
Pick your battles

By: Bas Smit

Class: AP-S4

Version: 2 (Changed with feedback from Brice)

Date: 14/04/2024




Table of content

Table of content.....	2
1. The way out.....	3
1.2 Approach.....	3
1.3 Input.....	3
1.4 Output.....	4
1.5 Time complexity.....	4
1.6 Code.....	4
1.6.1 Initial setup.....	5
1.6.2 The BFS loop.....	6
1.6.3 Reconstructing the path.....	6
1.7 Performance & Unit tests.....	9
1.8 Terminal output.....	11
1.9 Time complexity conclusion.....	11
2. Send more money!.....	13
2.1 Approach.....	13
2.2 Input.....	13
2.3 Output.....	14
2.4 Time complexity.....	14
3. From X to Y.....	15
3.1 Approach.....	15
3.2 Input.....	15
3.3 Output.....	15
3.4 Time complexity.....	16
4. Shortest paths.....	17
4.1 Approach.....	17
4.2 Input.....	17
4.3 Output.....	17
4.4 Time complexity.....	17

1. The way out

Given is a maze modeled as a matrix $M \times N$, with cells 0 = wall, 1 = available and 2 = exit door, and a start position (row, column).

A character can move through the maze by stepping up, down, left and right, but no diagonals.

Find a shortest path, that is one with the minimum number of steps, to exit the maze.

1.2 Approach

To navigate the maze efficiently and find the shortest path from the start to the exit, I would implement a Breadth-First Search (BFS) algorithm. BFS guarantees that we reach the exit with the fewest steps possible by exploring all neighboring cells in a systematic manner, level by level, starting from the initial position. This approach ensures that once the exit is found, it is at the minimum possible distance from the start.

1.3 Input

The input for the code would be the given matrix. This would be put into the terminal by row:

Insert Row 1: 0 0 1 0 0

Insert Row 2: 0 0 1 1 0

Insert Row 3: 0 0 0 2 0

The value of each cell indicates whether it's a wall (0), an available path (1), or the exit door (2). The starting position of the character within the maze is also provided.

1.4 Output

The output would be the given matrix but with the path being displayed with 3's:

Path 1: 0 0 3 0 0

Path 2: 0 0 3 3 0

Path 3: 0 0 0 3 0

The output of the algorithm is the same matrix representation of the maze, with the cells traversed by the character marked as part of the shortest path. This is achieved by updating the corresponding cells in the matrix to indicate the path taken (e.g., changing 1s to 3s).

1.5 Time complexity

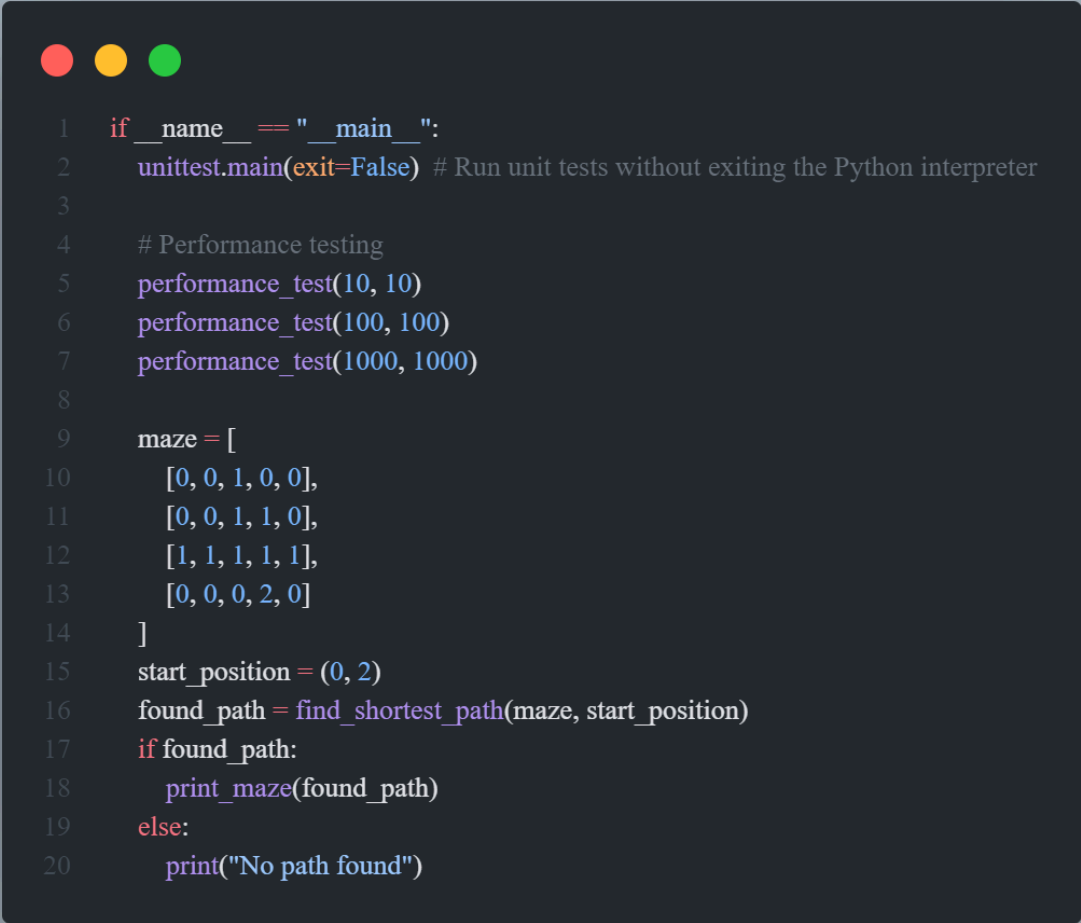
The time complexity of the BFS algorithm is $O(M*N)$, where M is the length of each row and N is the amount of rows in the given maze. This complexity arises from the need to traverse each cell in the matrix.

