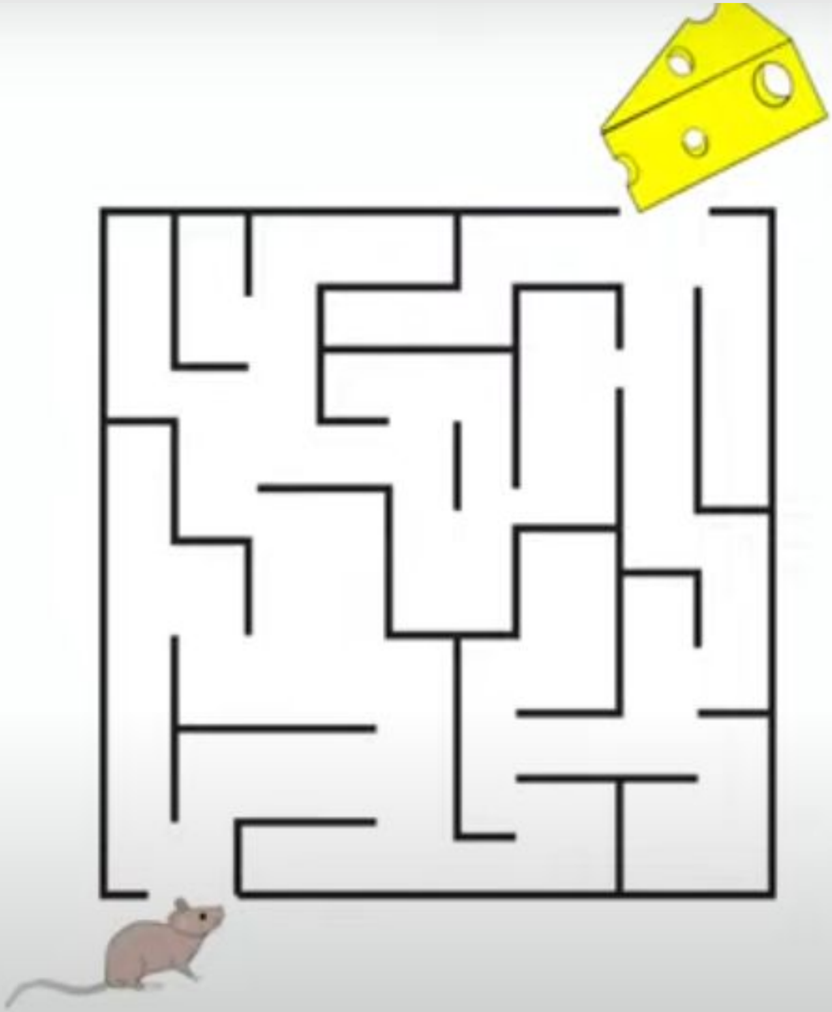
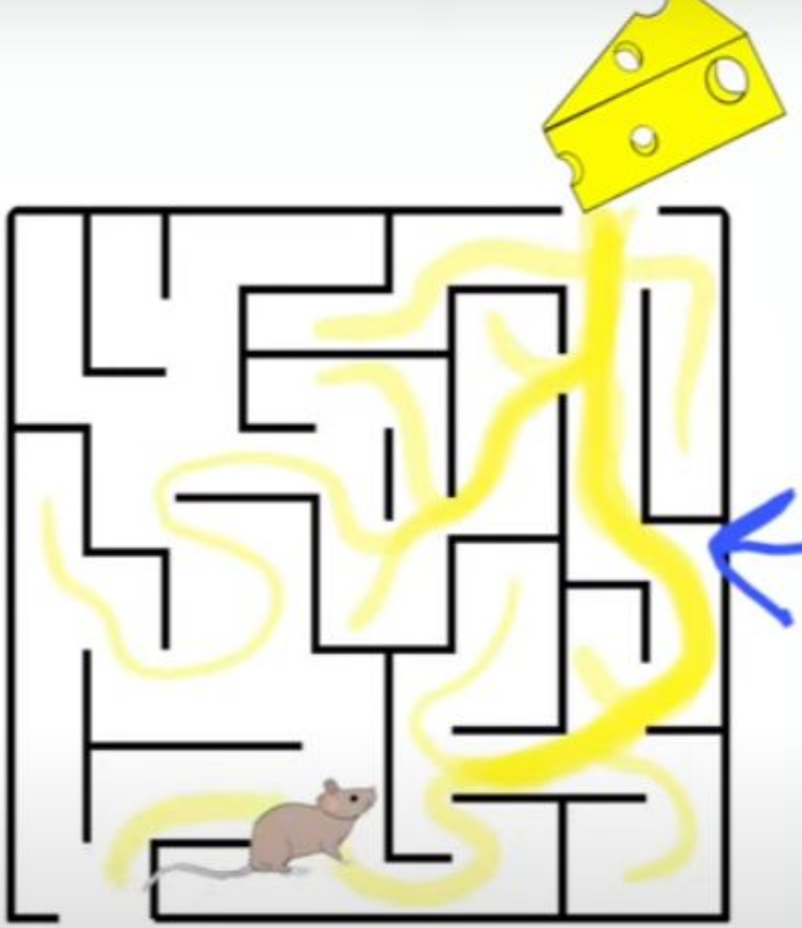
A* Algorithm: Implementation

