HW1

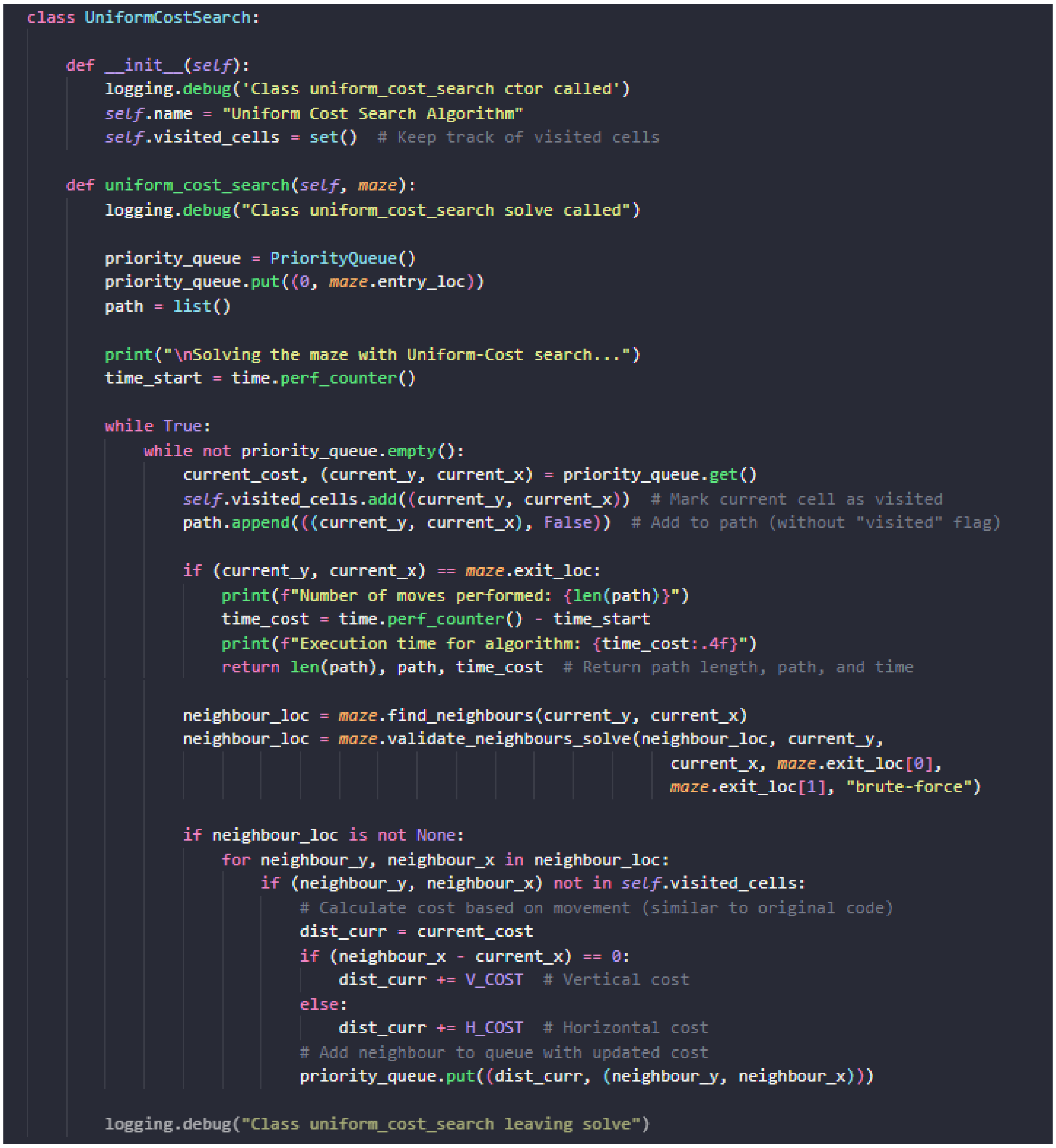
Name: Truong Khanh Nhi

Student ID: 2021315339

1. Explain Uniform-Cost Search Algorithm
2. Explain A Star Search Algorithm
3. Visualization

1. Explain Uniform-Cost Search Algorithm

Uniform-Cost Search (UCS) to find the lowest-cost path between the nodes representing the start and the goal states. UCS is very similar to Breadth-First Search. When all the edges have equal costs, Breadth-First Search finds the optimal solution.



```python
class UniformCostSearch:

    def __init__(self):
        logging.debug('Class uniform_cost_search ctor called')
        self.name = "Uniform Cost Search Algorithm"
        self.visited_cells = set()  # Keep track of visited cells

    def uniform_cost_search(self, maze):
        logging.debug("Class uniform_cost_search solve called")

        priority_queue = PriorityQueue()
        priority_queue.put((0, maze.entry_loc))
        path = list()

        print("\nSolving the maze with Uniform-Cost search...")
        time_start = time.perf_counter()

        while True:
            while not priority_queue.empty():
                current_cost, (current_y, current_x) = priority_queue.get()
                self.visited_cells.add((current_y, current_x))  # Mark current cell as visited
                path.append(((current_y, current_x), False))  # Add to path (without "visited" flag)

                if (current_y, current_x) == maze.exit_loc:
                    print(f"Number of moves performed: {len(path)}")
