Laboratory 1

Variant #3 (A*)
Mariya Zacharneva and Ruslan Melnyk

NOTE: It is highly recommended to access the report via Google Docs because that way, you will be able to see gifs :)
https://docs.google.com/document/d/1W7q4sexmtR-qnh1IqVaFhFEVOuKWZ0zLS1jNlyXcpms/edit?usp=sharing

Introduction

In this laboratory, we implemented the A* search algorithm to solve a 2D maze. The maze consists of walls, open spaces, a start, and an end position.

The objective is to find the path from start to end while visualizing all visited cells. The implementation includes two h(n) heuristics to evaluate their effectiveness.

A* search is an *informed* search algorithm, which means that it uses problem-specific knowledge to find solutions more efficiently. Namely, it expands the node with the lowest value of g(n) + h(n), where

- g(n) = cost to reach node (how many steps it had to take to get to the current node)
- h(n) = estimated cost to the goal (e.g., distance from the node to the goal)

Properties of the algorithm:

- **Complete**: meaning it always finds a solution if one exists, provided the search space is finite;

- **Optimal**: assuming that h(n) is
    - Admissible: never overestimates the true cost;
    - Consistent: for every node n and successor n' with step cost c, h(n) <= h(n') + c (in other words, f(n) <= f(n') i.e., f never decreases along the path). Contrary to the informed Greedy BFS, the algorithm is optimal because it considers both h and g.

- **Time**: the time complexity of A* depends on the quality of the heuristic function. The worst-case time complexity of A* is O(b^d) (like in uninformed BFS), where
    - b is the branching factor (the maximum number of children a node can have);
    - d is the depth of the shallowest goal node. However, it is important to remember, that the better the heuristic, the better the performance. In our case, we are finding the path in a maze with a single goal. Thus, b = 4 (diagonal movements are now allowed), d = number of steps to reach the goal (if such is reachable). Actual time complexity is much better.

- **Space**: the space complexity of standard A* is always O(b^d) since we need to always track every node in the graph, even ones that we've never visited and are never going to.

A* is commonly used in problems where an optimal path has to be found through a large search space. It is widely applied in: Pathfinding in Maps or Mazes, Routing Systems, Robotics, and Autonomous Vehicles (it was invented by researchers working on Shakey the Robot's path planning), AI-based Applications (decision-making systems where finding an optimal path or solution is required, such as puzzles, network routing, and AI planning tasks.

Advantages of A* algorithm:

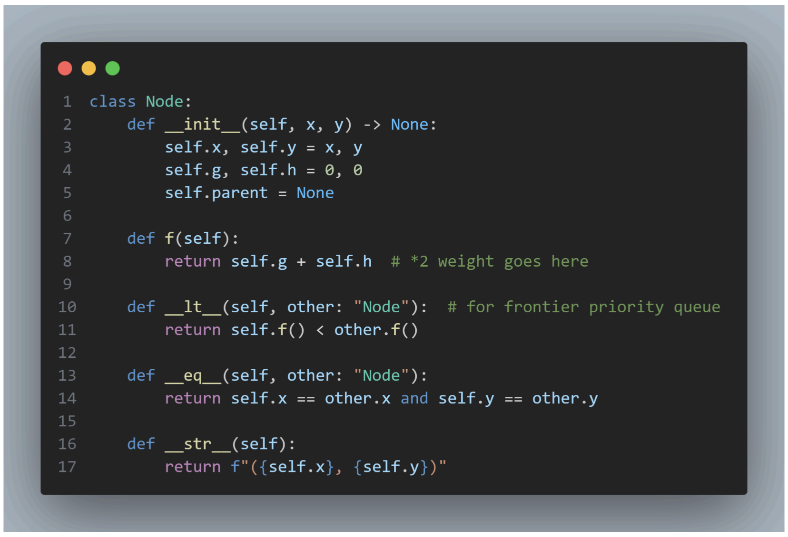
- **Optimality**: A* is guaranteed to find the shortest path if the heuristic is admissible and consistent. In contrast, DFS and BFS are not guaranteed to find the optimal path, and Greedy Best-First Search can lead to suboptimal solutions because it only considers the heuristic, not the path cost.
- **Efficiency**: When a good heuristic is used, A* is more efficient than BFS, DFS, and Uniform Cost Search. BFS explores every node level by level and can be much slower, while DFS may get stuck exploring deep branches without reaching the goal. The cheapest first approach of UCS may lead us far from the goal.
- **Flexibility**: A* can adapt to different problem domains by adjusting its heuristic, making it more flexible than DFS, DFS with Depth Limiting, and even Uniform Cost, which are less adaptable.
- **Dynamic Environment Handling**: A* can adapt to changes in the environment (e.g., new obstacles) better than algorithms like BFS or DFS, which need to restart or revisit many nodes after changes.

Limitations of A* algorithm:

- **High Space Complexity**: A* can be memory-intensive (O(b^d)) compared to DFS and DFS with Iterative Deepening, which uses much less memory (O(bd) for IDDFS).
- **Slower for Large Search Spaces**: When there is a high branching factor, A* can become slower than Greedy Best-First Search, which only focuses on the heuristic and doesn’t consider path costs. However, Uniform Cost Search is usually slower than A* since it doesn't prioritize nodes based on the heuristic at all.
- **Sensitive to Heuristic Choice**: The performance of A* depends heavily on the heuristic. If the heuristic is poor, A* can degrade to a BFS or Uniform Cost Search, making it as slow as those algorithms.
- **Real-Time Limitations**: In real-time systems or environments requiring frequent re-planning (like in robotics or games), A*’s high computational demand can be limiting compared to Bidirectional Search.
- **Scalability Issues**: For extremely large or complex problems, A* might struggle with time and space complexity compared to DFS with Iterative Deepening, which can handle larger depths by systematically deepening without storing all nodes at once.

