



T.C.

MARMARA UNIVERSITY

FACULTY OF ENGINEERING

COMPUTER ENGINEERING DEPARTMENT

CSE 4082 – Assignment 2

Report

Group Members

Ömercan Göktaş – 150119671

1. Cell Class

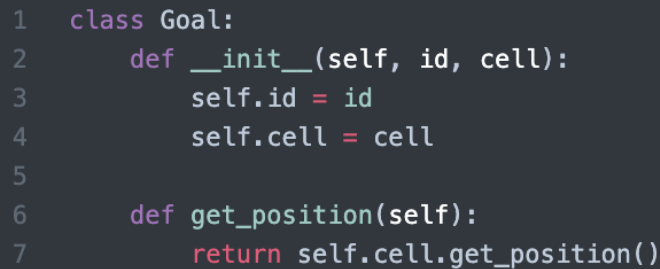
The Cell class has properties such as row, col, east, south, west, and north. Row and col identify the position of the cell in the maze. East, south, west, and north properties identify the neighboring cells of the current cell. If the value of any of these properties is None, it means that there is no neighboring cell in that direction. As an instance, the cell (3,2) which is the starting cell has no neighboring cell in the north and east. So, the value of north and east property of the cell is None as it has neighboring cells in the south and west directions. That's why, the values of south and west are going to contain the tree objects of the neighboring cells in the south and west directions respectively, and so on.

- **get_position():** is a method of Cell class that returns the row and column of the current cell.

```
1  class Cell:
2      def __init__(self, row, col, east, south, west, north):
3          self.row = row
4          self.col = col
5          self.east = east
6          self.south = south
7          self.west = west
8          self.north = north
9
10     # get the row and column of the current cell
11     def get_position(self):
12         return self.row, self.col
```

2. Goal Class

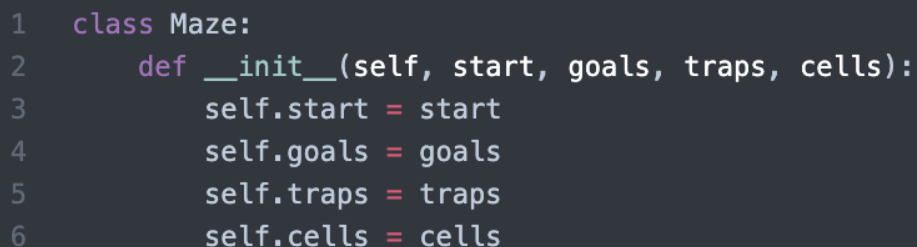
The Goal class contains only two properties which are id, and cell. 'id' determines the number of goal cell, and 'cell' is the object from the Cell class.



```
1 class Goal:
2     def __init__(self, id, cell):
3         self.id = id
4         self.cell = cell
5
6     def get_position(self):
7         return self.cell.get_position()
```

3. Maze Class

The Maze class is used to create a maze from a json file (maze.json) and get the neighbors of a cell. It has properties such as start, goals, traps, and cells. The start property is the starting cell of the maze. The goals property is a list of goals in the maze. The traps property is a list of traps in the maze, and the cells property is a list of cells in the maze.



```
1 class Maze:
2     def __init__(self, start, goals, traps, cells):
3         self.start = start
4         self.goals = goals
5         self.traps = traps
6         self.cells = cells
```

- **create_from_json()**: is a class method that creates a maze from a json file (maze.json).
- **get_neighbors()**: is a method of Maze class that returns the neighbors of the current cell (north, east, south, west) as a dictionary. For example, if the current cell is (3, 2), the return value of get_neighbors() is {'North': None, 'East': None, 'South': (4, 2), 'West': (3, 1)}.

4. TreeNode Class

The TreeNode class is used to build the tree. The tree is built using the start cell. The class has properties for each direction (east, south, west, north) and each property is a TreeNode object. The class has a parent property which is also a TreeNode object. The parent property is used to find the path to the goal cell. The class has a cell property which is a Cell object. The cell property is used to find the position of the current cell.

```
1
2 # TreeNode class for building the tree
3 class TreeNode:
4     def __init__(self, cell, parent=None):
5         self.cell = cell
6         self.north = None
7         self.south = None
8         self.west = None
9         self.east = None
10        self.parent = parent
```