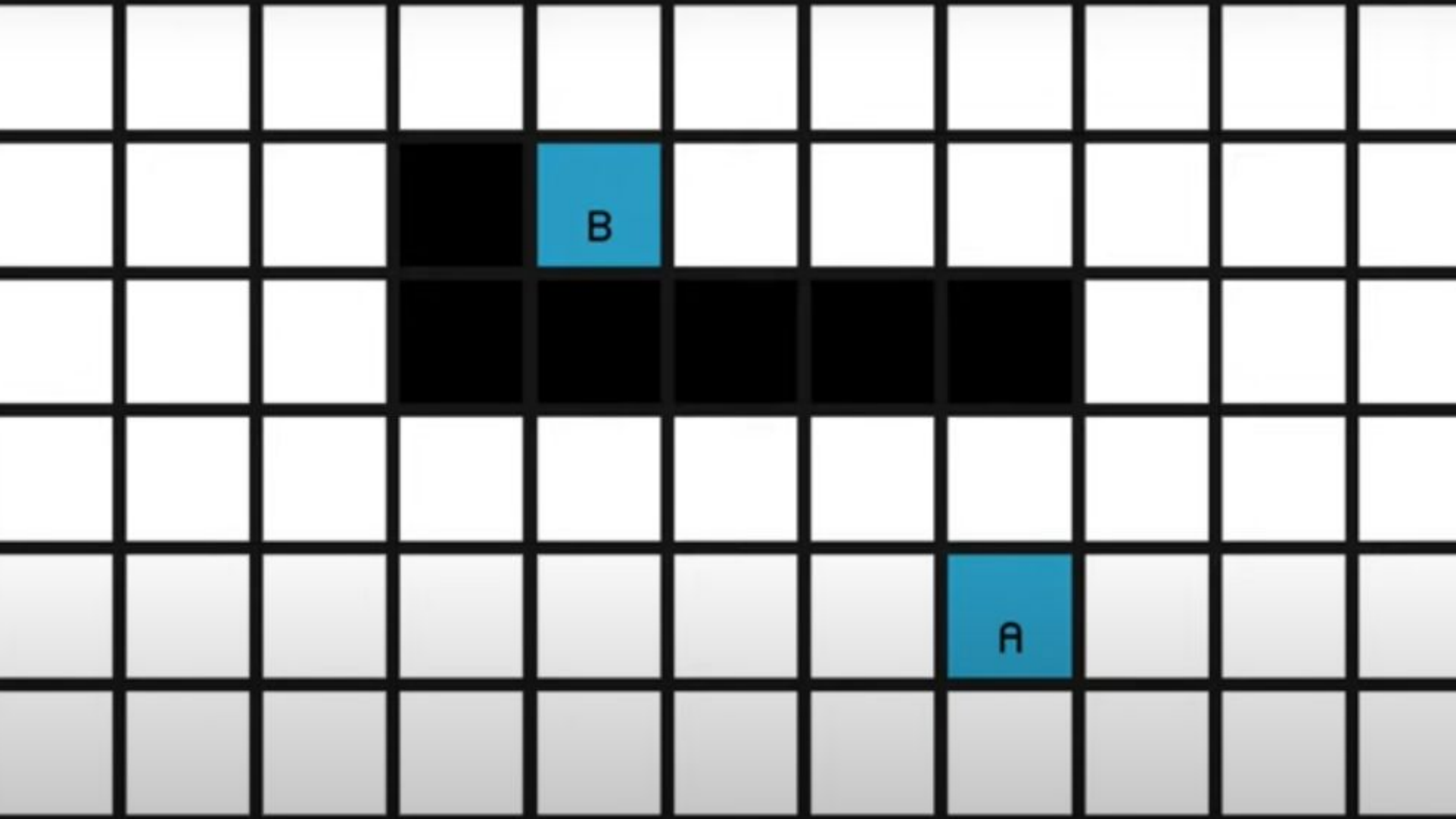
Navigating the Grid

Mohamed Zaghloul
192300513

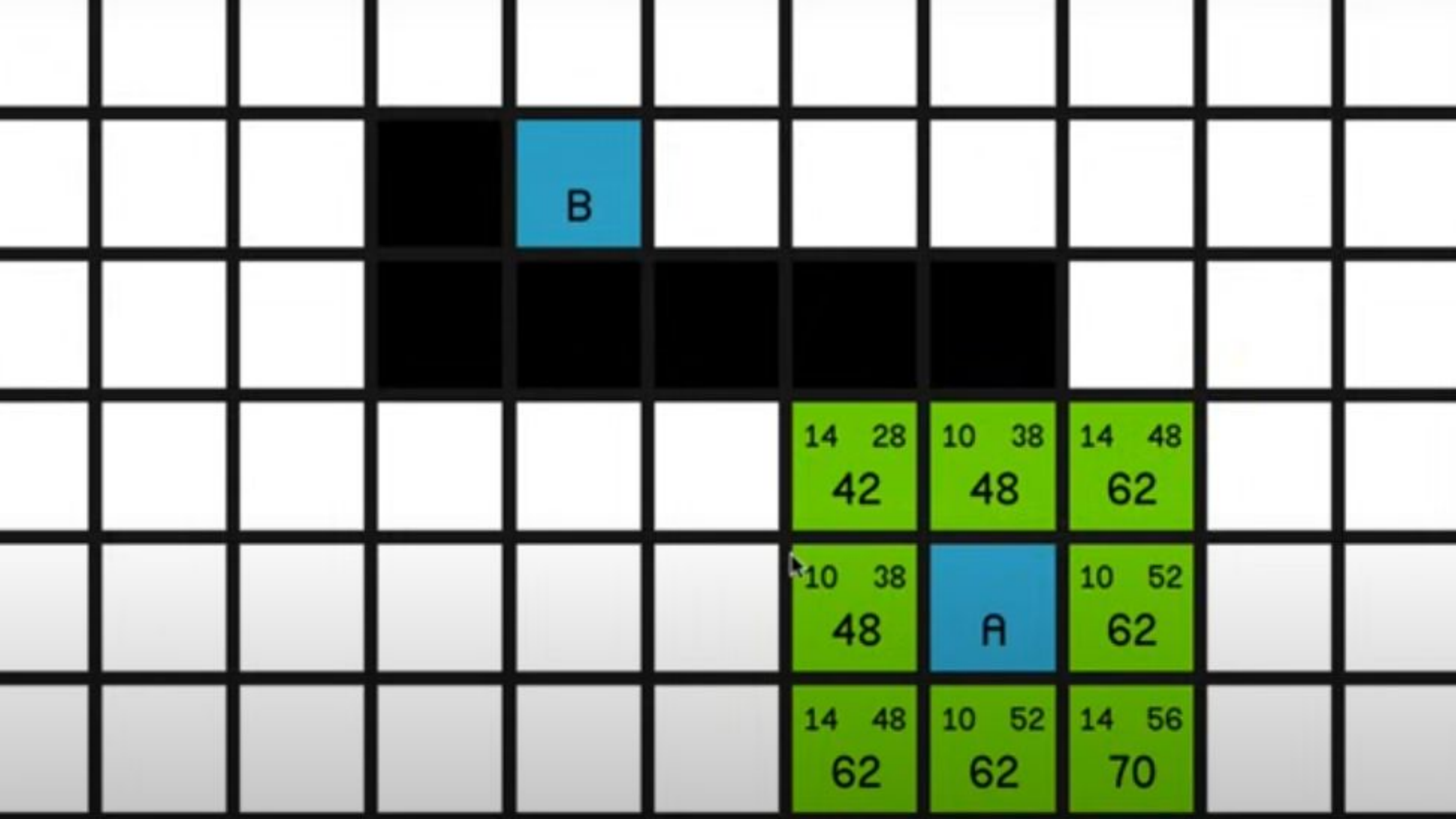


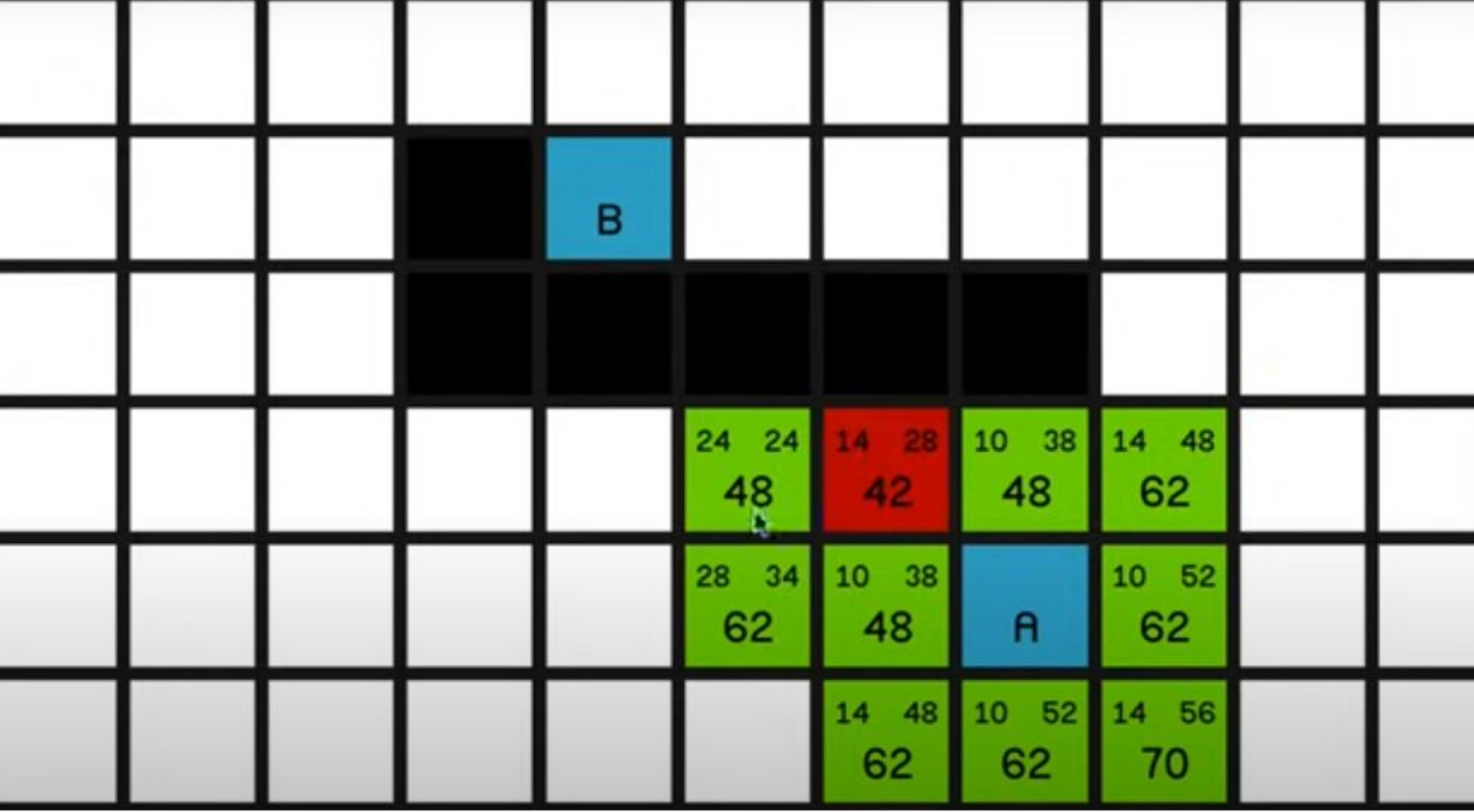


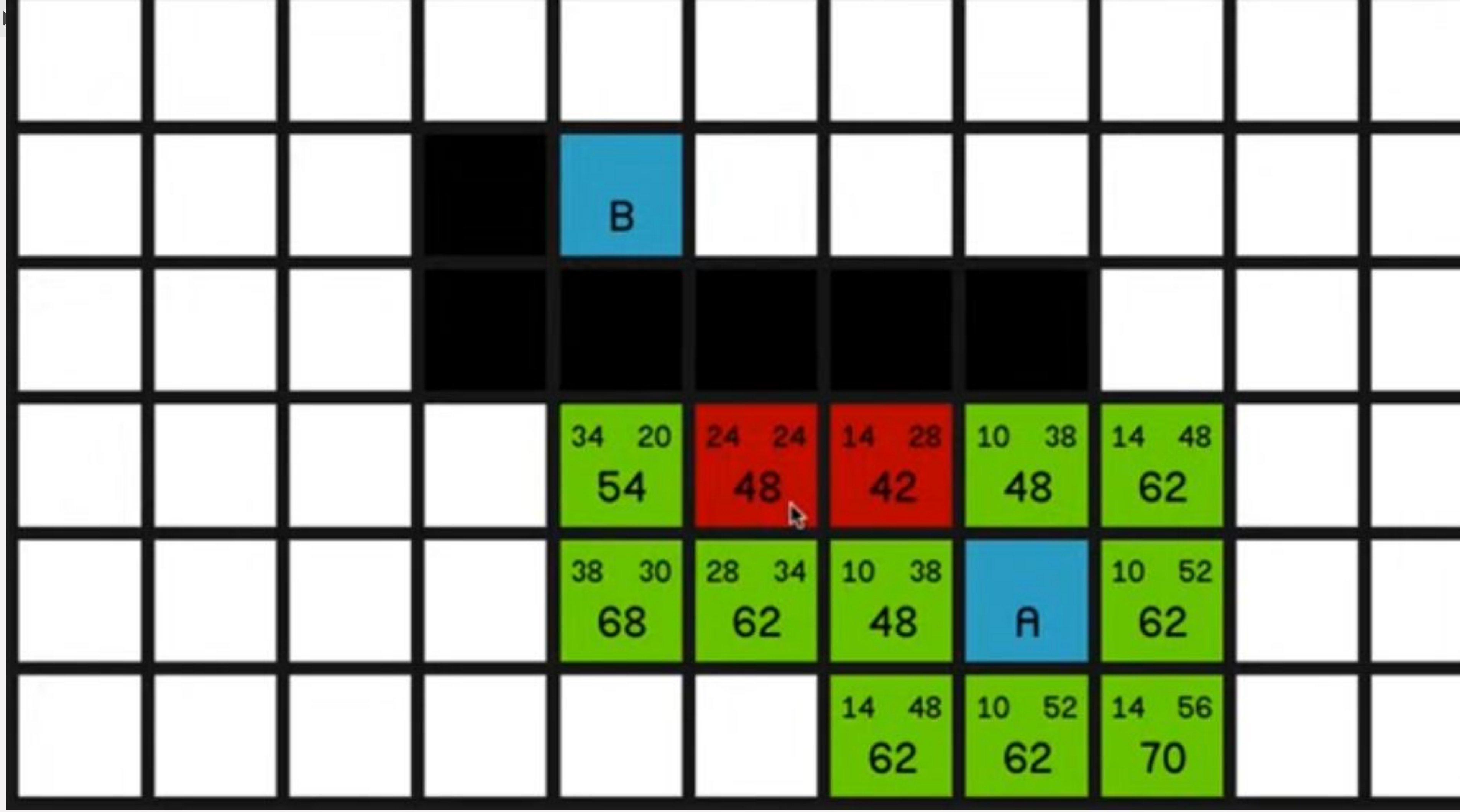
heuristic



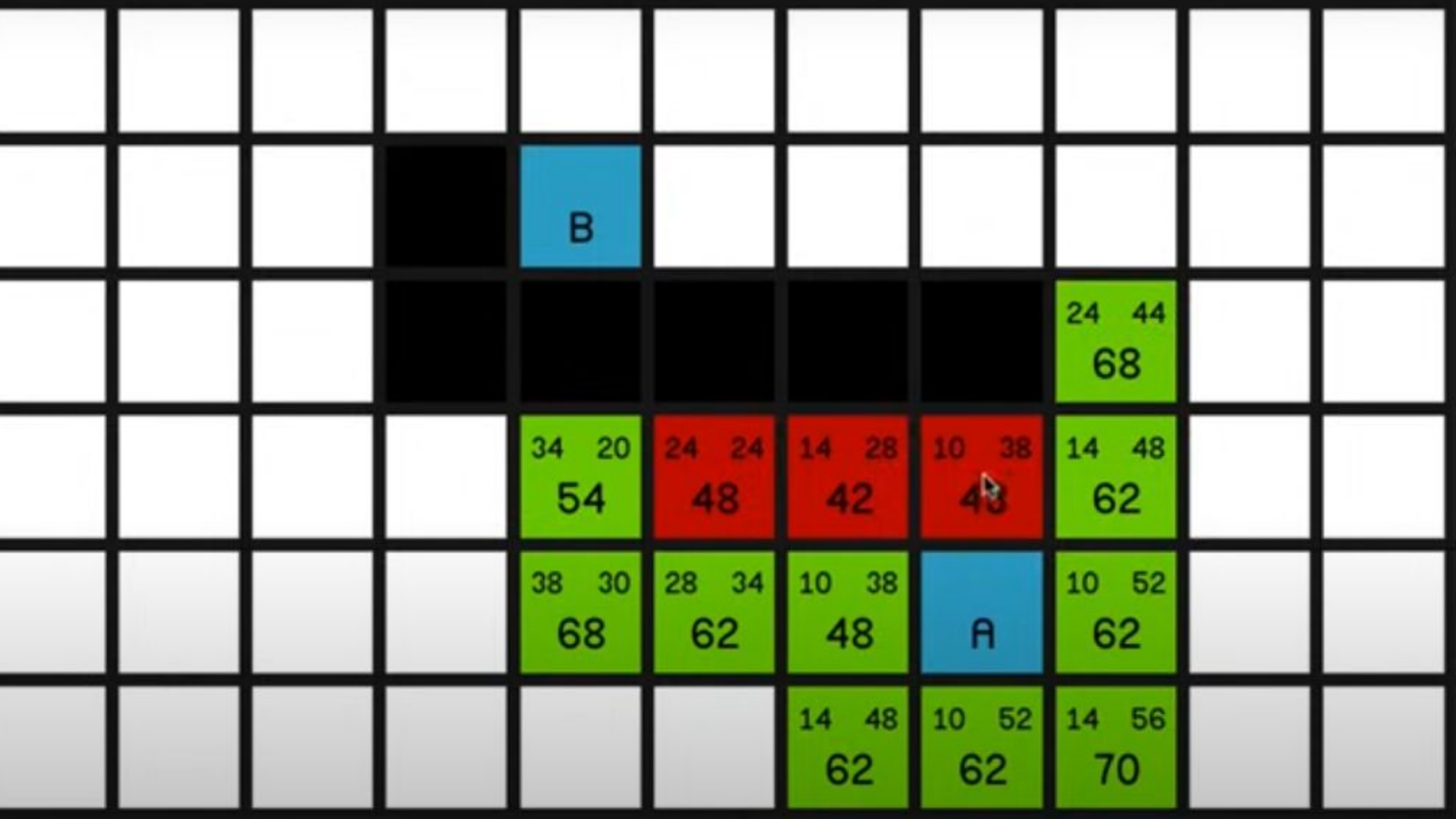
B
A


B
14 28 42
10 38 48
14 48 62
10 38 48
A
10 52 62
14 48 62
10 52 62
14 56 70

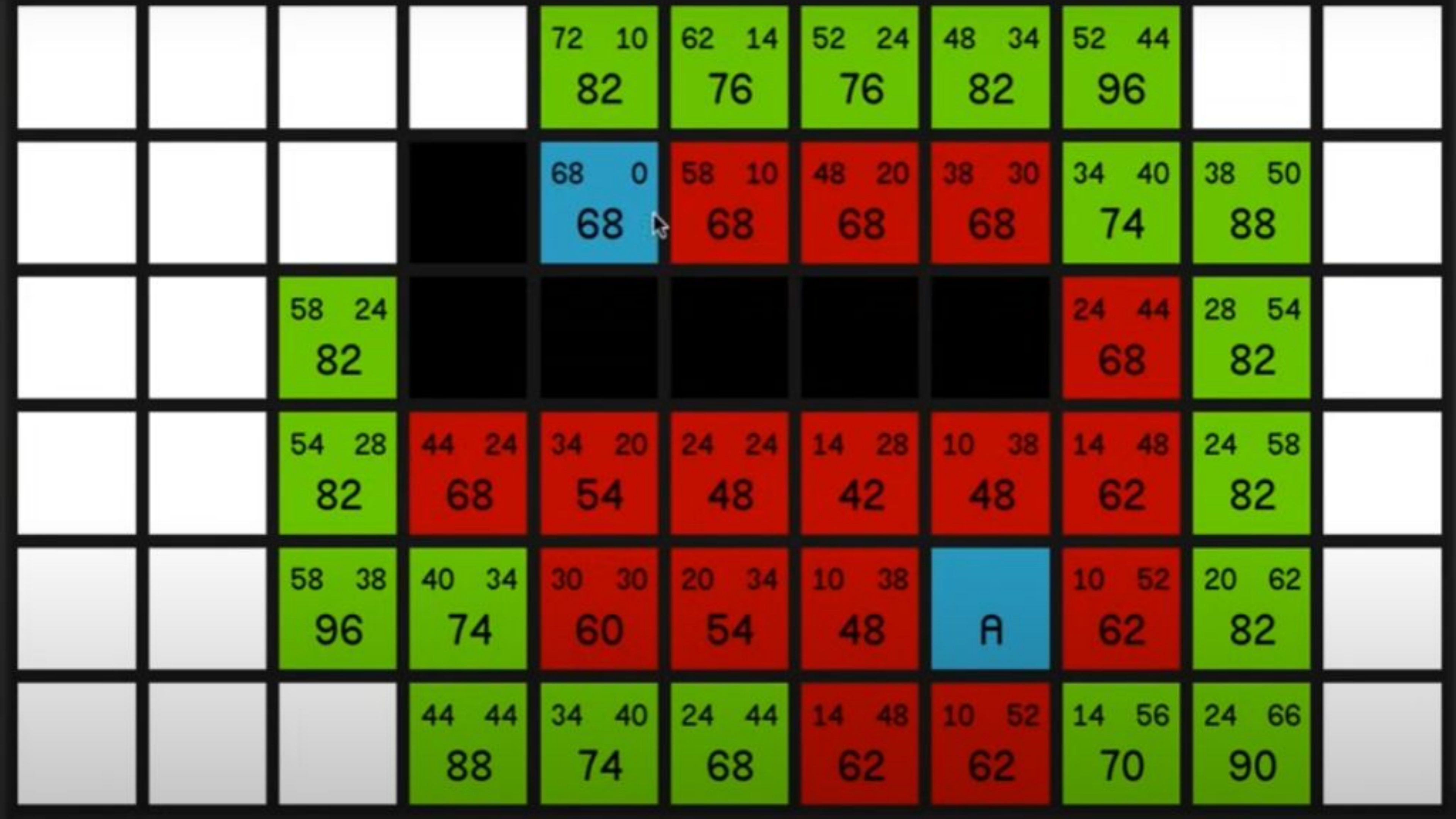

B
24 24
48
14 28
42
10 38
48
14 48
62
28 34
62
10 38
48
A
10 52
62
14 48
62
10 52
62
14 56
70



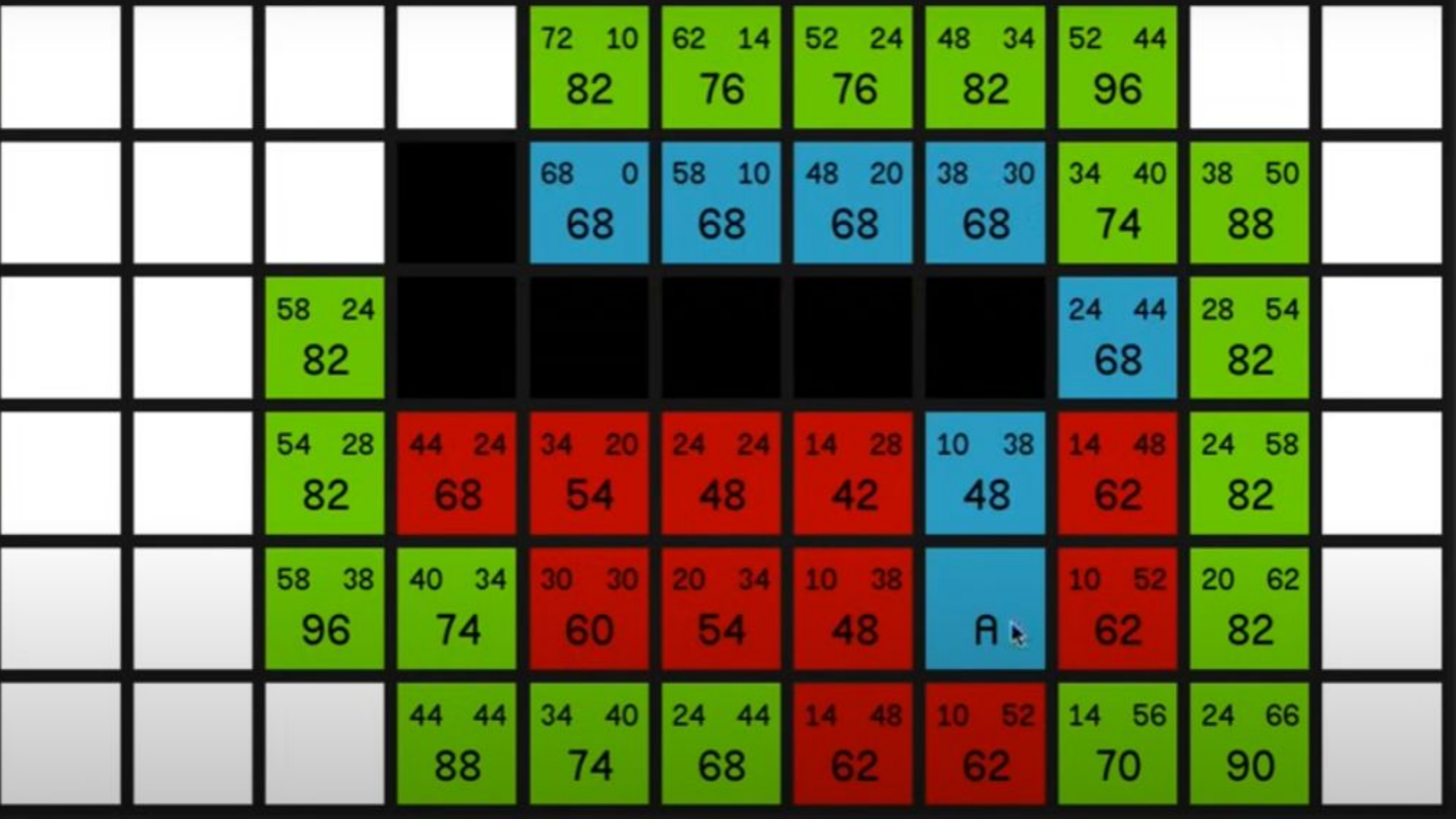
			■	B						