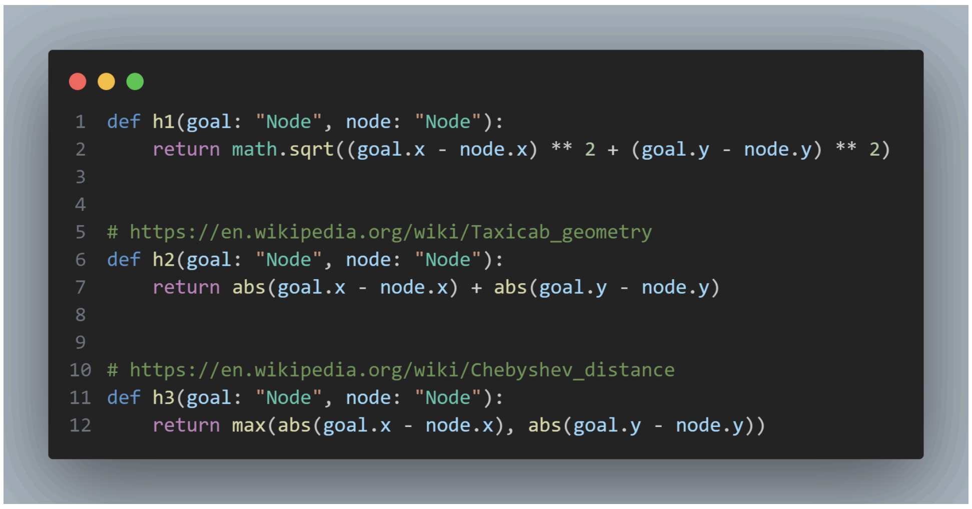
Implementation

First, we have defined a node for a cell on a grid. Each node has its coordinates, values of h and g, and methods such as “less than” and “equal” which are needed for creating a priority queue.



```python
class Node:
    def __init__(self, x, y) -> None:
        self.x, self.y = x, y
        self.g, self.h = 0, 0
        self.parent = None

    def f(self):
        return self.g + self.h  # *2 weight goes here

    def __lt__(self, other: "Node"):  # for frontier priority queue
        return self.f() < other.f()

    def __eq__(self, other: "Node"):
        return self.x == other.x and self.y == other.y

    def __str__(self):
        return f"({self.x}, {self.y})"
```

We have used Euclidian distance, manhattan distance, and Chebyshev distance as heuristics to compare their performance:

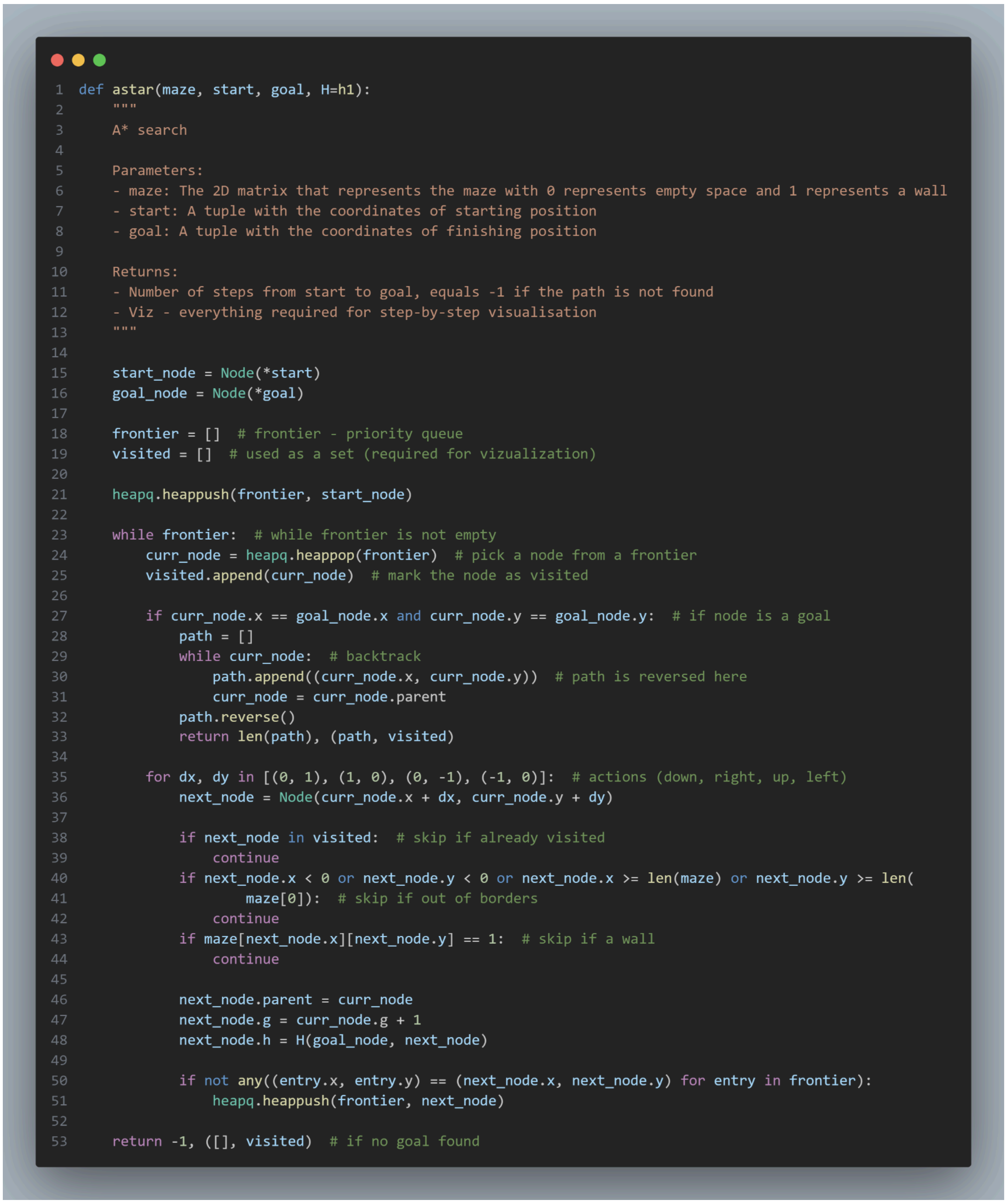


```python
def h1(goal: "Node", node: "Node"):
    return math.sqrt((goal.x - node.x) ** 2 + (goal.y - node.y) ** 2)


# https://en.wikipedia.org/wiki/Taxicab_geometry
def h2(goal: "Node", node: "Node"):
    return abs(goal.x - node.x) + abs(goal.y - node.y)


# https://en.wikipedia.org/wiki/Chebyshev_distance
def h3(goal: "Node", node: "Node"):
    return max(abs(goal.x - node.x), abs(goal.y - node.y))
```