- **build_tree():** the idea behind the build_tree method is to check the available moves for the current cell and create a TreeNode object for each available move.

```

1  # build the tree using the maze recursively
2  def _build_tree_recursive(self, maze, current_node, visited_cells):
3      if current_node.cell in visited_cells:
4          return
5      # add the current cell to the visited cells
6      visited_cells.add(current_node.cell)
7      # get the available moves for the current cell
8      available_moves = maze.get_neighbors(current_node.cell)
9
10     for available_move in available_moves:
11         # because of the directions, we need to clear the visited cells.
12         # so that start cell will start checking other directions with empty visited cells
13         if current_node.cell == maze.start:
14             visited_cells.clear()
15             visited_cells.add(current_node.cell)
16         # if the movement is valid (there is no wall for the movement)
17         if available_moves[available_move]:
18             child_cell = TreeNode(cell=available_moves[available_move], parent=current_node)
19             setattr(current_node, available_move.lower(), child_cell)
20             self._build_tree_recursive(maze=maze, current_node=child_cell, visited_cells=visited_cells)

```

- **is_cell_goal():** is used to check if the current cell is a goal cell.
- **is_cell_trap():** is used to check if the current cell is a trap cell.
- **find_solution_path_and_cost():** is used to find the path to the goal cell and the cost of the path.

```

1  # find the path to the goal cell
2  def _find_solution_path_and_cost(self, traps):
3      solution_cost = 0
4      solution_path = []
5      if self:
6          current_node = self
7          while current_node:
8              solution_path.append(current_node.cell.get_position())
9              solution_cost += 1
10             if current_node._is_cell_trap(traps):
11                 solution_cost += 6
12             current_node = current_node.parent
13
14     return solution_cost, self.reverse_list(solution_path)

```

- **search_algorithms():** is used to return the results of all search algorithms. It uses the `depth_first_search()`, `breadth_first_search()`, `iterative_deepening_search()`, `gbfs_astar_ucs()` methods.

```

1  # return the results of all search algorithms
2  def search_algorithms(self, maze):
3      return {
4          "DFS result": self.depth_first_search(maze),
5          "BFS result": self.breadth_first_search(maze),
6          "IDS result": self.iterative_deepening_search(maze, 10),
7          "UCS result": self.gbfs_astar_ucs(maze, "ucs"),
8          "Greedy Best First Search result": self.gbfs_astar_ucs(maze, "greedy"),
9          "A* result": self.gbfs_astar_ucs(maze, "a_star")
10     }

```

4.2. Applied searching algorithms:

- **depth_first_search():** is used to find the path to the goal cell using depth first search algorithm. The algorithm uses postorder traversal.

```

1  # finding the path to the goal cell using depth first search algorithm with postorder traversal
2  def _depth_first_search_postorder(self, maze, expanded_nodes):
3      directions = ["east", "south", "west", "north"]
4      # if the current node is not empty
5      if self:
6          # for each direction
7          for direction in directions:
8              child_node = getattr(self, direction)
9              expanded_nodes.append(self.cell)
10             # if the child node is not empty
11             if child_node:
12                 result = child_node._depth_first_search_postorder(maze, expanded_nodes)
13                 if result:
14                     return result
15             # if the current node is a goal cell
16             if self._is_cell_goal(maze.goals):
17                 path_and_cost = self._find_solution_path_and_cost(maze.traps)
18                 return self.cell, path_and_cost[0], path_and_cost[1], expanded_nodes
19     return None

```

- **breadth_first_search():** is used to find the path to the goal cell using breadth first search algorithm.

```

1  # finding the path to the closest goal cell using breadth first search algorithm
2  def _breadth_first_search(self, goals, traps, expanded_nodes):
3      directions = ["east", "south", "west", "north"]
4      # if the current node is not empty
5      if self:
6          queue = deque([self])
7          while queue:
8              current_node = queue.popleft()
9              if current_node.cell in expanded_nodes:
10                 continue
11             expanded_nodes.append(current_node.cell)
12             if current_node._is_cell_goal(goals):
13                 solution_cost_path = current_node._find_solution_path_and_cost(traps)
14                 return current_node.cell, solution_cost_path[0], solution_cost_path[1], expanded_nodes
15             for direction in directions:
16                 child_node = getattr(current_node, direction)
17                 if child_node:
18                     queue.append(child_node)
19             return False