			■	■	■	■	■			
				34 20 54	24 24 48	14 28 42	10 38 48	14 48 62		
				38 30 68	28 34 62	10 38 48	A	10 52 62		
						14 48 62	10 52 62	14 56 70		



			⬛	B						
			⬛	⬛	⬛	⬛	⬛	24 44 68		
				34 20 54	24 24 48	14 28 42	10 38 48	14 48 62		
				38 30 68	28 34 62	10 38 48	A	10 52 62		
						14 48 62	10 52 62	14 56 70		



				72 10 82	62 14 76	52 24 76	48 34 82	52 44 96		
			■	68 0 68	58 10 68	48 20 68	38 30 68	34 40 74	38 50 88	
		58 24 82	■	■	■	■	■	24 44 68	28 54 82	
		54 28 82	44 24 68	34 20 54	24 24 48	14 28 42	10 38 48	14 48 62	24 58 82	
		58 38 96	40 34 74	30 30 60	20 34 54	10 38 48	A	10 52 62	20 62 82	
			44 44 88	34 40 74	24 44 68	14 48 62	10 52 62	14 56 70	24 66 90	

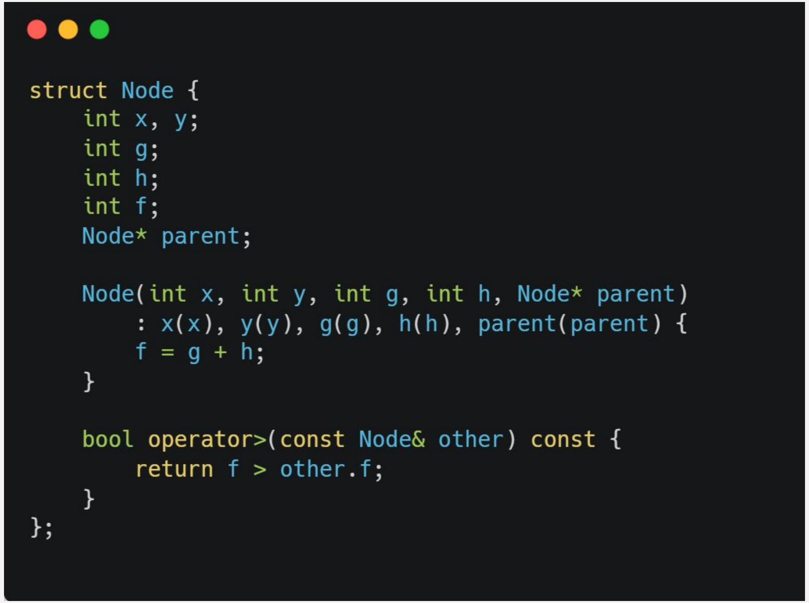


				72 10 82	62 14 76	52 24 76	48 34 82	52 44 96		
			■	68 0 68	58 10 68	48 20 68	38 30 68	34 40 74	38 50 88	
		58 24 82	■	■	■	■	■	24 44 68	28 54 82	
		54 28 82	44 24 68	34 20 54	24 24 48	14 28 42	10 38 48	14 48 62	24 58 82	
		58 38 96	40 34 74	30 30 60	20 34 54	10 38 48	A	10 52 62	20 62 82	
			44 44 88	34 40 74	24 44 68	14 48 62	10 52 62	14 56 70	24 66 90	