The main algorithm:

1. We start with a node representing a start state, and a “frontier” which is a priority queue. Initially, this queue is going to contain just the start state. We also keep track of the visited nodes in a “visited” set.
2. While the frontier is not empty, we pop the node from the frontier, and we mark it as visited. The next step is to check whether the node is the goal. If it is the goal, we backtrack our way to see which actions we took to reach the goal state. Otherwise, we check for all possible next positions - nodes we can explore next (to the top, bottom, right, left).

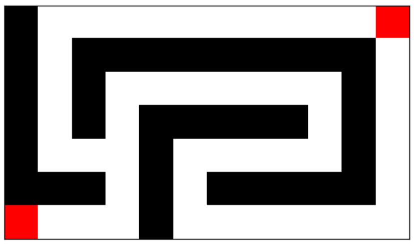
3. For each neighbor, we check a few properties. Of course, we want to skip the nodes which we have already visited. We also not going to visit walls or positions outside of the borders of the maze. Also, we want to avoid having the same node in the queue twice. If the node does not belong to any of the previous cases, we add it to the priority queue with priority h + g. So next time the node with the lowest f = h + g is popped out of the queue first.



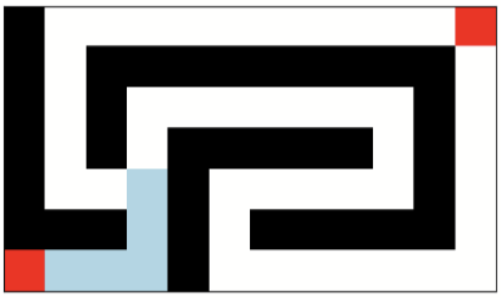
```python
def astar(maze, start, goal, H=h1):
    """
    A* search

    Parameters:
    - maze: The 2D matrix that represents the maze with 0 represents empty space and 1 represents a wall
    - start: A tuple with the coordinates of starting position
    - goal: A tuple with the coordinates of finishing position

    Returns:
    - Number of steps from start to goal, equals -1 if the path is not found
    - Viz - everything required for step-by-step visualisation
    """

    start_node = Node(*start)
    goal_node = Node(*goal)

    frontier = []  # frontier - priority queue
    visited = []  # used as a set (required for vizualization)

    heapq.heappush(frontier, start_node)

    while frontier:  # while frontier is not empty
        curr_node = heapq.heappop(frontier)  # pick a node from a frontier
        visited.append(curr_node)  # mark the node as visited

        if curr_node.x == goal_node.x and curr_node.y == goal_node.y:  # if node is a goal
            path = []
            while curr_node:  # backtrack
                path.append((curr_node.x, curr_node.y))  # path is reversed here
                curr_node = curr_node.parent
            path.reverse()
            return len(path), (path, visited)

        for dx, dy in [(0, 1), (1, 0), (0, -1), (-1, 0)]:  # actions (down, right, up, left)
            next_node = Node(curr_node.x + dx, curr_node.y + dy)

            if next_node in visited:  # skip if already visited
                continue
            if next_node.x < 0 or next_node.y < 0 or next_node.x >= len(maze) or next_node.y >= len(
                    maze[0]):  # skip if out of borders
                continue
            if maze[next_node.x][next_node.y] == 1:  # skip if a wall
                continue

            next_node.parent = curr_node
            next_node.g = curr_node.g + 1
            next_node.h = H(goal_node, next_node)

            if not any((entry.x, entry.y) == (next_node.x, next_node.y) for entry in frontier):
                heapq.heappush(frontier, next_node)

    return -1, ([], visited)  # if no goal found
```

Visualization

The behavior of the algorithm with h = Manhattan distance:

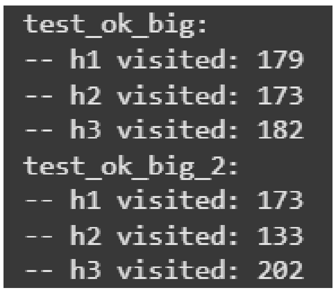


For example, it chose to go up at the decision point (see the picture below) because it decreases the Manhattan distance to the goal. At some point, though, it realizes that it is too costly to continue exploring that way and returns to the action left from the queue, which leads to an optimal solution.