The BFS algorithm involves visiting each cell once and performing constant-time operations to check neighbors and update the visited status. While this approach results in a quadratic time complexity, it ensures an optimal solution by systematically exploring all possible paths from the start to the exit.

1.6 Code

The Main function looks as follows. First we run the unit tests to make sure our code is working. After that we run 3 performance tests for a maze of 10x10, 100x100 and 1000x1000.

After that we go to the main algorithm and run that with a given maze and starting position. If we find a valid path we print that to the console, otherwise we print no Path found.



```
1  if __name__ == "__main__":
2      unittest.main(exit=False) # Run unit tests without exiting the Python interpreter
3
4      # Performance testing
5      performance_test(10, 10)
6      performance_test(100, 100)
7      performance_test(1000, 1000)
8
9      maze = [
10         [0, 0, 1, 0, 0],
11         [0, 0, 1, 1, 0],
12         [1, 1, 1, 1, 1],
13         [0, 0, 0, 2, 0]
14     ]
15     start_position = (0, 2)
16     found_path = find_shortest_path(maze, start_position)
17     if found_path:
18         print_maze(found_path)
19     else:
20         print("No path found")
```

1.6.1 Initial setup

We first start by getting the dimensions of the maze. After that we define how we can move through the maze. As specified we can not move diagonally so we can only change our X and Y coordinates independently.

Then BFS is implemented using a queue (here, a deque for efficiency). The queue is initialized with a tuple containing the start row index, the start column index, and the initial number of steps (0).

A set named `visited_positions` is used to track which cells (positions) have already been visited to prevent processing the same cell multiple times and thus avoid infinite loops.

1.6.2 The BFS loop

The BFS loop starts from the part: "While queue: ". This loop keeps repeating itself till the queue is empty. In each iteration, it dequeues an element, which represents the current position in the maze and the number of steps taken to reach there.

If the current position is the exit (`maze[current_row][current_column] == 2`), the function returns the result of `trace_back_path`, which reconstructs the path taken to reach the exit.

The code explores adjacent cells as follows:

- For each direction possible from the current position, it calculates the new position (`next_row`, `next_column`).
- It checks whether this new position is within the bounds of the maze and whether it has not been visited yet and is not a wall (`maze[next_row][next_column] != 0`).
- If the new position is valid, it is added to the `visited_positions` set and enqueued with an incremented step count.

If the queue empties and no exit has been found, the function returns `None`, indicating that no path exists from the start position to the exit.