```

- **iterative_deepening_search():** is used to find the path to the goal cell using iterative deepening search algorithm. It takes max_depth as a parameter. Then it checks the goal cell for each depth from 0 to max_depth.

```

1  # return the result of iterative deepening search algorithm
2  def iterative_deepening_search(self, maze, max_depth):
3      # from 0 to max_depth
4      for depth in range(max_depth):
5          expanded_nodes = list()
6          result = self._iterative_deepening_search(maze, depth, expanded_nodes)
7          if result:
8              return result
9      return None
10
11 # finding the path to the goal cell using iterative deepening search algorithm
12 def _iterative_deepening_search(self, maze, max_depth, expanded_nodes):
13     directions = ["east", "south", "west", "north"]
14     # if the current node is not empty
15     if self:
16         queue = deque([self])
17         queue.append(max_depth)
18         # while the queue is not empty
19         while queue:
20             current_node = queue.popleft()
21             current_depth = queue.popleft()
22             if current_depth == 0:
23                 continue
24             # if the current node is a goal cell
25             if current_node._is_cell_goal(maze.goals):
26                 path_and_cost = current_node._find_solution_path_and_cost(maze.traps)
27                 return current_node.cell, path_and_cost[0], path_and_cost[1], expanded_nodes
28             # for each direction
29             for direction in directions:
30                 child_node = getattr(current_node, direction)
31                 expanded_nodes.append(current_node.cell)
32                 if child_node:
33                     queue.append(child_node)
34                     queue.append(current_depth - 1)

```

- **gbfs_astar_ucs():** is used to find the optimum path to the goal cell using greedy best first search and A* search algorithms and uniform cost search. It takes algorithm as a parameter. Then it checks the goal cell using the algorithm.

```

1  # finding the goal cell using greedy best first search algorithm
2  def _gbfs_astar_ucs(self, maze, expanded_nodes, algorithm):
3      directions = ["east", "south", "west", "north"]
4      # if the current node is not empty
5      if self:
6          queue = [self]
7          # while the queue is not empty
8          while queue:
9              current_node = queue.pop(0)
10             if current_node._is_cell_goal(maze.goals):
11                 path_and_cost = current_node._find_solution_path_and_cost(maze.traps)
12                 return current_node.cell, path_and_cost[0], path_and_cost[1], expanded_nodes
13             # for each direction
14             for direction in directions:
15                 child_node = getattr(current_node, direction)
16                 expanded_nodes.append(current_node.cell)
17                 if child_node:
18                     queue.append(child_node)

```

- A* search algorithm uses manhattan distance heuristic. The idea behind the manhattan distance heuristic is to find the distance between the current cell and the goal cell. Then it returns the minimum distance between the current cell and the goal cell.
- Greedy Best First Search algorithm sorts the queue by cost. The idea behind the greedy best first search algorithm is to find the goal cell using the minimum cost.
- Uniform Cost Search algorithm sorts the queue by path length. The idea behind the uniform cost search algorithm is to find the goal cell using the minimum path length.

- **manhattan_distance():** is used to find the manhattan distance between the current cell and the goal cell. It takes cell1 and goals as parameters. Then it returns the minimum distance between the current cell and the goal cell.

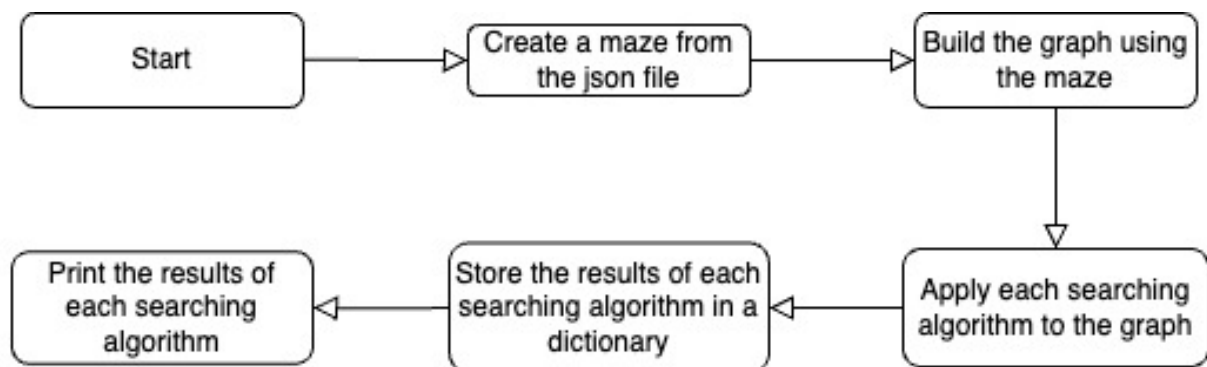
```

1  # return the result of A* search algorithm using manhattan distance heuristic
2  def manhattan_distance(self, cell1, goals):
3      distances = []
4      for goal in goals:
5          distances.append(abs(cell1[0] - goal.cell.row) + abs(cell1[1] - goal.cell.col))
6      return min(distances)

```

5. Flowchart of the Program

A user only needs to run main.py file like 'python3 main.py'. The program will automatically run necessary functions to create a maze, build a graph, apply searching algorithms, and print the results.