Discussion

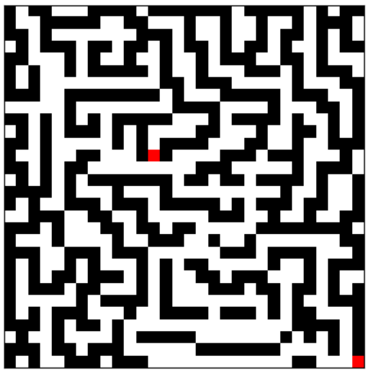
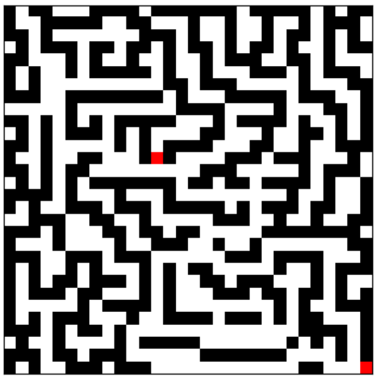
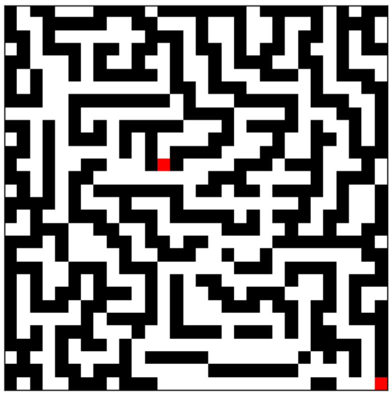
We compared the number of visited nodes in a big maze for different heuristic functions:



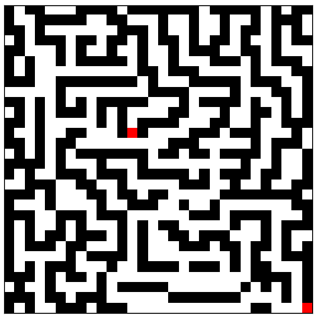
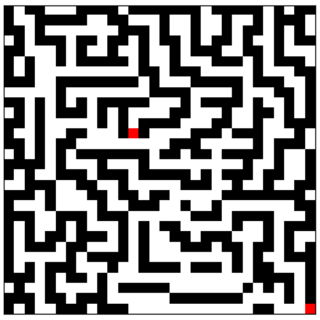
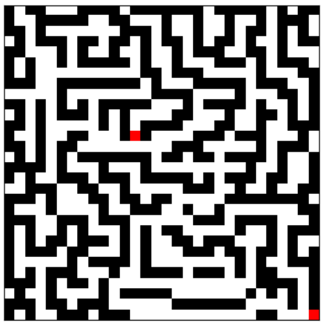
```
test_ok_big:
-- h1 visited: 179
-- h2 visited: 173
-- h3 visited: 182
test_ok_big_2:
-- h1 visited: 173
-- h2 visited: 133
-- h3 visited: 202
```

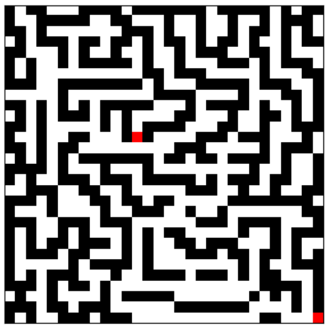
As expected, Manhattan Distance (h2) showed the best performance regarding the number of visited nodes as it shows the actual shortest path. Because diagonal movement is not allowed, Manhattan Distance is the most representative (Euclidian Distance would make more sense if diagonal movement were allowed). Chebyshev's distance underestimates sometimes.

Visual comparison:

A* (h - euclidian distance)	**A*** (h - manhattan distance)	**A*** (h - Chebyshev distance)
		

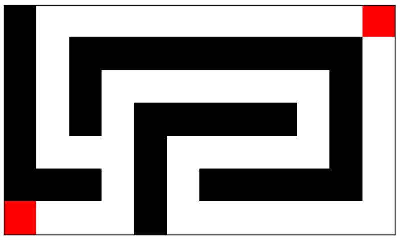
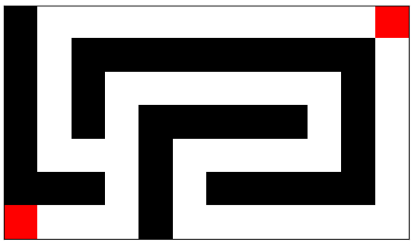
We also compared several similar algorithms:

DFS - not optimal - uninformed - faster than BFS, slower than A*	
BFS - uninformed - least performant - optimal	
A* (h - manhattan distance) - second by performance - optimal	

GBFS (h - manhattan distance) - most performant - not optimal (but we are lucky with our maze)	
UCS - does not make much sense here as step cost is = 1 so it is just BFS.	–

