# Desain dan Analisis Algoritma
# "Maze Solver dengan Backtracking"

Nama Kelompok

Achmad Fauzi Aranda - 225150200111005

Gede Indra Adi Brata  -  225150200111006

**Fakultas Ilmu Komputer**

**November 2023**

**Maze.py**

Maze.py ini adalah contoh program Python untuk mencari jalur keluar dari labirin dengan algoritma backtracking + depth first search.

| Maze.py |
| --- |

```
'''
Author: Mahendra Data - https://github.com/mahendradata
References: https://www.geeksforgeeks.org/rat-in-a-maze/
'''


class Maze:

    MOVE = (
        (-1, 0),  # UP
        (1, 0),  # Down
        (0, -1),  # Left
        (0, 1),  # Right
    )

    # Public function for solving the maze
    def solve(self, maze):
        self.MAZE = maze
        self.SIZE = (len(maze), len(maze[0]))
        self.FINISH = (self.SIZE[0]-1, self.SIZE[1]-1)

        self.PATH = [[0 for i in range(self.SIZE[1])] for i in range(self.SIZE[0])]
        self.PATH[0][0] = 1

        print("Maze")
        Maze.display(maze)

        if self.__run__(0, 0):
            print(f"Solution")
            Maze.display(self.PATH)
        else:
            print("Solution does not exist")

    # Helper function to display 2D array
    def display(matrix):
        for r in matrix:
            for c in r:
```

```python
            print("." if c else "#", end=" ")
        print()

    # Private recursive function for solving the maze
    def __run__(self, row, col):
        if (row, col) == self.FINISH:
            return True

        for move_row, move_col in Maze.MOVE:
            new_row = row + move_row
            new_col = col + move_col
            if self.__is_valid__(new_row, new_col):
                self.PATH[new_row][new_col] = 1
                if self.__run__(new_row, new_col):
                    return True
                self.PATH[new_row][new_col] = 0

        return False

    # Private function to check the validity of a move
    def __is_valid__(self, row, col):
        # If out of the maze's border
        if row < 0 or col < 0 or row >= self.SIZE[0] or col >= self.SIZE[1]:
            return False

        # If hit the maze's wall
        if self.MAZE[row][col] == 0:
            return False

        # If the path has been visited
        if self.PATH[row][col] == 1:
            return False

        return True  # If the move is valid


# Driver program to test Maze class
if __name__ == "__main__":
    solver = Maze()

    maze = [[1, 0, 0, 0],
            [1, 1, 0, 1],
            [0, 1, 0, 0],
            [1, 1, 1, 1]]
    solver.solve(maze)
    print()
```

```
maze = [[1, 0, 0, 0],
        [1, 1, 1, 1],
        [0, 1, 0, 0],
        [1, 1, 0, 1]]
solver.solve(maze)
print()

maze = [[1, 0, 1, 1, 1],
        [1, 0, 1, 0, 1],
        [1, 0, 1, 1, 1],
        [1, 1, 1, 0, 1],
        [1, 0, 1, 0, 1]]
solver.solve(maze)
print()

maze = [[1, 1, 1, 1, 1],
        [1, 0, 1, 0, 1],
        [1, 1, 1, 1, 1],
        [1, 0, 1, 0, 1],
        [1, 1, 1, 1, 1]]
solver.solve(maze)
```

1. Tugas Anda pada Project 1 adalah menambahkan fungsi pada program Maze.py untuk menghitung langkah yang diperlukan untuk mencapai titik akhir dari titik awal?

