = 1; j < n - 1; ++j){
                        v.push_back({i, j});
                }
        }
        //random_device rd;
        //mt19937 g(rd());
        //random_shuffle(v.begin(), v.end(), myrandom); // generate random permutation of
points, select first wallFactor * m * n coordinates to place wall
        //shuffle(v.begin(), v.end(), g);
        v = myshuffle(v);
        int k = (int)floor(wallFactor * m * n);
        for(int i = 0; i < k; ++i){
                int x = v[i][0];
                int y = v[i][1];
                grid[x][y] = 1;
        }
        return grid;
}

void printPath(vector<vector<int>>& path){
        cout << '{';
        for(int i = 0; i < path.size(); ++i){
                cout << '{' << path[i][0] << ',' << path[i][1] << '}';
                if(i != (int)path.size() - 1) cout << ',';
        }
        cout << endl << endl;
}

void printGrid(vector<vector<int>>& grid){
        cout << "grid:" << endl;
        for(int i = 0; i < grid.size(); ++i){
                for(int j = 0; j < grid[0].size(); ++j){
                        cout << grid[i][j];
                }
                cout << endl;
        }
        cout << endl;
}


void printMaze(vector<vector<int>>& maze) {
   string cur = "";
```

```
    for (int i = 0; i < maze.size(); i++) {
        for (int j = 0; j < maze[i].size(); j++) {
            if (maze[i][j] == 0) {
                cur += ". ";
            }
            else if (maze[i][j] == 1) {
                cur += "# ";
            }
            else if (maze[i][j] == 2) {
                cur += "* ";
            }
        }
        cur += "\n";
    }
    cout << cur << endl;
}

void updateMazeBfs(vector<vector<int>>& maze, vector<vector<int>>& visited,
queue<vector<int>>& q) {
    int qSize = q.size();
    vector<vector<int>> dirs = { {1, 0}, {0, 1}, {-1, 0}, {0, -1} };
    for (int i = 0; i < qSize; i++) {
        auto cur = q.front();
        maze[cur[0]][cur[1]] = 2;
        q.pop();
        if (cur[0] == ROWS - 2 && cur[1] == COLS - 2) {
            return;
        }
        for (auto& d : dirs) {
            int x = cur[0] + d[0], y = cur[1] + d[1];
            if (x > 0 && x < maze.size() - 1 && y > 0 && y < maze[0].size() - 1 && maze[x][y] ==
0 && visited[x][y] == 0) {
                q.push({ x, y });
                visited[x][y] = 1;
            }
        }
    }
}

void updateA_star(vector<vector<int>>& maze2, vector<vector<int>>& visited2,
priority_queue<vector<int>, vector<vector<int>>,
                                        greater<vector<int>>>& pq, vector<vector<int>>& fScore,
vector<vector<int>>& gScore, unordered_set<int>& openSet,
                                        vector<vector<vector<int>>>& cameFrom){
        vector<vector<int>> dirs = {{-1, 0}, {0, 1}, {1, 0}, {0, -1}};
```

```
        // This operation can occur in O(Log(N)) time if openSet is a min-heap or a priority
queue
        //current := the node in openSet having the lowest fScore[] value
        int x = pq.top()[1];
        int y = pq.top()[2];
        vector<int> current = {x, y};
        pq.pop();
        openSet.erase(getKey(x, y));
        //for each neighbor of current
        for(vector<int>& dir : dirs){
                // d(current,neighbor) is the weight of the edge from current to neighbor
                // tentative_gScore is the distance from start to the neighbor through current
                // unweighted graph, each edge has edge length 1
                //tentative_gScore := gScore[current] + d(current, neighbor)
                int nx = x + dir[0];
                int ny = y + dir[1]; // neighbor x, neighbor y
                if(maze2[nx][ny] == 1) continue; // wall
                vector<int> neighbor = {nx, ny};
                int tentative_gScore = gScore[x][y] + 1;
                //if tentative_gScore < gScore[neighbor]
                if(tentative_gScore < gScore[nx][ny]){
                        // This path to neighbor is better than any previous one. Record it!
                        //cameFrom[neighbor] := current
                        //gScore[neighbor] := tentative_gScore
                        //fScore[neighbor] := tentative_gScore + h(neighbor, grid)
                        //if neighbor not in openSet
                        //    openSet.add(neighbor)
                        cameFrom[nx][ny] = current;
                        gScore[nx][ny] = tentative_gScore;
                        //fScore[nx][ny] = tentative_gScore + h(neighbor, grid);
                        fScore[nx][ny] = tentative_gScore + h(neighbor, maze2);
                        if(openSet.count(getKey(nx, ny)) == false){
                                openSet.insert(getKey(nx, ny));
                                pq.push({fScore[nx][ny], nx, ny});
                                visited2[nx][ny] = 2;
                        }
                }
        }
}