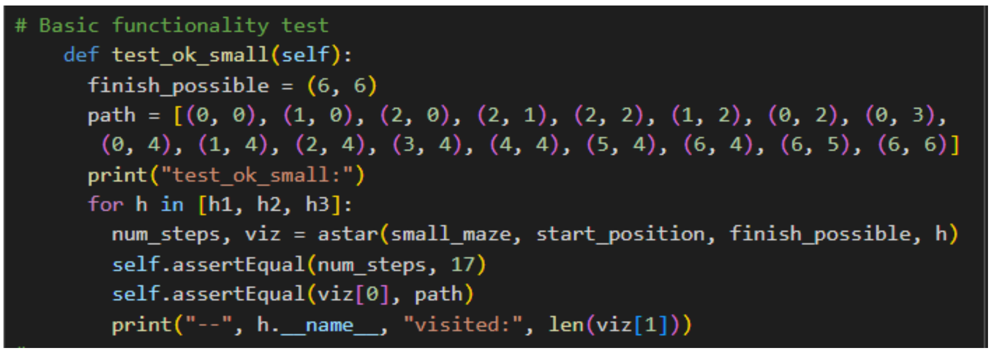
As we can see, GBFS performed best for this big maze and was lucky to be optimal.

The following case shows show that GBFP is not always optimal:

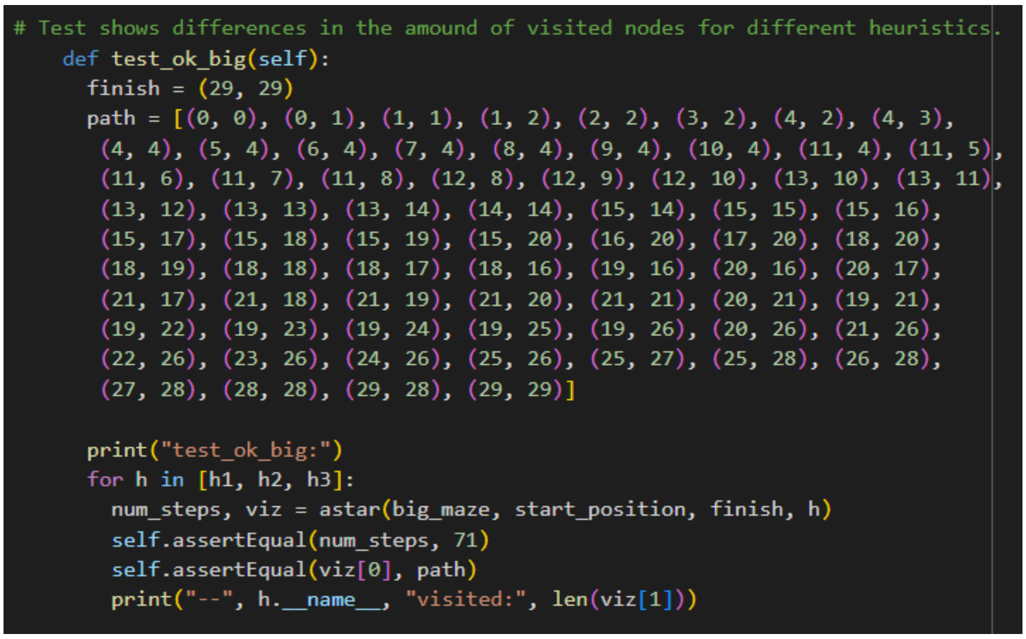
GBFS (h - manhattan distance)	**A*** (h - manhattan distance)
	

Test cases

- We have tested basic functionality in small and big mazes:

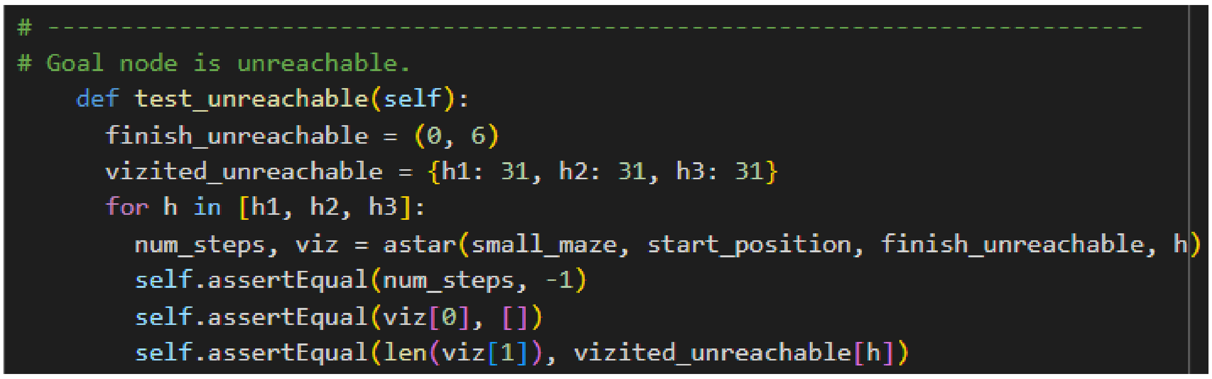


```
# Basic functionality test
    def test_ok_small(self):
      finish_possible = (6, 6)
      path = [(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (1, 2), (0, 2), (0, 3),
       (0, 4), (1, 4), (2, 4), (3, 4), (4, 4), (5, 4), (6, 4), (6, 5), (6, 6)]
      print("test_ok_small:")
      for h in [h1, h2, h3]:
        num_steps, viz = astar(small_maze, start_position, finish_possible, h)
        self.assertEqual(num_steps, 17)
        self.assertEqual(viz[0], path)
        print("--", h.__name__, "visited:", len(viz[1]))
```