                    time_cost = time.perf_counter() - time_start
                    print(f"Execution time for algorithm: {time_cost:.4f}")
                    return len(path), path, time_cost  # Return path length, path, and time

                neighbour_loc = maze.find_neighbours(current_y, current_x)
                neighbour_loc = maze.validate_neighbours_solve(neighbour_loc, current_y,
                                                               current_x, maze.exit_loc[0],
                                                               maze.exit_loc[1], "brute-force")

                if neighbour_loc is not None:
                    for neighbour_y, neighbour_x in neighbour_loc:
                        if (neighbour_y, neighbour_x) not in self.visited_cells:
                            # Calculate cost based on movement (similar to original code)
                            dist_curr = current_cost
                            if (neighbour_x - current_x) == 0:
                                dist_curr += V_COST  # Vertical cost
                            else:
                                dist_curr += H_COST  # Horizontal cost
                            # Add neighbour to queue with updated cost
                            priority_queue.put((dist_curr, (neighbour_y, neighbour_x)))

        logging.debug("Class uniform_cost_search leaving solve")
```

Advantage: Complete and Optimal

Disadvantage: Slow

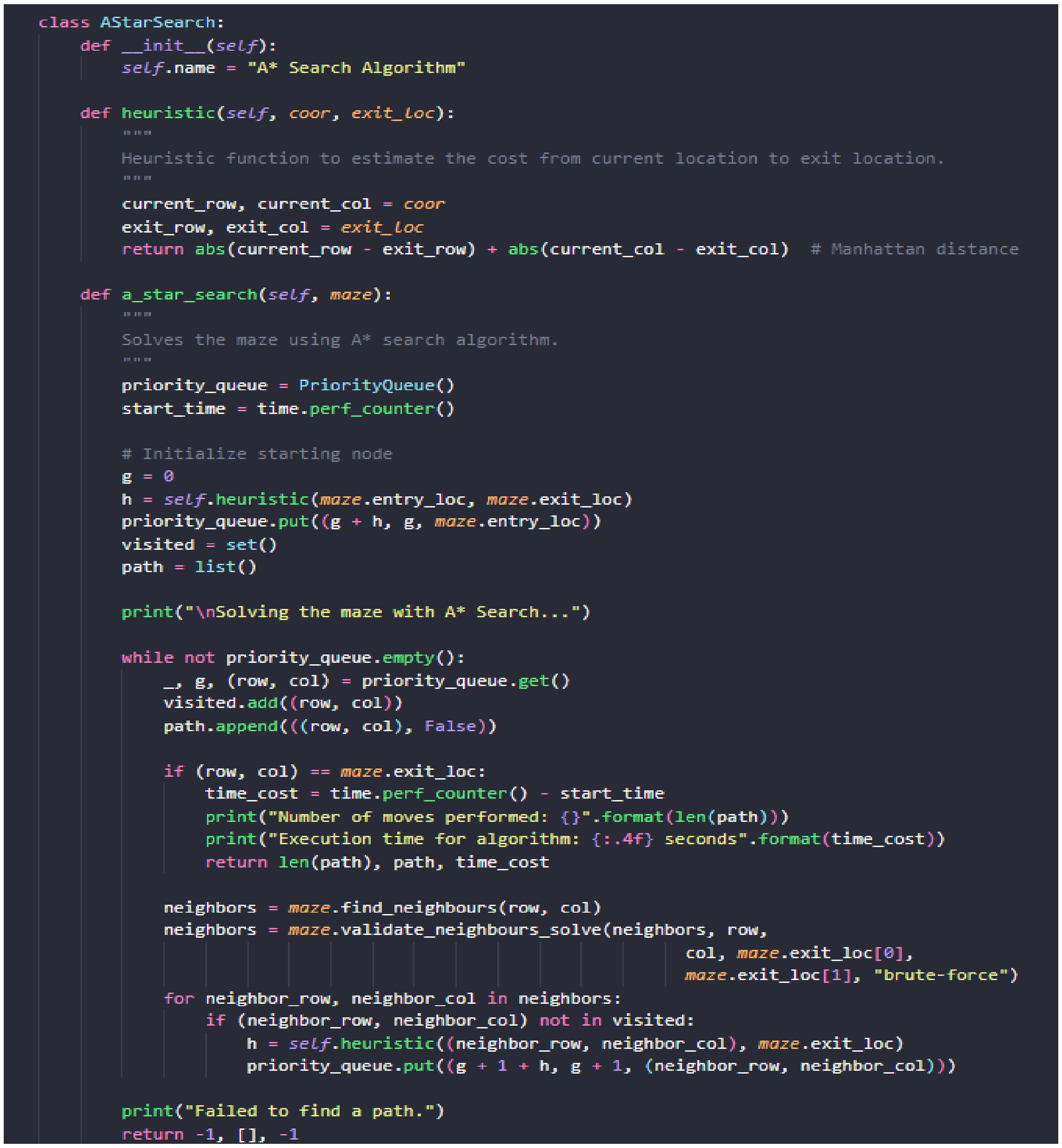
Time complexity: $O(b^1 + [C^*/\varepsilon])$in the worst case