Maze1.py

```
import time

class Maze:

    MOVE = (
        (-1, 0),  # UP
        (1, 0),   # Down
        (0, -1),  # Left
        (0, 1),   # Right
    )

    def solve(self, maze):
        # Initialize the maze by adding an array
        self.MAZE = maze
        # Create a size of maze (y -> row, x -> col)
        self.SIZE = (len(maze), len(maze[0]))
        # Create a finish point
        self.FINISH = (self.SIZE[0] - 1, self.SIZE[1] - 1)
        # Create a path (array 2 d that filled with zero)
        self.PATH = [[0 for i in range(self.SIZE[1])] for i in range(self.SIZE[0])]
        # Set the starting point
        self.PATH[0][0] = 1
        # Initialize steps counter (to count how many steps to reach the finish point)
        self.steps = 1
        # Display how the maze look
        print("Maze")
        Maze.display(maze)
        # Run the maze (this function using recursive to find the solution)
        if self._run_(0, 0):
            print(f"Solution (Total Steps: {self.steps})")
            Maze.display(self.PATH)
        else:
            print("Solution does not exist")

    # Helper static function to display 2D array
    @staticmethod
    def display(matrix):
        for r in matrix:
            for c in r:
                print("." if c else "#", end=" ")
```

```
        print()
    print()

# Private recursive function for solving the maze
def _run_(self, row, col):
    # If the current position is the finish point
    if (row, col) == self.FINISH:
        return True

    # Display the path using delay 1 second
    time.sleep(0.5)
    Maze.display(self.PATH)

    # Looping through the MOVE tuple and using recursive to find the solution
    for move_row, move_col in Maze.MOVE:
        # Create a new row and col based on value from Maze.MOVE
        new_row = row + move_row #1
        new_col = col + move_col
        # Check if the new row and col is valid
        if self._is_valid_(new_row, new_col):
            # Change the value of the new row and col to 1
            self.PATH[new_row][new_col] = 1
            # Increment the steps counter
            self.steps += 1
            # Using recursive to find the solution
            if self._run_(new_row, new_col):
                #Return True if the solution is found
                return True
            # Backtracking (if the solution is not found)
            self.PATH[new_row][new_col] = 0
            # Decrement the steps counter when backtracking
            self.steps -= 1


    return False

# Private function to check the validity of a move
def _is_valid_(self, row, col):
    if row < 0 or col < 0 or row >= self.SIZE[0] or col >= self.SIZE[1]:
        return False
    if self.MAZE[row][col] == 0:
        return False
    if self.PATH[row][col] == 1:
        return False
    return True
```

```
# Driver program to test Maze class
if __name__ == "__main__":
    # Define a object from Maze class
    solver = Maze()

    # Example 1
    maze = [[1, 0, 0, 0],
            [1, 1, 0, 1],
            [0, 1, 0, 0],
            [1, 1, 1, 1]]
    solver.solve(maze)
    print()

    # Example 2
    maze = [[1, 0, 0, 0],
            [1, 1, 1, 1],
            [0, 1, 0, 0],
            [1, 1, 0, 1]]
    solver.solve(maze)
    print()

    # Example 3
    maze = [[1, 0, 1, 1, 1],
            [1, 0, 1, 0, 1],
            [1, 0, 1, 1, 1],
            [1, 1, 1, 0, 1],
            [1, 0, 1, 0, 1]]
    solver.solve(maze)
    print()

    # Example 4
    maze = [[1, 1, 1, 1, 1],
            [1, 0, 1, 0, 1],
            [1, 1, 1, 1, 1],
            [1, 0, 1, 0, 1],
            [1, 1, 1, 1, 1]]
    solver.solve(maze)
```

Output :

```
Maze
. # # #
. . # .
# . # #
. . . .
Solution (Total Steps: 7)
. # # #
. . # #
# . # #
# . . .

Maze
. # # #
. . . .
# . # #
. . # .
Solution does not exist

Maze
. # . . .
. # . # .
. # . . .
. . . # .
. # . # .
Solution (Total Steps: 15)
. # . . .
. # . # .
. # . # .
. . . # .
# # # # .

Maze
. . . . .
. # . # .
. . . . .
. # . # .
. . . . .
Solution (Total Steps: 17)
. # . . .
. # . # .
. # . # .
. # . # .
. . . # .
```

Penjelasan :

Untuk bisa menghasilkan output seperti *screenshot* yang terdapat pada nomor 1 maka diperlukan sedikit modifikasi pada program sebelumnya. Disini menambahkan properti berlum self.steps yang berguna untuk menyimpan berapa kali langkah yang dilakukan untuk mencari garis finish. Kemudian properti ini akan bertambah 1 setiap kali koordinat berubah dan berkurang 1 ketika terjadi *backtracking*. Kemudian untuk menampilkan jumlah langkahnya dilakukan perintah output (print) untuk menampilkan jumlah langkah

2. Tugas Anda pada Project 2 adalah memodifikasi program Maze.py pada Project 1 agar dapat digunakan untuk mencari solusi labirin dengan titik awal dan titik akhir yang berbeda?