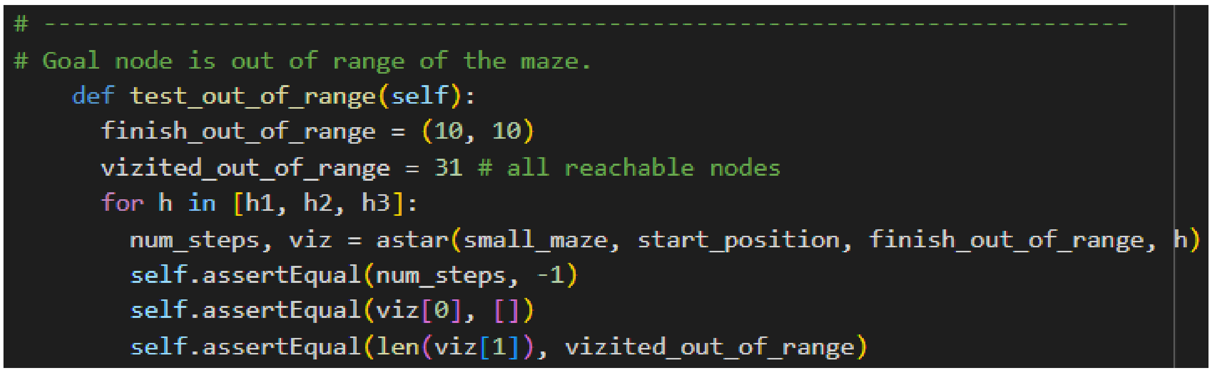


```python
# Test shows differences in the amound of visited nodes for different heuristics.
    def test_ok_big(self):
      finish = (29, 29)
      path = [(0, 0), (0, 1), (1, 1), (1, 2), (2, 2), (3, 2), (4, 2), (4, 3),
       (4, 4), (5, 4), (6, 4), (7, 4), (8, 4), (9, 4), (10, 4), (11, 4), (11, 5),
       (11, 6), (11, 7), (11, 8), (12, 8), (12, 9), (12, 10), (13, 10), (13, 11),
       (13, 12), (13, 13), (13, 14), (14, 14), (15, 14), (15, 15), (15, 16),
       (15, 17), (15, 18), (15, 19), (15, 20), (16, 20), (17, 20), (18, 20),
       (18, 19), (18, 18), (18, 17), (18, 16), (19, 16), (20, 16), (20, 17),
       (21, 17), (21, 18), (21, 19), (21, 20), (21, 21), (20, 21), (19, 21),
       (19, 22), (19, 23), (19, 24), (19, 25), (19, 26), (20, 26), (21, 26),
       (22, 26), (23, 26), (24, 26), (25, 26), (25, 27), (25, 28), (26, 28),
       (27, 28), (28, 28), (29, 28), (29, 29)]

      print("test_ok_big:")
      for h in [h1, h2, h3]:
        num_steps, viz = astar(big_maze, start_position, finish, h)
        self.assertEqual(num_steps, 71)
        self.assertEqual(viz[0], path)
        print("--", h.__name__, "visited:", len(viz[1]))
```

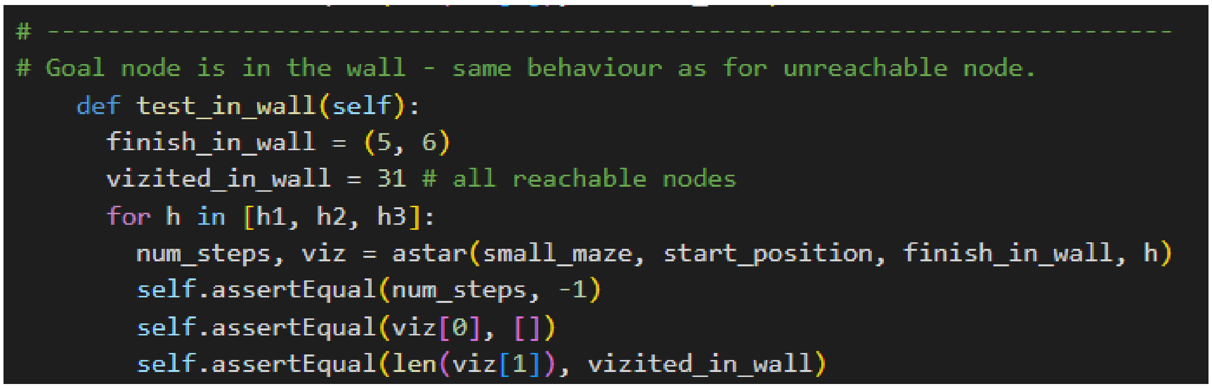
We have also tested several cases when the goal cannot be reached:



```python
# ---------------------------------------------------------------------------
# Goal node is unreachable.
    def test_unreachable(self):
      finish_unreachable = (0, 6)
      vizited_unreachable = {h1: 31, h2: 31, h3: 31}
      for h in [h1, h2, h3]:
        num_steps, viz = astar(small_maze, start_position, finish_unreachable, h)
        self.assertEqual(num_steps, -1)
        self.assertEqual(viz[0], [])
        self.assertEqual(len(viz[1]), vizited_unreachable[h])
```