Space complexity: $O(b^1 + [C^*/\varepsilon])$

Total: Uniform-Cost Search in this situation provides a complete and optimal path to solve the maze.

2. Explain A Star Search Algorithm

It is a search algorithm that is used to find the shortest path between an initial and a final point. It is a handy algorithm that is often used for map traversal to find the shortest path to be taken. A* was initially designed as a graph traversal problem, to help build a robot that can find its own course. It remains a widely popular algorithm for graph traversal. It searches for shorter paths first, thus making it an optimal and complete algorithm. An optimal algorithm will find the least cost outcome for a problem, while a complete algorithm finds all the possible outcomes of a problem. Another aspect that makes A* so powerful is the use of weighted graphs in its implementation. A weighted graph uses numbers to represent the cost of taking each path or course of action.



```python
class AStarSearch:
    def __init__(self):
        self.name = "A* Search Algorithm"

    def heuristic(self, coor, exit_loc):
        """
        Heuristic function to estimate the cost from current location to exit location.
        """
        current_row, current_col = coor
        exit_row, exit_col = exit_loc
        return abs(current_row - exit_row) + abs(current_col - exit_col)  # Manhattan distance

    def a_star_search(self, maze):
        """
        Solves the maze using A* search algorithm.
        """
        priority_queue = PriorityQueue()
        start_time = time.perf_counter()

        # Initialize starting node
        g = 0
        h = self.heuristic(maze.entry_loc, maze.exit_loc)
        priority_queue.put((g + h, g, maze.entry_loc))
        visited = set()
        path = list()

        print("\nSolving the maze with A* Search...")

        while not priority_queue.empty():
            _, g, (row, col) = priority_queue.get()
            visited.add((row, col))
            path.append(((row, col), False))

            if (row, col) == maze.exit_loc:
                time_cost = time.perf_counter() - start_time
                print("Number of moves performed: {}".format(len(path)))
                print("Execution time for algorithm: {:.4f} seconds".format(time_cost))
                return len(path), path, time_cost

            neighbors = maze.find_neighbours(row, col)
            neighbors = maze.validate_neighbours_solve(neighbors, row,
                                                       col, maze.exit_loc[0],
                                                       maze.exit_loc[1], "brute-force")
            for neighbor_row, neighbor_col in neighbors:
                if (neighbor_row, neighbor_col) not in visited:
                    h = self.heuristic((neighbor_row, neighbor_col), maze.exit_loc)
                    priority_queue.put((g + 1 + h, g + 1, (neighbor_row, neighbor_col)))

        print("Failed to find a path.")
        return -1, [], -1
```

Advantage: Complete

Disadvantage: Optimal, more space

Time complexity: *O(b^d) worst case*

Space Complexity: *O(b^d)*

Total: A Star Algorithm in this situation provide a complete but not optimal path to solve the maze.