| Maze2.py |
| --- |
| import time<br><br>class Maze:<br><br>    MOVE = (<br>        (-1, 0),  # UP<br>        (1, 0),   # Down<br>        (0, -1),  # Left<br>        (0, 1),   # Right<br>    )<br><br>    def solve(self, maze, start=(0, 0), finish=None):<br>        # Initialize the maze by adding an array<br>        self.MAZE = maze<br>        # Create a size of maze (y -> row, x -> col)<br>        self.SIZE = (len(maze), len(maze[0]))<br>        # Create a finish point<br>        self.FINISH = finish if finish is not None else (self.SIZE[0] - 1, self.SIZE[1] - 1)<br>        # Create a path (array 2 d that filled with zero)<br>        self.PATH = [[0 for i in range(self.SIZE[1])] for i in range(self.SIZE[0])]<br>        # Define a starting point from argument<br>        start_row, start_col = start<br>        # Set the starting point<br>        self.PATH[start_row][start_col] = 1<br>        # Initialize steps counter (to count how many steps to reach the finish point)<br>        self.steps = 1<br>        # Display how the maze look<br>        print("Maze")<br>        print("start  : ", start)<br>        print("finish : ", finish)<br>        Maze.display(maze)<br>        print()<br><br>        # Run the maze (this function using recursive to find the solution)<br>        if self._run_(start_row, start_col):<br>            print(f"Solution (Total Steps: {self.steps})")<br>            Maze.display(self.PATH)<br>        else:<br>            print("Solution does not exist") |

```
# Helper static function to display 2D array
@staticmethod
def display(matrix):
    for r in matrix:
        for c in r:
            print("." if c else "#", end=" ")
        print()
    print()

# Private recursive function for solving the maze
def _run_(self, row, col):
    if (row, col) == self.FINISH:
        return True

    # Display the path using delay  second
    time.sleep(0.5)
    Maze.display(self.PATH)

    # Looping through the MOVE tuple and using recursive to find the solution
    for move_row, move_col in Maze.MOVE:
        # Create a new row and col based on value from Maze.MOVE
        new_row = row + move_row
        new_col = col + move_col
        # Check if the new row and col is valid
        if self._is_valid_(new_row, new_col):
            # Change the value of the new row and col to 1
            self.PATH[new_row][new_col] = 1
            # Increment the steps counter
            self.steps += 1
            # Using recursive to find the solution
            if self._run_(new_row, new_col):
                #Return True if the solution is found
                return True
            # Backtracking (if the solution is not found)
            self.PATH[new_row][new_col] = 0
            # Decrement the steps counter when backtracking
            self.steps -= 1
    return False

# Private function to check the validity of a move
def _is_valid_(self, row, col):
    if row < 0 or col < 0 or row >= self.SIZE[0] or col >= self.SIZE[1]:
        return False
    if self.MAZE[row][col] == 0:
        return False
    if self.PATH[row][col] == 1:
```

```
            return False

        return True


# Driver program to test Maze class
if __name__ == "__main__":
    # Define a object from Maze class
    solver = Maze()

    # Example 1
    maze = [[1, 0, 0, 0],
            [1, 1, 0, 1],
            [0, 1, 0, 0],
            [1, 1, 1, 1]]
    solver.solve(maze, start=(1, 1), finish=(2, 3))
    print()

    # Example 2
    maze = [[1, 0, 0, 0],
            [1, 1, 1, 1],
            [0, 1, 0, 0],
            [1, 1, 0, 1]]
    solver.solve(maze, start=(0, 0), finish=(3, 3))
    print()

    # Example 3
    maze = [[1, 0, 1, 1, 1],
            [1, 0, 1, 0, 1],
            [1, 0, 1, 1, 1],
            [1, 1, 1, 0, 1],
            [1, 0, 1, 0, 1]]
    solver.solve(maze, start=(0, 0), finish=(4, 4))
    print()

    # Example 4
    maze = [[1, 1, 1, 1, 1],
            [1, 0, 1, 0, 1],
            [1, 1, 1, 1, 1],
            [1, 0, 1, 0, 1],
            [1, 1, 1, 1, 1]]
    solver.solve(maze, start=(3, 0), finish=(1, 4))
```

Output :

```
Maze
start :  (1, 1)
finish :  (2, 3)

. # # #
. . # .
# . # #
. . . .

Solution does not exist

Maze
start :  (0, 0)
finish :  (3, 3)

. # # #
. . . .
# . # #
. . # .

Solution does not exist
```

```
Maze
start :  (0, 0)
finish :  (4, 4)

. # . . .
. # . # .
. # . . .
. . . # .
. # . # .

Solution (Total Steps: 15)
. # . . .
. # . # .
. # . # .
. . . # .
# # # # .


Maze
start :  (3, 0)
finish :  (1, 4)

. . . . .
. # . # .
. . . . .
. # . # .
. . . . .

Solution (Total Steps: 15)
. . . # #
. # . # .
. # . # .
. # . # .
# # . . .
```