Code Insights

Overview of the Implementation

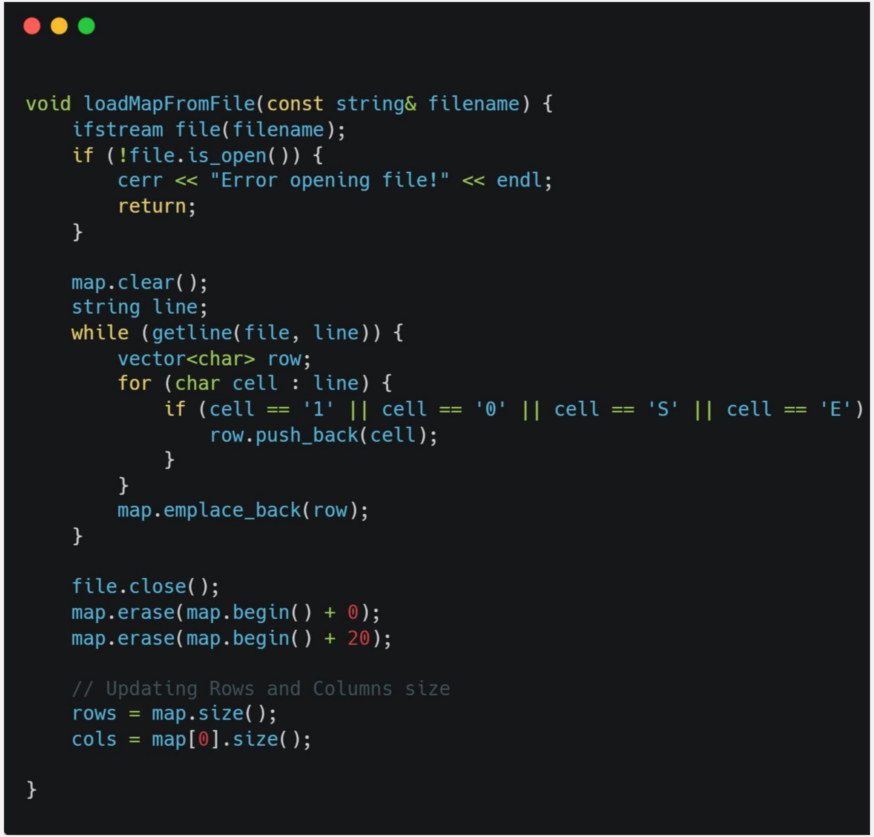
- Reads a grid map from a file.
- Utilises A* algorithm to find the shortest path from start (S) to goal (E).
- Visualises the path on the grid.



```
struct Node {
    int x, y;
    int g;
    int h;
    int f;
    Node* parent;

    Node(int x, int y, int g, int h, Node* parent)
        : x(x), y(y), g(g), h(h), parent(parent) {
        f = g + h;
    }

    bool operator>(const Node& other) const {
        return f > other.f;
    }
};
```