6. Result

Result of each algorithm are stored in a dictionary. The dictionary contains the following information:

- **Found Goal:** the position of the found goal node.
- **Solution Cost:** the cost of the solution path.
- **Solution Path:** the path from the start node to the found goal node.
- **Expanded Nodes:** the nodes that have been expanded while searching the goal node.
- **Number of nodes expanded:** the number of nodes that have been expanded.

6.1. Depth First Search

- **Found goal:** (7, 2)
- **Solution cost:** 35
- **Solution path:** (3, 2) (4, 2) (4, 3) (5, 3) (6, 3) (6, 2) (6, 1) (7, 1) (7, 2) (7, 3) (7, 2)
- **Expanded nodes:** (3, 2) (3, 2) (4, 2) (4, 3) (4, 3) (5, 3) (5, 3) (6, 3) (6, 4) (6, 4) (6, 4) (6, 3) (6, 3) (6, 3) (6, 4) (5, 4) (5, 4) (6, 4) (6, 4) (6, 4) (6, 4) (5, 4) (5, 4) (4, 4) (4, 5) (4, 5) (5, 5) (5, 5) (6, 5) (6, 5) (7, 5) (7, 5) (7, 5) (7, 5) (6, 5) (6, 5) (6, 5) (6, 5) (6, 5) (6, 5) (5, 5) (5, 5) (5, 5) (5, 5) (5, 5) (4, 5) (4, 5) (4, 5) (4, 5) (4, 5) (4, 4) (4, 4) (4, 4) (4, 4) (4, 5) (4, 4) (5, 4) (5, 4) (5, 4) (5, 4) (4, 4) (4, 4) (6, 3) (6, 3) (6, 2) (6, 3) (6, 3) (6, 3) (6, 3) (6, 2) (6, 2) (6, 1) (6, 2) (6, 2) (6, 2) (6, 2) (6, 1) (7, 1) (7, 2) (7, 3) (7, 4) (7, 4) (8, 4) (8, 4) (8, 4) (8, 4) (7, 4) (7, 4) (7, 4) (7, 4) (7, 4) (7, 4) (7, 3) (7, 3) (7, 3) (7, 3) (7, 4) (7, 3) (8, 3) (8, 3) (8, 3) (8, 3) (7, 3) (7, 3) (7, 3) (7, 3) (7, 3) (7, 3) (7, 2) (7, 2) (7, 2) (7, 2)
- **Number of nodes expanded:** 112

6.2. Breath First Search

- **Found goal:** (7, 2)
- **Solution cost:** 25
- **Solution path:** (3, 2) (3, 1) (4, 1) (5, 1) (6, 1) (7, 1) (7, 2)
- **Expanded nodes:** (3, 2) (4, 2) (3, 1) (4, 3) (5, 2) (4, 1) (2, 1) (5, 3) (3, 3) (5, 1) (1, 1) (6, 3) (6, 1) (1, 2) (6, 4) (6, 2) (7, 1) (2, 2) (5, 4) (7, 2)
- **Number of nodes expanded:** 20