1.6.3 Reconstructing the path

Starting from the exit, it traces back to the start by following the path criteria defined (in this case, you might maintain a parent pointer or similar logic to backtrack accurately, which would need additional implementation details). As it backtracks, it marks the cells in the maze that form part of the path. These are then highlighted in green in the `print_maze` function

```

1  import unittest
2  from collections import deque
3  import timeit
4
5  def find_shortest_path(maze, start_position):
6      num_rows, num_columns = len(maze), len(maze[0])
7      directions = [(-1, 0), (1, 0), (0, -1), (0, 1)] # Up, Down, Left, Right
8      queue = deque([(start_position[0], start_position[1], 0)]) # (row_index, column_index, steps_taken)
9      visited_positions = set()
10     visited_positions.add((start_position[0], start_position[1]))
11
12     while queue:
13         current_row, current_column, steps_taken = queue.popleft()
14
15         # Exit found
16         if maze[current_row][current_column] == 2:
17             return trace_back_path(maze, start_position, (current_row, current_column))
18
19         # Check adjacent cells
20         for row_offset, column_offset in directions:
21             next_row, next_column = current_row + row_offset, current_column + column_offset
22             if 0 <= next_row < num_rows and 0 <= next_column < num_columns:
23                 if (next_row, next_column) not in visited_positions and maze[next_row][next_column] != 0:
24                     visited_positions.add((next_row, next_column))
25                     queue.append((next_row, next_column, steps_taken + 1))
26
27     return None # No path to exit was found
28
29 def trace_back_path(maze, start_position, end_position):
30     path = []
31     current_position = end_position
32     while current_position != start_position:
33         path.append(current_position)
34         current_row, current_column = current_position
35         for row_offset, column_offset in [(-1, 0), (1, 0), (0, -1), (0, 1)]: # Adjacent cells
36             next_row, next_column = current_row + row_offset, current_column + column_offset
37             if 0 <= next_row < len(maze) and 0 <= next_column < len(maze[0]) and maze[next_row][next_column] == 1:
38                 if (next_row, next_column) not in path: # Check if already included to avoid loops
39                     current_position = (next_row, next_column)
40                 break
41     path.append(start_position) # Include start position in the path
42     path.reverse() # Reverse to show path from start to end
43
44     # Mark the path in the maze
45     for row, column in path:
46         maze[row][column] = 3 # 3 could represent the path visually
47
48     return maze
49
50 def print_maze(maze):
51     green = '\033[92m' # ANSI color code for green
52     reset = '\033[0m' # ANSI reset code to return to default text color
53     for row in maze:
54         for cell in row:
55             if cell == 3:
56                 print(f'{green}{cell}{reset}', end=' ')
57             else:
58                 print(f'{cell}', end=' ')
59     print()

```


1.7 Performance & Unit tests

For the unit tests I decided to test 3 different mazes to make sure the code works as intended.

```
1  class TestMazeSolver(unittest.TestCase):
2
3      def test_path_found(self):
4          maze = [
5              [0, 0, 1, 0, 0],
6              [0, 0, 1, 1, 0],
7              [0, 0, 0, 2, 0]
8          ]
9          start_position = (0, 2)
10         result = find_shortest_path(maze, start_position)
11         expected = [
12             [0, 0, 3, 0, 0],
13             [0, 0, 3, 3, 0],
14             [0, 0, 0, 3, 0]
15         ]
16         self.assertEqual(result, expected)
17
18     def test_no_path_found(self):
19         maze = [
20             [0, 0, 1, 0, 0],
21             [0, 0, 1, 0, 0],
22             [0, 0, 0, 2, 0]
23         ]
24         start_position = (0, 2)
25         result = find_shortest_path(maze, start_position)
26         self.assertIsNone(result)
27
28     def test_small_maze(self):
29         maze = [
30             [1, 2]
31         ]
32         start_position = (0, 0)
33         result = find_shortest_path(maze, start_position)
34         expected = [
35             [3, 3]
36         ]
37         self.assertEqual(result, expected)
38
```

The performance tests were done with 3 mazes ranging from 10 to 1000 in both axes. I ran the performance tests 10 times and returned the average in Scientific time notation because the numbers were really small.

```
1 def performance_test(maze_size, path_length, number_of_runs=10):
2     # Move the maze setup inside the setup_code to ensure it is recognized
3     setup_code = f"""
4     from __main__ import find_shortest_path
5     maze = [[0] * {maze_size} for _ in range({maze_size})]
6     for i in range(min({maze_size}, {path_length})):
7         maze[i][i] = 1
8     maze[{maze_size}-1][{maze_size}-1] = 2 # Place the exit at the last cell of the diagonal
9     start_position = (0, 0)
10    """
11
12    test_code = 'find_shortest_path(maze, start_position)'
13
14    # Use timeit.repeat to run the test multiple times and gather results
15    times = timeit.repeat(setup=setup_code, stmt=test_code, number=1, repeat=number_of_runs, globals=globals())
16
17    # Calculate the average execution time
18    average_time = sum(times) / len(times)
19    print(f"Average execution time for {maze_size}x{maze_size} maze over {number_of_runs} runs: {average_time:.2e} seconds")
20
```