int main()
{
        // use BFS, calculate shortest path, use A*, calculate shortest path, investigate
        // average runtime of each, check if they produce same shortest path value.
        // play around with randomized number of "wall" cells. Default 20x20 freespace (if
include boundary outer wall, 22x22)
```

```
        /// check to see what value to
        // on average have 0.5 success rate of getting from (1, 1) to (m - 2, n - 2) 0-indexed, can binary search this value?
        vector<vector<int>> grid = generateGrid(ROWS + 2, COLS + 2, WALL_FACTOR);
        int m = ROWS;
        int n = COLS;
        printGrid(grid);

        // render this grid in SFML?
        vector<int> start = {1, 1};
        vector<int> goal = {m, n};

        // explore different mxn, see if wallFactor that results in ~half success via Monte Carlo randomly generated grids is
        // some kind of function of mxn?
        vector<vector<int>> path = A_Star(start, goal, grid);
        cout << "A_star path:" << endl;
        printPath(path);
        path = BFS(grid);
        cout << "path.size() " << path.size() << endl;
        cout << "BFS path:" << endl;
        printPath(path);


        vector<vector<int>> maze1 = grid;
        vector<vector<int>> maze2 = grid;
        //vector<int> start = {1, 1};
        //vector<int> goal = {ROWS, COLS};

    vector<vector<int>> visited1 = grid;
        for(int i = 0; i < visited1.size(); ++i){
                for(int j = 0; j < visited1[0].size(); ++j){
                        if(visited1[i][j] == 1) visited1[i][j] = 2;
                }
        }
        //vector<vector<int>> visited2 = visited1;
        vector<vector<int>> visited2 = grid;
        //for(int i = 0; i < visited2.size(); ++i){
        //      for(int j = 0; j < visited2[0].size(); ++j){
        //              if(visited2[i][j] == 1) visite

    queue<vector<int>> q; // BFS queue


        int start_x = start[0];
        int start_y = start[1];
```

```
        int goal_x = goal[0];
        int goal_y = goal[1];
        if(maze2[start_x][start_y] == 1 || maze2[goal_x][goal_y] == 1){  // if wall at start or
finish locations
                cout << "start or exit blocked by wall" << endl;
                return 0;
        }

        //int m = grid.size();
        //int n = grid[0].size();
        vector<vector<vector<int>>> cameFrom (m, vector<vector<int>> (n, vector<int>() ) );
        vector<vector<int>> gScore (m, vector<int> (n, INT_MAX ) );
   gScore[start_x][start_y] = 0;
        vector<vector<int>> fScore (m, vector<int> (n, INT_MAX ) );
        fScore[start_x][start_y] = h(start, grid);
   priority_queue<vector<int>, vector<vector<int>>, greater<vector<int>>> pq; // A* priority
queue
        unordered_set<int> openSet; // shadows pq
        pq.push({fScore[start_x][start_y], start_x, start_y});
        openSet.insert(getKey(start_x, start_y));
   q.push({1, 1});
   visited1[1][1] = 1;
        //visited2[1][1] = 1;
        visited2[1][1] = 2;
   int count1 = 0;
        int count2 = 0;
   while (q.size() || pq.size()) {
      // update maze with bfs call
                if(q.size()){
                        updateMazeBfs(maze1, visited1, q);
                        // clear window to draw updated maze
                        system("cls");

                        ++count1;
                }
                cout << "Iteration " << count1 << endl;
                cout << "\n";
      cout << "Breadth First Search" << endl;
      printMaze(maze1);

                if(pq.size()){
                        updateA_star(maze2, visited2, pq, fScore, gScore, openSet, cameFrom);
                        // call a* and update maze

                        ++count2;
                }
```

```
                //printMaze(maze2);
                cout << "Iteration " << count2 << endl;
                cout << "\n";
                cout << "A* Search" << endl;
                printMaze(visited2);

        string input = "";
        while (input != "\n") {
            cout << "Press Enter to continue" << endl;
            input = cin.get();
        }
    }


        return 0;
}
```