6.3. Iterative Deepening Search

- **Found goal:** (7, 2)
- **Solution cost:** 25
- **Solution path:** (3, 2) (3, 1) (4, 1) (5, 1) (6, 1) (7, 1) (7, 2)
- **Expanded nodes:** (3, 2) (3, 2) (3, 2) (3, 2) (4, 2) (4, 2) (4, 2) (4, 2) (3, 1) (3, 1) (3, 1) (3, 1) (4, 3) (4, 3) (4, 3) (4, 3) (5, 2) (5, 2) (5, 2) (5, 2) (3, 2) (3, 2) (3, 2) (3, 2) (3, 2) (3, 2) (3, 2) (3, 2) (4, 1) (4, 1) (4, 1) (4, 1) (2, 1) (2, 1) (2, 1) (2, 1) (5, 3) (5, 3) (5, 3) (5, 3) (4, 2) (4, 2) (4, 2) (4, 2) (3, 3) (3, 3) (3, 3) (3, 3) (4, 2) (4, 2) (4, 2) (4, 2) (5, 1) (5, 1) (5, 1) (5, 1) (3, 1) (3, 1) (3, 1) (3, 1) (3, 1) (3, 1) (3, 1) (3, 1) (1, 1) (1, 1) (1, 1) (1, 1) (6, 3) (6, 3) (6, 3) (6, 3) (4, 3) (4, 3) (4, 3) (4, 3) (4, 3) (4, 3) (4, 3) (4, 3) (6, 1) (6, 1) (6, 1) (6, 1) (4, 1) (4, 1) (4, 1) (4, 1) (1, 2) (1, 2) (1, 2) (1, 2) (2, 1) (2, 1) (2, 1) (2, 1) (6, 4) (6, 4) (6, 4) (6, 4) (6, 2) (6,

2) (6, 2) (6, 2) (5, 3) (5, 3) (5, 3) (5, 3) (6, 2) (6, 2) (6, 2) (6, 2) (7, 1) (7, 1) (7, 1) (7, 1) (5, 1) (5, 1) (5, 1) (5, 1) (2, 2) (2, 2) (2, 2) (2, 2) (1, 1) (1, 1) (1, 1) (1, 1) (6, 3) (6, 3) (6, 3) (6, 3) (5, 4) (5, 4) (5, 4) (5, 4) (6, 3) (6, 3) (6, 3) (6, 3) (6, 1) (6, 1) (6, 1) (6, 1) (6, 3) (6, 3) (6, 3) (6, 3) (6, 1) (6, 1) (6, 1) (6, 1)

- **Number of nodes expanded:** 152

6.4. Uniform Cost Search

- **Found goal:** (6, 7)
- **Solution cost:** 19
- **Solution path:** (3, 2) (3, 1) (2, 1) (1, 1) (1, 2) (2, 2) (2, 3) (1, 3) (1, 4) (1, 5) (1, 6) (1, 7) (1, 8) (2, 8) (3, 8) (4, 8) (5, 8) (6, 8) (6, 7)
- **Expanded nodes:** (3, 2) (3, 2) (3, 2) (3, 2) (3, 1) (3, 1) (3, 1) (3, 1) (3, 2) (3, 2) (3, 2) (3, 2) (2, 1) (2, 1) (2, 1) (2, 1) (3, 1) (3, 1) (3, 1) (3, 1) (1, 1) (1, 1) (1, 1) (1, 1) (1, 2) (1, 2) (1, 2) (1, 2) (2, 1) (2, 1) (2, 1) (2, 1) (2, 2) (2, 2) (2, 2) (2, 2) (1, 1) (1, 1) (1, 1) (1, 1) (2, 3) (2, 3) (2, 3) (2, 3) (1, 2) (1, 2) (1, 2) (1, 2) (4, 2) (4, 2) (4, 2) (4, 2) (2, 2) (2, 2) (2, 2) (2, 2) (1, 3) (1, 3) (1, 3) (1, 3) (4, 1) (4, 1) (4, 1) (4, 1) (4, 3) (4, 3) (4, 3) (4, 3) (5, 2) (5, 2) (5, 2) (5, 2) (3, 2) (3, 2) (3, 2) (3, 2) (1, 4) (1, 4) (1, 4) (1, 4) (2, 3) (2, 3) (2, 3) (2, 3) (5, 1) (5, 1) (5, 1) (5, 1) (3, 1) (3, 1) (3, 1) (3, 1) (3, 3) (3, 3) (3, 3) (3, 3) (1, 5) (1, 5) (1, 5) (1, 5) (1, 3) (1, 3) (1, 3) (1, 3) (4, 3) (4, 3) (4, 3) (4, 3) (1, 6) (1, 6) (1, 6) (1, 6) (2, 5) (2, 5) (2, 5) (2, 5) (1, 4) (1, 4) (1, 4) (1, 4) (1, 7) (1, 7) (1, 7) (1, 7) (2, 6) (2, 6) (2, 6) (2, 6) (1, 5) (1, 5) (1, 5) (1, 5) (1, 8) (1, 8) (1, 8) (1, 8) (2, 7) (2, 7) (2, 7) (2, 7) (1, 6) (1, 6) (1, 6) (1, 6) (2, 4) (2, 4) (2, 4) (2, 4) (2, 8) (2, 8) (2, 8) (2, 8) (1, 7) (1, 7) (1, 7) (1, 7) (3, 8) (3, 8) (3, 8) (3, 8) (2, 7) (2, 7) (2, 7) (2, 7) (1, 8) (1, 8) (1, 8) (1, 8) (5, 3) (5, 3) (5, 3) (5, 3) (4, 2) (4, 2) (4, 2) (4, 2) (4, 2) (4, 2) (4, 2) (4, 2) (2, 4) (2, 4) (2, 4) (2, 4) (4, 8) (4, 8) (4, 8) (4, 8) (2, 8) (2, 8) (2, 8) (2, 8) (2, 8) (2, 8) (2, 8) (2, 8) (2, 6) (2, 6) (2, 6) (2, 6) (6, 1) (6, 1) (6, 1) (6, 1) (4, 1) (4, 1) (4, 1) (4, 1) (6, 3) (6, 3) (6, 3) (6, 3) (4, 3) (4, 3) (4, 3) (4, 3) (5, 8) (5, 8) (5, 8) (5, 8) (4, 7) (4, 7) (4, 7) (4, 7) (3, 8) (3, 8) (3, 8) (3, 8) (2, 7) (2, 7) (2, 7) (2, 7) (6, 2) (6, 2) (6, 2) (6, 2) (5, 1) (5, 1) (5, 1) (5, 1) (6, 4) (6, 4) (6, 4) (6, 4) (6, 2) (6, 2) (6, 2) (6, 2) (6, 8) (6, 8) (6, 8) (6, 8) (4, 8) (4, 8) (4, 8) (4, 8) (4, 8) (4, 8) (4, 8) (4, 8) (6, 3) (6, 3) (6, 3) (6, 3) (6, 3) (6, 3) (6, 3) (6, 3) (5, 4) (5, 4) (5, 4) (5, 4) (6, 3) (6, 3) (6, 3) (6, 3)
- **Number of nodes expanded:** 276