Manhattan Distance

The Manhattan Distance is the total of the absolute values of the discrepancies between the x and y coordinates of the current and the goal cells.

The formula is summarized below -

h = abs (curr_cell.x – goal.x) +

    abs (curr_cell.y – goal.y)

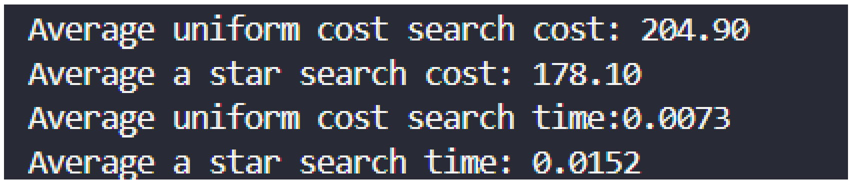
Use this heuristic method when there are only permitted to move in four directions - top, left, right, and bottom.

The algorithm calculates the cost to all its immediate neighboring nodes, n, and chooses the one incurring the least cost. This process repeats until no new nodes can be chosen and all paths have been traversed. Then, you should consider the best path among them. If f(n) represents the final cost, then it can be denoted as :

f(n) = g(n) + h(n), where :

g(n) = cost of traversing from one node to another. This will vary from node to node h(n) = heuristic approximation of the node's value. This is not a real value but an approximation cost.

Result of the maze

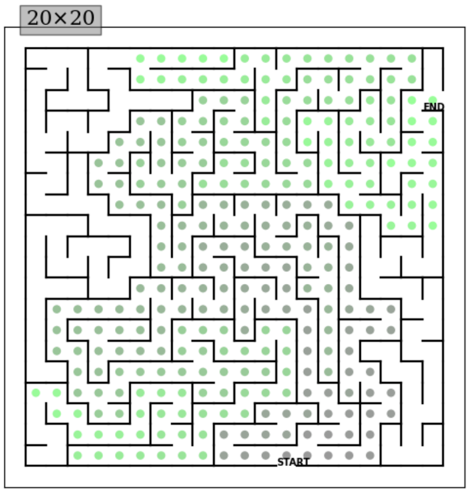


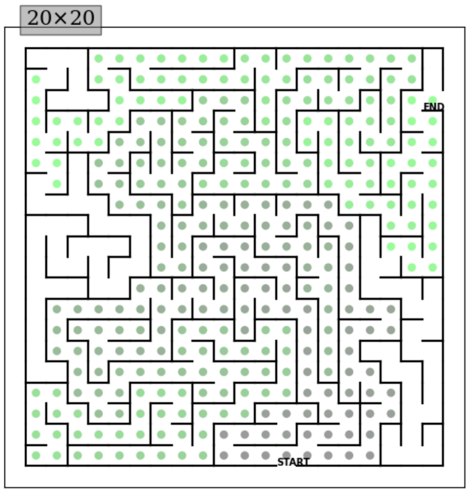
```
Average uniform cost search cost: 204.90
Average a star search cost: 178.10
Average uniform cost search time:0.0073
Average a star search time: 0.0152
```

Look at the average time between Uniform Cost and A Star, clearly, the cost of A Star is way less compared to Uniform Cost, but the average time A Star took to complete the maze is longer than Uniform Cost.