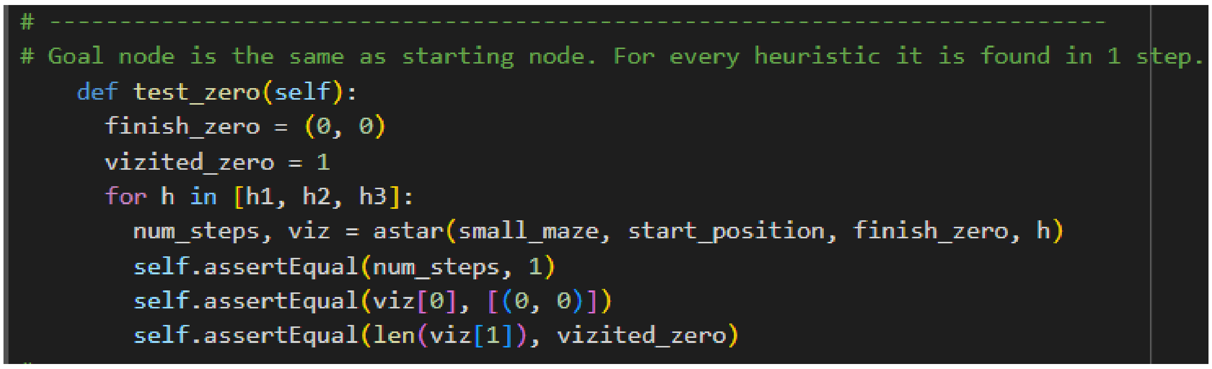


```python
# ---------------------------------------------------------------------------
# Goal node is out of range of the maze.
    def test_out_of_range(self):
      finish_out_of_range = (10, 10)
      vizited_out_of_range = 31 # all reachable nodes
      for h in [h1, h2, h3]:
        num_steps, viz = astar(small_maze, start_position, finish_out_of_range, h)
        self.assertEqual(num_steps, -1)
        self.assertEqual(viz[0], [])
        self.assertEqual(len(viz[1]), vizited_out_of_range)
```



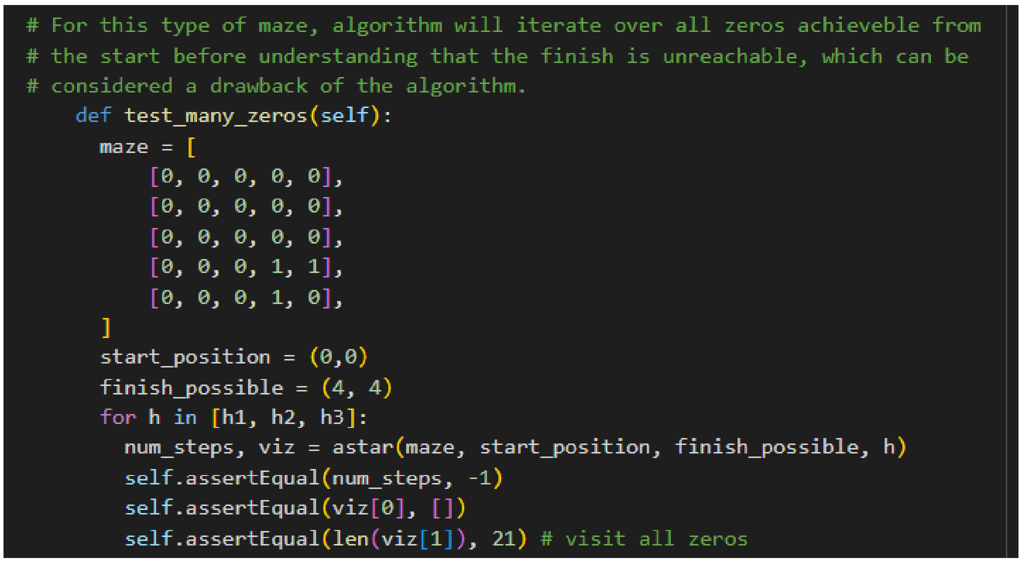
```python
# ----------------------------------------------------------------------------
# Goal node is in the wall - same behaviour as for unreachable node.
    def test_in_wall(self):
      finish_in_wall = (5, 6)
      vizited_in_wall = 31 # all reachable nodes
      for h in [h1, h2, h3]:
        num_steps, viz = astar(small_maze, start_position, finish_in_wall, h)
        self.assertEqual(num_steps, -1)
        self.assertEqual(viz[0], [])
        self.assertEqual(len(viz[1]), vizited_in_wall)
```

- Another corner case is when the starting node is the same as the goal node:



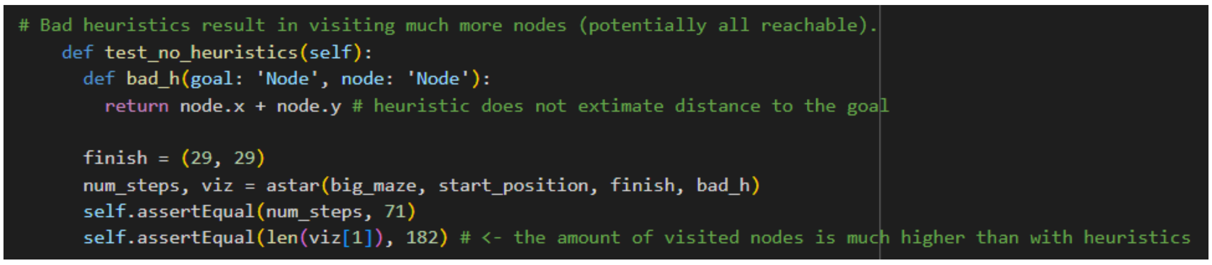
```python
# ---------------------------------------------------------------------------
# Goal node is the same as starting node. For every heuristic it is found in 1 step.
    def test_zero(self):
      finish_zero = (0, 0)
      vizited_zero = 1
      for h in [h1, h2, h3]:
        num_steps, viz = astar(small_maze, start_position, finish_zero, h)
        self.assertEqual(num_steps, 1)
        self.assertEqual(viz[0], [(0, 0)])
        self.assertEqual(len(viz[1]), vizited_zero)
```

- This test shows, that the algorithm can behave poorly when there are a huge number of zeros available from the starting point: it cannot realize, that the goal is unreachable until it iterates over all zeros

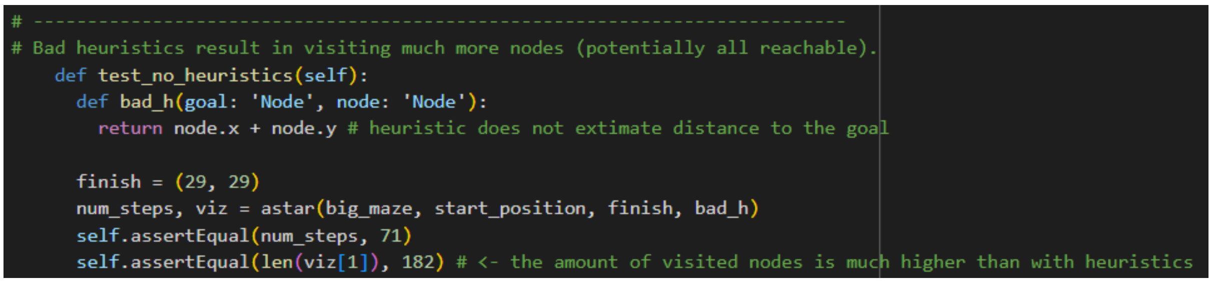


```python
# For this type of maze, algorithm will iterate over all zeros achieveble from
# the start before understanding that the finish is unreachable, which can be
# considered a drawback of the algorithm.
    def test_many_zeros(self):
      maze = [
          [0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0],
          [0, 0, 0, 1, 1],
          [0, 0, 0, 1, 0],
      ]
      start_position = (0,0)
      finish_possible = (4, 4)
      for h in [h1, h2, h3]:
        num_steps, viz = astar(maze, start_position, finish_possible, h)
        self.assertEqual(num_steps, -1)
        self.assertEqual(viz[0], [])
        self.assertEqual(len(viz[1]), 21) # visit all zeros
```

- We have also tested the behavior of the algorithm without heuristic functions and with poor ones (in these cases, the algorithm is basically pure Djkstra=UCS):



```python
# Bad heuristics result in visiting much more nodes (potentially all reachable).
  def test_no_heuristics(self):
    def bad_h(goal: 'Node', node: 'Node'):
      return node.x + node.y # heuristic does not extimate distance to the goal

    finish = (29, 29)
    num_steps, viz = astar(big_maze, start_position, finish, bad_h)
    self.assertEqual(num_steps, 71)
    self.assertEqual(len(viz[1]), 182) # <- the amount of visited nodes is much higher than with heuristics
```



```python
# ---------------------------------------------------------------------------
# Bad heuristics result in visiting much more nodes (potentially all reachable).
  def test_no_heuristics(self):
    def bad_h(goal: 'Node', node: 'Node'):
      return node.x + node.y # heuristic does not extimate distance to the goal

    finish = (29, 29)
    num_steps, viz = astar(big_maze, start_position, finish, bad_h)
    self.assertEqual(num_steps, 71)
    self.assertEqual(len(viz[1]), 182) # <- the amount of visited nodes is much higher than with heuristics
```

Conclusion

Through this laboratory experiment, we explored the A* search algorithm and analyzed its effectiveness in solving a 2D maze using different heuristic functions. We found that while A* guarantees optimality when using an admissible and consistent heuristic, its performance varies significantly based on the heuristic choice.

The results showed that the Manhattan Distance heuristic performed the best in our case, correctly estimating the shortest path without overestimating. Although useful when diagonal movement is allowed, Euclidean Distance was less efficient in this scenario. Chebyshev Distance underestimates, leading to suboptimal performance. Additionally, we confirmed that Greedy Best-First Search (GBFS) can be faster but is not always optimal, highlighting the trade-off between speed and correctness.

One of the key challenges faced was handling cases where the goal was unreachable. The algorithm had to explore a vast number of nodes before determining that no solution existed, which demonstrated the potential inefficiencies of A* in such situations. Another difficulty was ensuring that heuristics were correctly implemented and did not degrade the performance of A* into uninformed search methods like BFS.

Overall, this experiment helped us understand the strengths and weaknesses of different search algorithms, especially A*. It also showed that the effectiveness of a search algorithm depends on the specific problem and environment in which it is used (like choosing a heuristic function).