Loading the Map

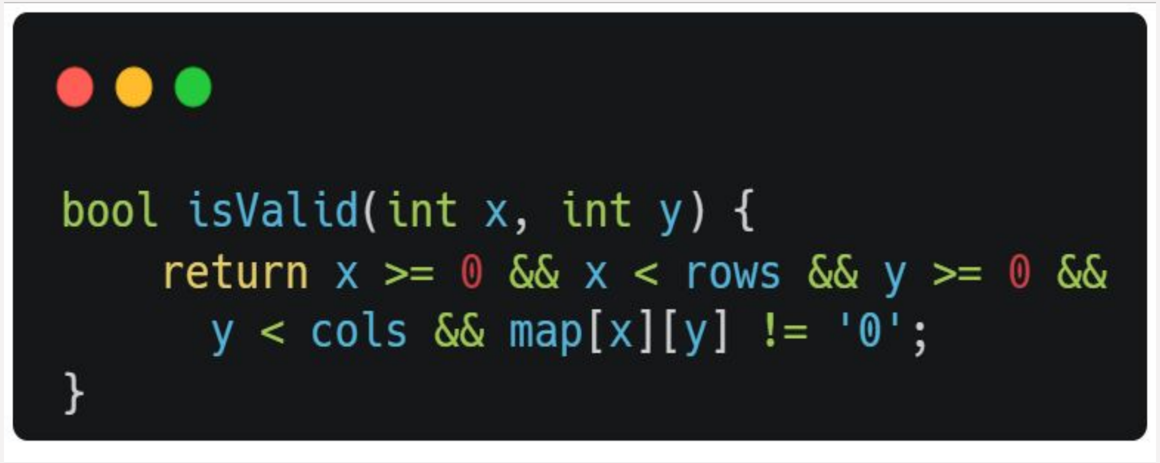
- Reads a map from a file.
- Stores the map in a 2D vector of characters.
- Identifies the size of the grid (rows and columns).



```
void loadMapFromFile(const string& filename) {
    ifstream file(filename);
    if (!file.is_open()) {
        cerr << "Error opening file!" << endl;
        return;
    }

    map.clear();
    string line;
    while (getline(file, line)) {
        vector<char> row;
        for (char cell : line) {
            if (cell == '1' || cell == '0' || cell == 'S' || cell == 'E')
                row.push_back(cell);
            }
        }
        map.emplace_back(row);
    }

    file.close();
    map.erase(map.begin() + 0);
    map.erase(map.begin() + 20);

    // Updating Rows and Columns size
    rows = map.size();
    cols = map[0].size();

}
```

Checking Valid Moves

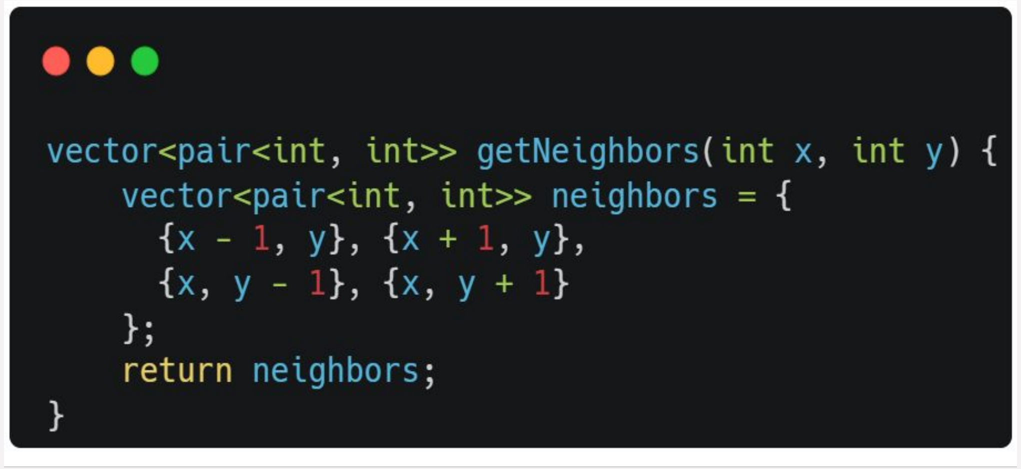
- Ensures the node is within bounds.
- Verifies the node is not an obstacle (0).
- Returns true if the move is valid, false otherwise.



```
bool isValid(int x, int y) {
    return x >= 0 && x < rows && y >= 0 &&
      y < cols && map[x][y] != '0';
}
```

Generating Neighbors

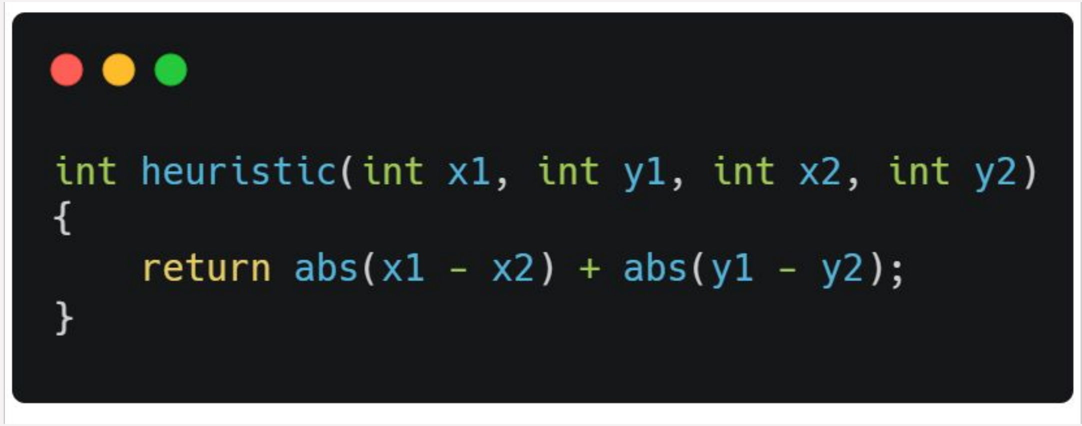
- Returns the 4 possible neighbors for a given node (up, down, left, right).
- Used to explore nodes during pathfinding.



```
vector<pair<int, int>> getNeighbors(int x, int y) {
    vector<pair<int, int>> neighbors = {
      {x - 1, y}, {x + 1, y},
      {x, y - 1}, {x, y + 1}
    };
    return neighbors;
}
```