3. Visualization

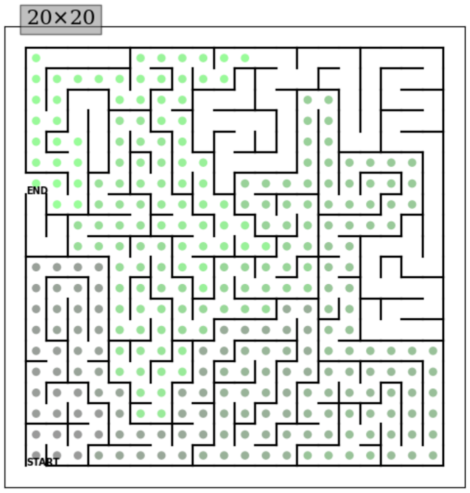
A Star Search **Uniform Cost Search**

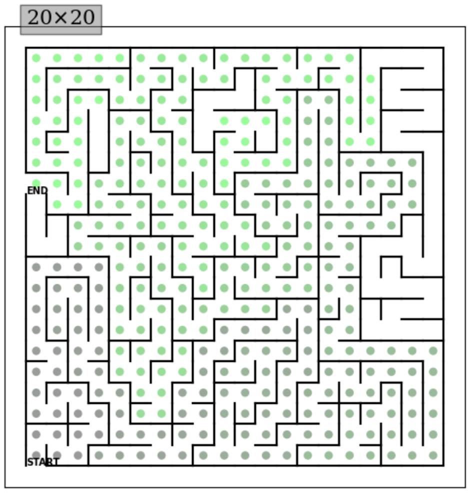


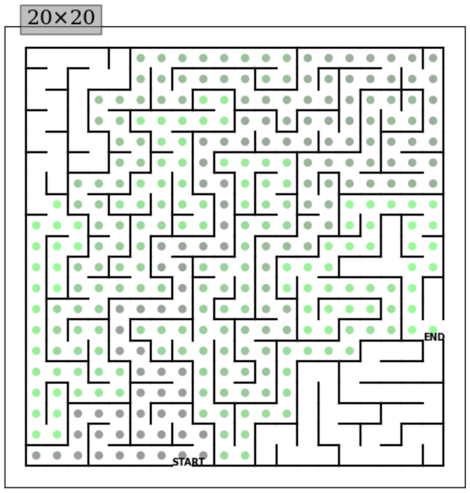


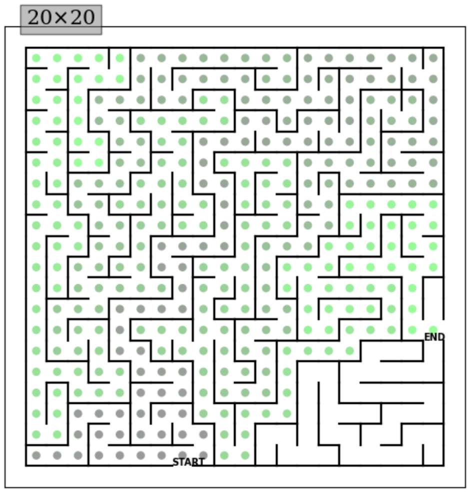
A Star Search

Uniform Cost Search



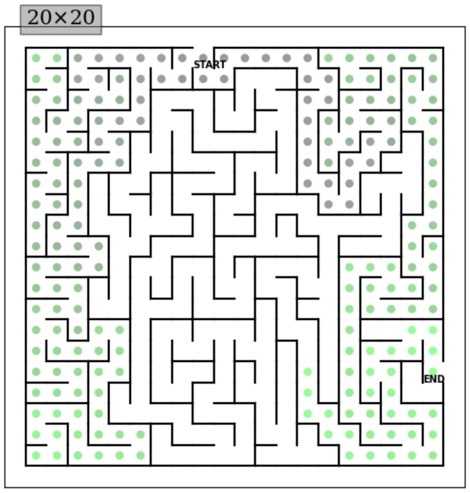