6.5. Greedy Best First Search

- **Found goal:** (3, 7)
- **Solution cost:** 24
- **Solution path:** (3, 2) (3, 1) (2, 1) (1, 1) (1, 2) (2, 2) (2, 3) (1, 3) (1, 4) (1, 5) (1, 6) (1, 7) (1, 8) (2, 8) (2, 7) (2, 6) (3, 6) (3, 7)
- **Expanded nodes:** (3, 2) (3, 2) (3, 2) (3, 2) (3, 1) (3, 1) (3, 1) (3, 1) (2, 1) (2, 1) (2, 1) (2, 1) (1, 1) (1, 1) (1, 1) (1, 1) (1, 1) (1, 2) (1, 2) (1, 2) (1, 2) (1, 2) (1, 1) (1, 1) (1, 1) (1, 1) (2, 2) (2, 2) (2, 2) (2, 2) (1, 2) (1, 2) (1, 2) (1, 2) (1, 2) (2, 3) (2, 3) (2, 3) (2, 3) (1, 3) (1, 3) (1, 3) (1, 3) (1, 4) (1, 4) (1, 4) (1, 4) (1, 3) (1, 3) (1, 3) (1, 3) (1, 5) (1, 5) (1, 5) (1, 5) (1, 4) (1, 4) (1, 4) (1, 4) (1, 6) (1, 6) (1, 6) (1, 6) (1, 5) (1, 5) (1, 5) (1, 5) (1, 7) (1, 7) (1, 7) (1, 7) (1, 6) (1, 6) (1, 6) (1, 6) (1, 8) (1, 8) (1, 8) (1, 8) (1, 7) (1, 7) (1, 7) (1, 7) (2, 8) (2, 8) (2, 8) (2, 8) (1, 8) (1, 8) (1, 8) (1, 8) (2, 7) (2, 7) (2, 7) (2, 7) (2, 6) (2, 6) (2, 6) (2, 6) (2, 7) (2, 7) (2, 7) (2, 7) (3, 6) (3, 6) (3, 6) (3, 6) (2, 6) (2, 6) (2, 6) (2, 6) (3, 5) (3, 5) (3, 5) (3, 5) (2, 5) (2, 5) (2, 5) (2, 5) (1, 5) (1, 5) (1, 5) (1, 5) (3, 5) (3, 5) (3, 5) (3, 5) (3, 4) (3, 4) (3, 4) (3, 4) (2, 4) (2, 4) (2, 4) (2, 4) (2, 3) (2, 3) (2, 3) (2, 3) (3, 4) (3, 4) (3, 4) (3, 4) (3, 5) (3, 5) (3, 5) (3, 5) (3, 6) (3, 6) (3, 6) (3, 6)
- **Number of nodes expanded:** 148