Calculating Heuristic

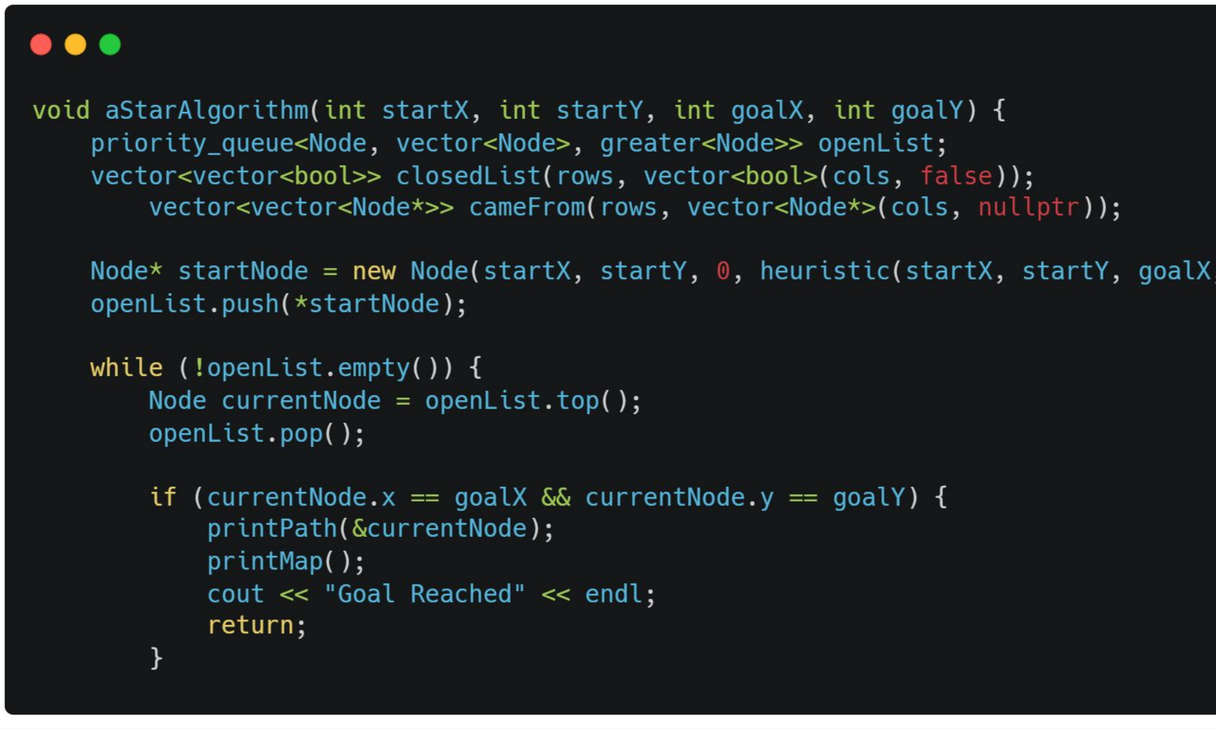
- Uses the Manhattan distance formula: abs(x1 - x2) + abs(y1 - y2).
- Estimates the cost to the goal.
- Ensures the algorithm remains efficient.



```
int heuristic(int x1, int y1, int x2, int y2)
{
    return abs(x1 - x2) + abs(y1 - y2);
}
```

Pathfinding Logic

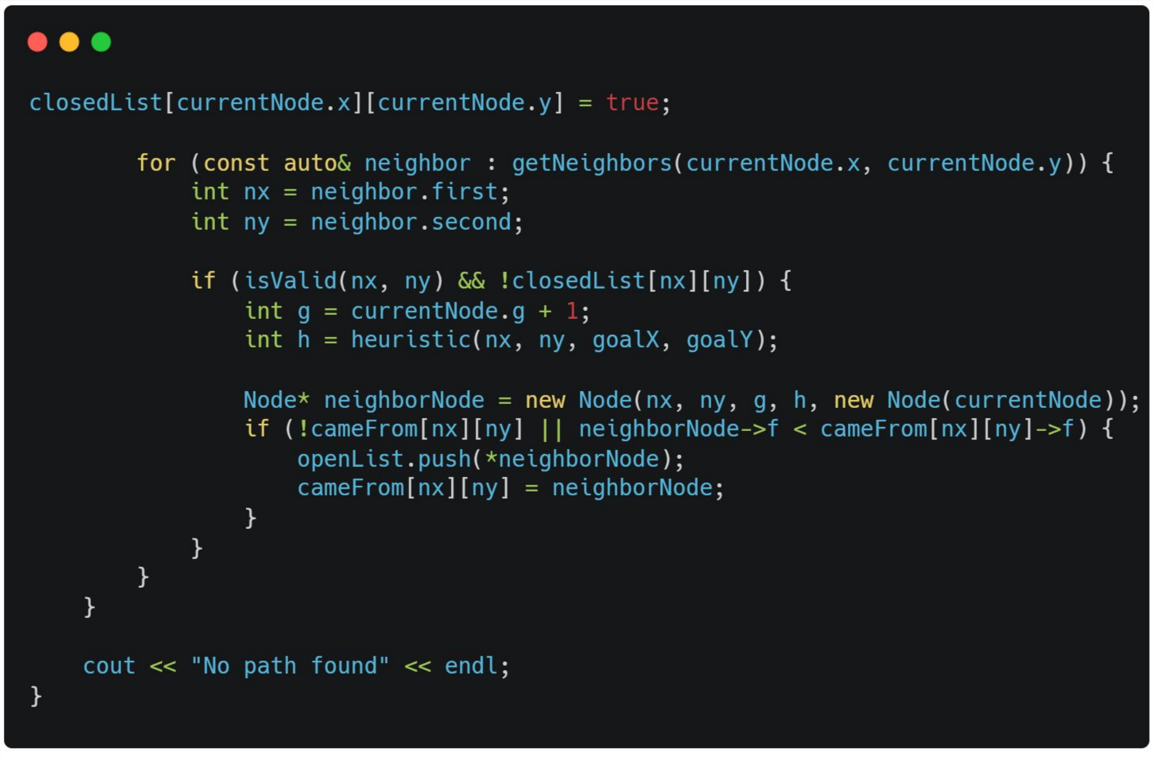
- Initializes the priority queue (openList).
- Processes nodes with the lowest f value first.
- Updates the path until the goal is reached or no path is found.



```
void aStarAlgorithm(int startX, int startY, int goalX, int goalY) {
    priority_queue<Node, vector<Node>, greater<Node>> openList;
    vector<vector<bool>> closedList(rows, vector<bool>(cols, false));
        vector<vector<Node*>> cameFrom(rows, vector<Node*>(cols, nullptr));

    Node* startNode = new Node(startX, startY, 0, heuristic(startX, startY, goalX
    openList.push(*startNode);

    while (!openList.empty()) {
        Node currentNode = openList.top();
        openList.pop();

        if (currentNode.x == goalX && currentNode.y == goalY) {
            printPath(&currentNode);
            printMap();
            cout << "Goal Reached" << endl;
            return;
        }
```