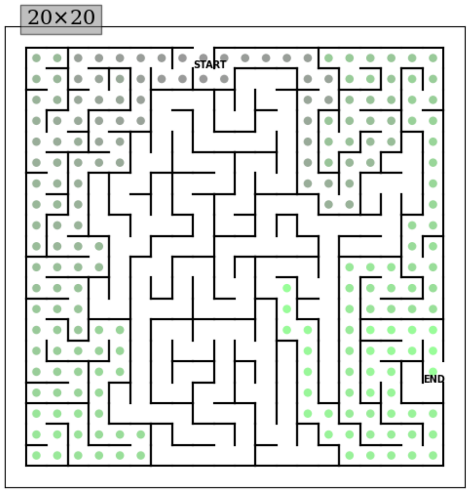


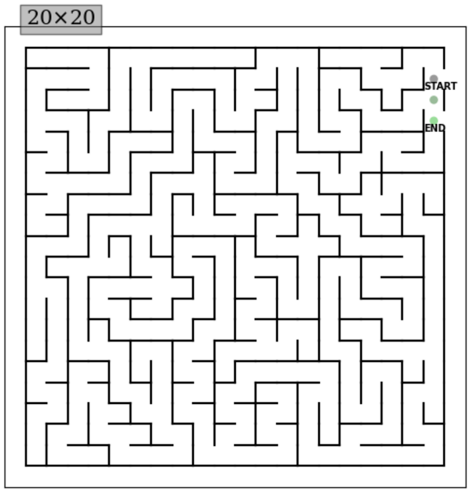


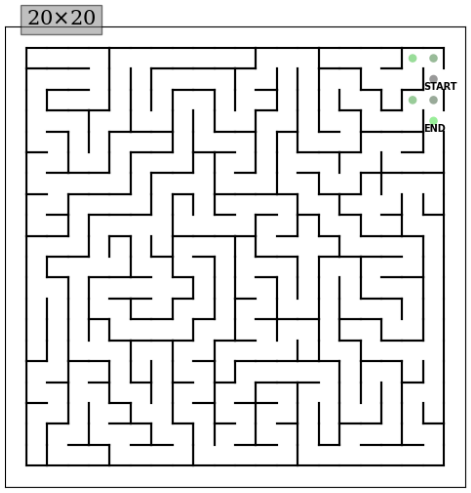
A Star Search

Uniform Cost Search

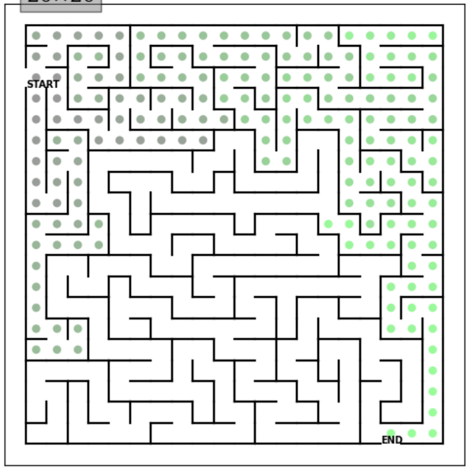




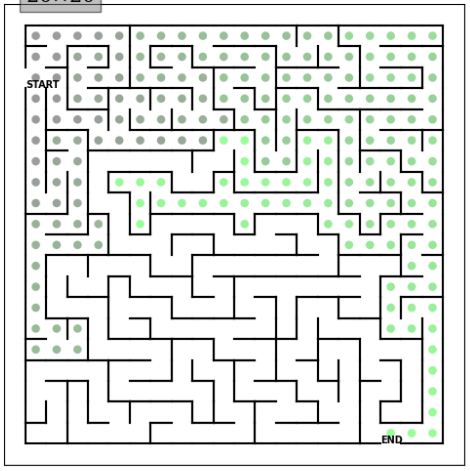




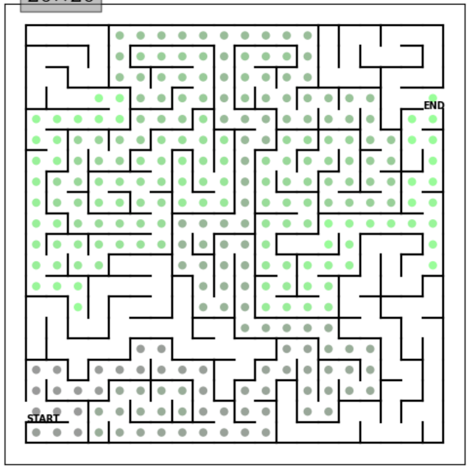
20×20