1.8 Terminal output

Here we can see that the time it takes from a 10x10 to a 100x100 does not differ a lot. But we can see that from 100x100 to 1000x1000 it almost differs by a factor 6.

```
PS D:\AP\Academic-prep\algorithms\pickYourBattles> python .\Maze.py
...
-----
Ran 3 tests in 0.000s

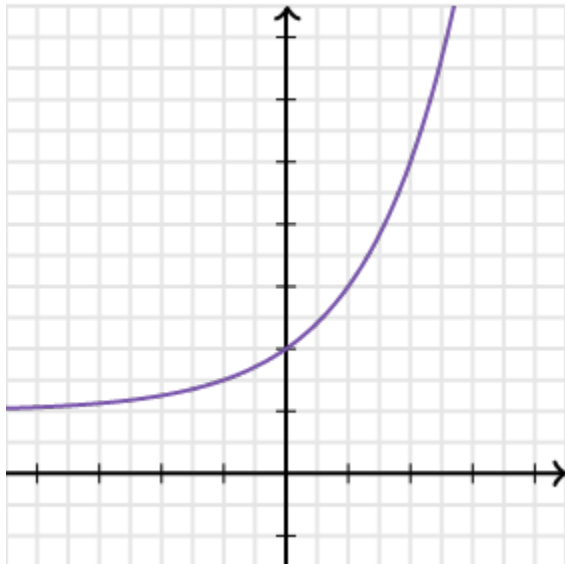
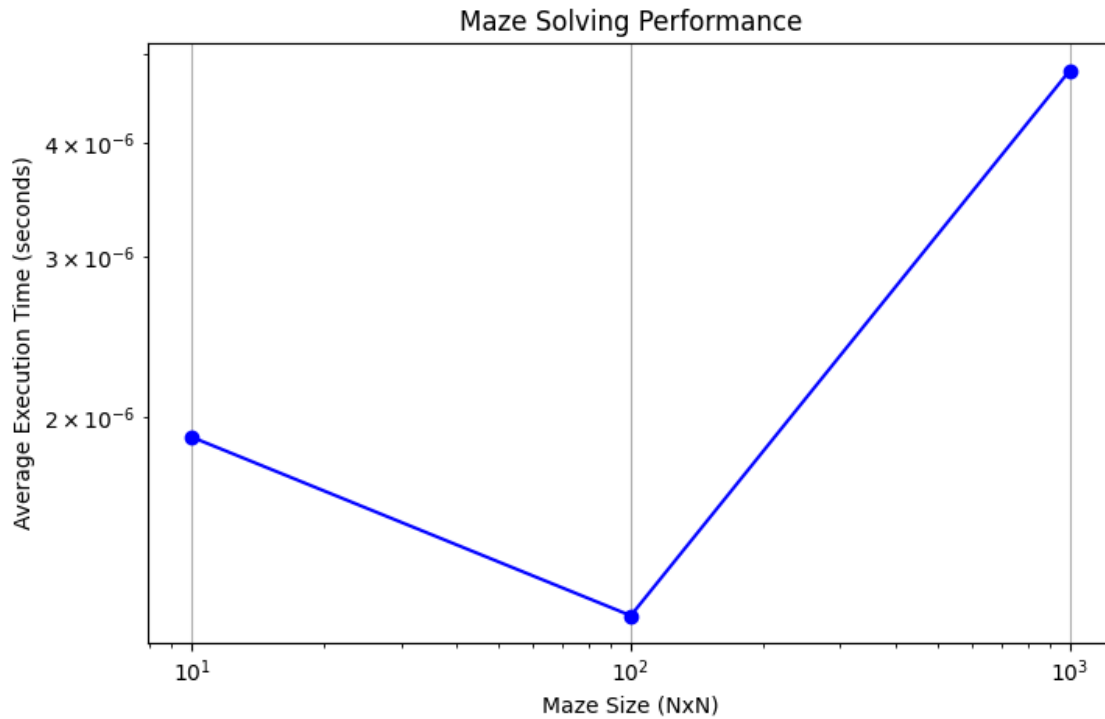
OK
Average execution time for 10x10 maze over 10 runs: 1.46e-06 seconds
Average execution time for 100x100 maze over 10 runs: 1.65e-06 seconds
Average execution time for 1000x1000 maze over 10 runs: 9.18e-06 seconds
0 0 3 0 0
0 0 3 3 0
1 1 1 3 1
0 0 0 3 0
```