Pathfinding Logic

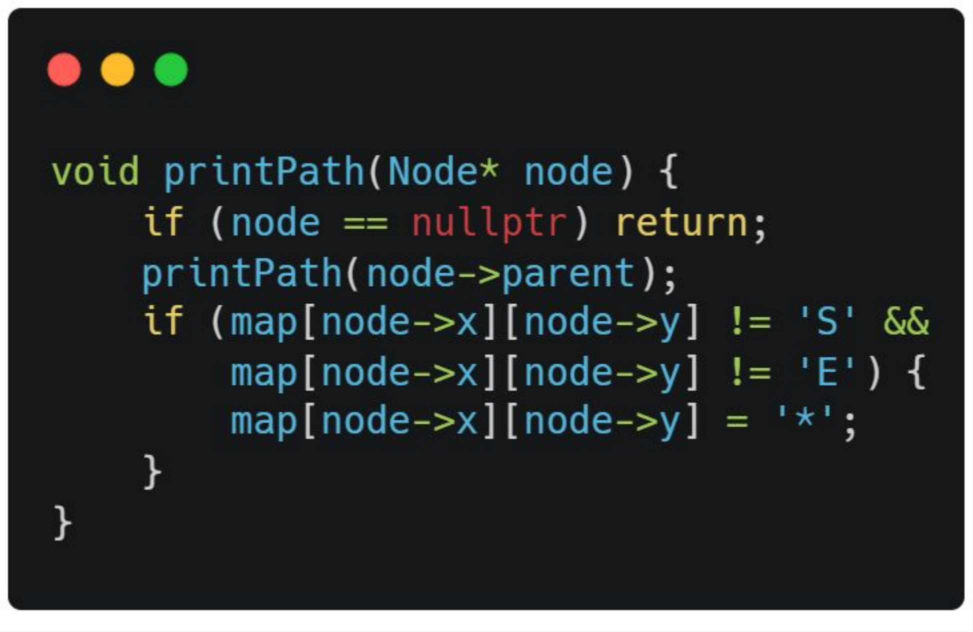
- Initializes the priority queue (openList).
- Processes nodes with the lowest f value first.
- Updates the path until the goal is reached or no path is found.



```
closedList[currentNode.x][currentNode.y] = true;

        for (const auto& neighbor : getNeighbors(currentNode.x, currentNode.y)) {
            int nx = neighbor.first;
            int ny = neighbor.second;

            if (isValid(nx, ny) && !closedList[nx][ny]) {
                int g = currentNode.g + 1;
                int h = heuristic(nx, ny, goalX, goalY);

                Node* neighborNode = new Node(nx, ny, g, h, new Node(currentNode));
                if (!cameFrom[nx][ny] || neighborNode->f < cameFrom[nx][ny]->f) {
                    openList.push(*neighborNode);
                    cameFrom[nx][ny] = neighborNode;
                }
            }
        }
    }

    cout << "No path found" << endl;
}
```

Marking the Path

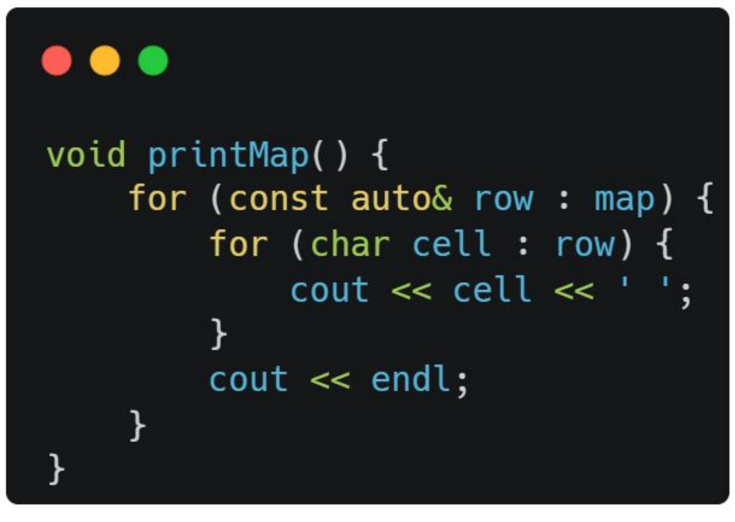
- Recursively traverses the parent nodes to reconstruct the path.
- Marks the path on the map using (*).



```cpp
void printPath(Node* node) {
    if (node == nullptr) return;
    printPath(node->parent);
    if (map[node->x][node->y] != 'S' &&
        map[node->x][node->y] != 'E') {
        map[node->x][node->y] = '*';
    }
}
```

Displaying the Result

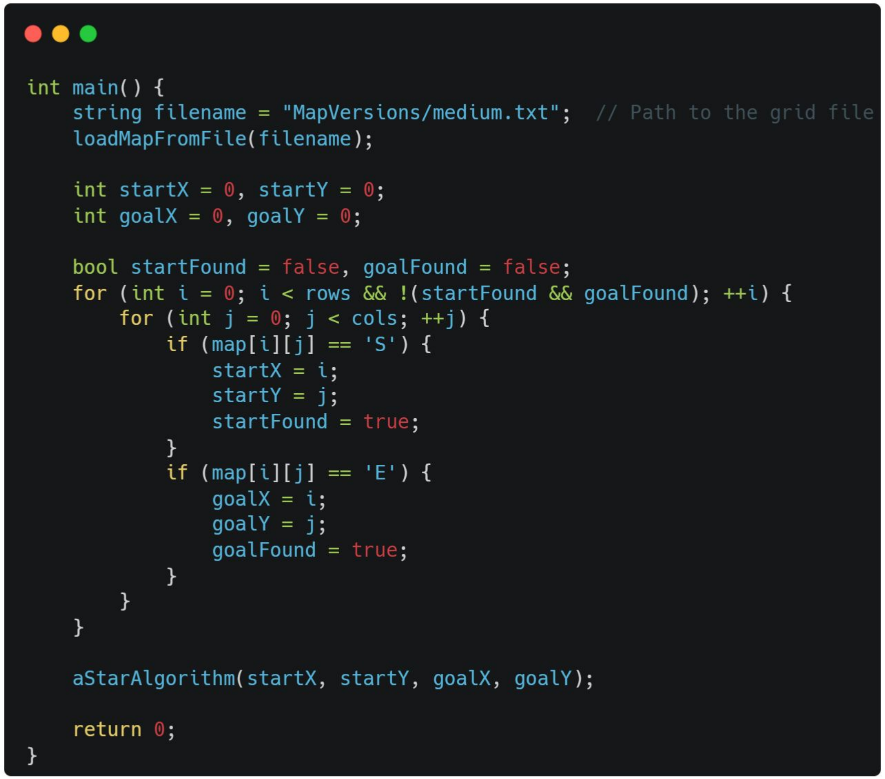
- Recursively traverses the parent nodes to reconstruct the path.
- Marks the path on the map using (*).



```
void printMap() {
    for (const auto& row : map) {
        for (char cell : row) {
            cout << cell << ' ';
        }
        cout << endl;
    }
}
```

Bringing It All Together

- Loads the map file.
- Finds the start (S) and goal (E) points.
- Runs the A* algorithm and prints the result.



```
int main() {
    string filename = "MapVersions/medium.txt";  // Path to the grid file
    loadMapFromFile(filename);

    int startX = 0, startY = 0;
    int goalX = 0, goalY = 0;

    bool startFound = false, goalFound = false;
    for (int i = 0; i < rows && !(startFound && goalFound); ++i) {
        for (int j = 0; j < cols; ++j) {
            if (map[i][j] == 'S') {
                startX = i;
                startY = j;
                startFound = true;
            }
            if (map[i][j] == 'E') {
                goalX = i;
                goalY = j;
                goalFound = true;
            }
        }
    }

    aStarAlgorithm(startX, startY, goalX, goalY);

    return 0;
}
```

Live Demonstration