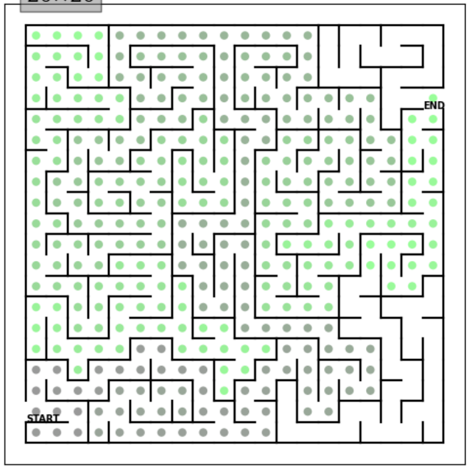
20×20



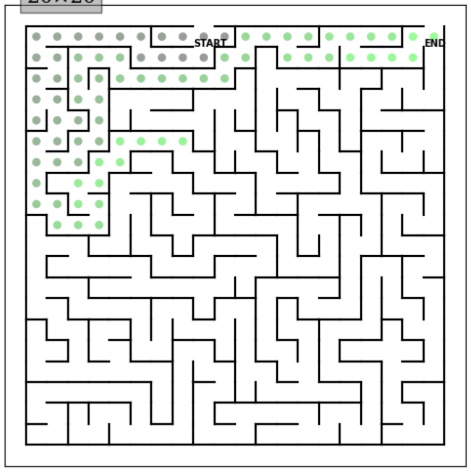
20×20



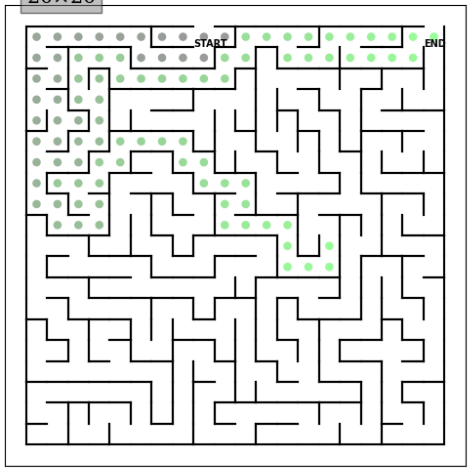
20×20



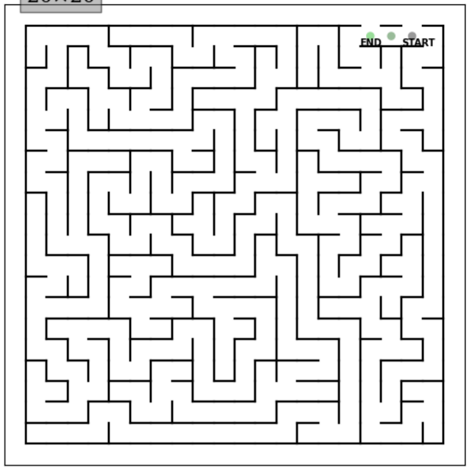
20×20



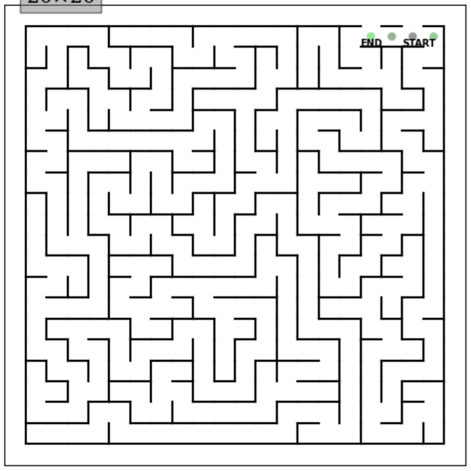
20×20



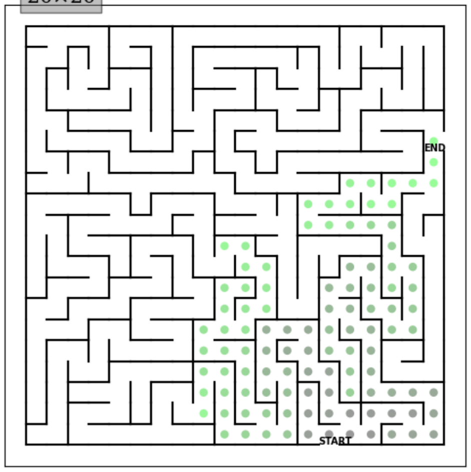
20×20



20×20



20×20



20×20