At the bottom we can see the path found by the algorithm shown in green 3's

1.9 Time complexity conclusion

After adding another maze to the performance test and trying to plot the first 3 points i get the following results:

```
Average execution time for 10x10 maze over 10 runs: 1.34e-06 seconds
Average execution time for 100x100 maze over 10 runs: 1.31e-06 seconds
Average execution time for 1000x1000 maze over 10 runs: 6.51e-06 seconds
Average execution time for 10000x10000 maze over 10 runs: 1.59e-05 seconds
```



I can draw a conclusion on if my time complexity analysis is correct.
The graph above is showing exponential growth

Which would mean that this algorithm is indeed of complexity $O(N \cdot M)$
and if both are of the same size $O(N^2)$

2. Send more money!

SEND + MORE = MONEY is one well-known example of a cryptarithmic puzzle, where each letter represents a digit (0..9) and the challenge is to find the letter-digit assignment that makes the equation true.

Can you write an algorithm that can solve (when a solution exists!) cryptarithmic puzzles of the form `string1 + string2 = string3`?

Assume that the input strings are always in capital letters. Feel free to combine a standard algorithm design strategy with sensible heuristics to limit the size of the explored solution space.

2.1 Approach

Firstly I will start with assigning a 0-9 to all unique letters in the SEND + MORE = MONEY string. I will also make sure that String 1,2 and 3 are always Uppercase by using python's `.upper()` function.

Then I will calculate all possible combinations with the characters and numbers.

After this I will apply some constraints to quicken the computing time such as that the first letter of each string cannot be a 0.

Finally I will check if the remaining combinations contain a valid answer. This answer will then be returned to the console

2.2 Input

The input will be SEND + MORE = MONEY. I will extract each unique letter from this and remove the spaces and + =.

2.3 Output

The output is gonna be each letter followed up by their assigned integer. After this it will also display the SUM.

S = 2

E = 1

etc...

1234 + 1234 = 1234

2.4 Time complexity

The time complexity is $O(10^N)$ where N is the amount of unique letters in the string. This is because 0-9 contains 10 possible numbers.

aaaa + aaaa + aaaaa (A can only be 1 int between 0-9)

abab + abab + ababa (A can be 0-9 and B can be the remaining options).

Following this pattern we can reduce the original time complexity to $O(10!)$ for this problem.

3.From X to Y

Given two positive integer numbers X and Y and a third positive integer number m, convert X into Y by a sequence of the following operations:

- (a) $X := \min(X * m, Y * 2)$
- (b) $X := X - 2$
- (c) $X := X - 1$

You can perform these operations in any order and any number of times. However, the problem requests to find the conversion sequence with the minimum number of operations.

3.1 Approach

First we ask for the X and Y values. Then we must find a value for m that results in the least amount of iterations of (abc).

Then we use a for loop that goes over the operation till the given end value is reached. This function also tracks a count. After doing this we loop over all possible values for m where the iteration count is bigger than 0.

3.2 Input

For the input we simply give a value for X and y

3.3 Output

For the output we give the m where the count from the loop is the lowest

3.4 Time complexity

The time complexity for this would be $O(N)$ because there is a simple loop that holds the count variable and returns that. There would also be a function that compares each count and returns the lowest but I think this function would have a lower time complexity or the same.

4. Shortest paths

Given a directed graph with a distance function on the arcs, and a source vertex, compute the shortest paths from the source to all the other vertices.

4.1 Approach

For this problem I am going to implement Dijkstra's algorithm which ensures the shortest path between 2 vertices.

4.2 Input

The Input would be a loop that asks 2 vertices and the weight of the path between them. This would loop till all given paths are presented

4.3 Output

The output would be a List of the vertices along the shortest path and the sum of the weight of their paths.

4.4 Time complexity

I don't know what the time complexity of my Dijkstra's algorithm would be. When googling it states that there are multiple answers for this depending on how optimized your solution is.