6.6. A* (Manhattan Distance)

- **Found goal:** (6, 7)
- **Solution cost:** 19
- **Solution path:** (3, 2) (3, 1) (2, 1) (1, 1) (1, 2) (2, 2) (2, 3) (1, 3) (1, 4) (1, 5) (1, 6) (1, 7) (1, 8) (2, 8) (3, 8) (4, 8) (5, 8) (6, 8) (6, 7)
- **Expanded nodes:** (3, 2) (3, 2) (3, 2) (3, 2) (3, 1) (3, 1) (3, 1) (3, 1) (3, 2) (3, 2) (3, 2) (3, 2) (2, 1) (2, 1) (2, 1) (2, 1) (3, 1) (3, 1) (3, 1) (3, 1) (4, 2) (4, 2) (4, 2) (4, 2) (1, 1) (1, 1) (1, 1) (1, 1) (5, 2) (5, 2) (5, 2) (5, 2) (1, 2) (1, 2) (1, 2) (1, 2) (2, 1) (2, 1) (2, 1) (2, 1) (2, 2) (2, 2) (2, 2) (2, 2) (2, 3) (2, 3) (2, 3) (2, 3) (4, 1) (4, 1) (4, 1) (4, 1) (4, 3) (4, 3) (4, 3) (4, 3) (3, 2) (3, 2) (3, 2) (3, 2) (1, 1) (1, 1) (1, 1) (1, 1) (1, 2) (1, 2) (1, 2) (1, 2) (2, 2) (2, 2) (2, 2) (2, 2) (5, 1) (5, 1) (5, 1) (5, 1) (1, 3) (1, 3) (1, 3) (1, 3) (3, 3) (3, 3) (3, 3) (3, 3) (1, 4) (1, 4) (1, 4) (1, 4) (2, 3) (2, 3) (2, 3) (2, 3) (1, 5) (1, 5) (1, 5) (1, 5) (1, 6) (1, 6) (1, 6) (1, 6) (2, 5) (2, 5) (2, 5) (2, 5) (1, 7) (1, 7) (1, 7) (1, 7) (2, 6) (2, 6) (2, 6) (2, 6) (2, 7) (2, 7) (2, 7) (2, 7) (3, 1) (3, 1) (3, 1) (3, 1) (4, 3) (4, 3) (4, 3) (4, 3) (1, 3) (1, 3) (1, 3) (1, 3) (1, 4) (1, 4) (1, 4) (1, 4)

4) (1, 5) (1, 5) (1, 5) (1, 5) (1, 8) (1, 8) (1, 8) (1, 8) (1, 6) (1, 6) (1, 6) (1, 6) (2, 8) (2, 8) (2, 8) (2, 8) (1, 7) (1, 7) (1, 7) (1, 7) (3, 8) (3, 8) (3, 8) (3, 8) (2, 7) (2, 7) (2, 7) (2, 7) (2, 4) (2, 4) (2, 4) (2, 4) (1, 8) (1, 8) (1, 8) (1, 8) (4, 8) (4, 8) (4, 8) (4, 8) (2, 8) (2, 8) (2, 8) (2, 8) (2, 8) (2, 8) (2, 8) (2, 8) (2, 6) (2, 6) (2, 6) (2, 6) (4, 7) (4, 7) (4, 7) (4, 7) (3, 8) (3, 8) (3, 8) (3, 8) (2, 7) (2, 7) (2, 7) (2, 7) (4, 2) (4, 2) (4, 2) (4, 2) (5, 3) (5, 3) (5, 3) (5, 3) (4, 2) (4, 2) (4, 2) (4, 2) (6, 1) (6, 1) (6, 1) (6, 1) (5, 8) (5, 8) (5, 8) (5, 8) (6, 3) (6, 3) (6, 3) (6, 3) (6, 2) (6, 2) (6, 2) (6, 2) (6, 8) (6, 8) (6, 8) (6, 8) (6, 2) (6, 2) (6, 2) (6, 2)

- **Number of nodes expanded:** 232