Penjelasan :

Pada code di atas terdapat sedikit modifikasi dari kode sebelumnya, dimana pada kode ini *user* bisa memberikan input berupa koordinat terkait titik awal (*start*) maupun titik akhir (*finish*) dimana pada kode ini kita cukup untuk menambahkan properti finish pada *class* tersebut self.FINISH akan menyimpan koordinat titik akhir, ketika user memberikan input berupa titik akhir maka secara otomatis akan menjadi titik di indeks terakhir array 2 D.

3. Tugas Anda pada Project 3 adalah memodifikasi program Maze.py hasil dari Project 2 agar dapat menghasilkan jalur terpendek.

| Maze3.py |
| --- |

```
# Your task in Project 3 is to modify the Maze.py program resulting from Project 2 so that it can produce the shortest path.
# from collections import deque

from collections import deque
import time

class Maze:
    MOVE = (
        (-1, 0),  # UP
        (1, 0),   # Down
        (0, -1),  # Left
        (0, 1),   # Right
    )

    def solve(self, maze, start=(0, 0), finish=None):
        # Initialize the maze by adding an array
        self.MAZE = maze
        # Create a size of maze (y -> row, x -> col)
        self.SIZE = (len(maze), len(maze[0]))
        # Create a finish point
        self.FINISH = finish if finish is not None else(self.SIZE[0] - 1, self.SIZE[1] - 1)
        # Create a path (array 2 d that filled with zero)
        self.PATH = [[0 for i in range(self.SIZE[1])] for i in range(self.SIZE[0])]
        # Set the starting point
        start_row, start_col = start
        # Set the starting point
        self.PATH[start_row][start_col] = 1
        # Create deque to store object that following the line.
        queue = deque([(start, [])])
        # Crate set to store visited object
        visited = set()
         # Display how the maze look
        print("Maze")
        Maze.display(maze)

        # Looping through the queue
        while queue:
            # Get the current position and path
            (current, path) = queue.popleft()
```

```
        # Get the row and col from current position
        row, col = current

        # If the current position is the finish point
        # stop the program with return and print the route
        if current == self.FINISH:
            self._update_path_(path + [current])
            print(f"Solution (Total Steps: {self._steps_(path + [current])})")
            Maze.display(self.PATH)
            return

        # If the current position is not the finish point
        # add the current position to visited set
        # and looping through the MOVE tuple
        if current not in visited:
            visited.add(current)
            for move_row, move_col in Maze.MOVE:
                new_row, new_col = row + move_row, col + move_col
                new_position = (new_row, new_col)
                if self._is_valid_(new_row, new_col) and new_position not in visited:
                    queue.append((new_position, path + [current]))

    print("No solution found")

def _update_path_(self, path):
    self.PATH = [[0 for _ in range(self.SIZE[1])] for _ in range(self.SIZE[0])]
    for row, col in path:
        self.PATH[row][col] = 1
        # Display the path using delay 1 second
        time.sleep(1)
        Maze.display(self.PATH)

def _steps_(self, path):
    return len(path)

# Helper static function to display 2D array
@staticmethod
def display(matrix):
    for r in matrix:
        for c in r:
            print("." if c else "#", end=" ")
        print()
    print()

# Private function to check the validity of a move
def _is_valid_(self, row, col):
```

```
        if 0 <= row < self.SIZE[0] and 0 <= col < self.SIZE[1] and self.MAZE[row][col] == 1:
            return True
        return False


# Driver program to test Maze class
if __name__ == "__main__":
    # Define a object from Maze class
    solver = Maze()

    # Example 1
    maze = [[1, 0, 0, 0],
            [1, 1, 0, 1],
            [0, 1, 0, 0],
            [1, 1, 1, 1]]
    solver.solve(maze, start=(1, 1), finish=(2, 3))
    print()

    # Example 2
    maze = [[1, 0, 0, 0],
            [1, 1, 1, 1],
            [0, 1, 0, 0],
            [1, 1, 0, 1]]
    solver.solve(maze, start=(0, 0), finish=(3, 3))
    print()

    # Example 3
    maze = [[1, 0, 1, 1, 1],
            [1, 0, 1, 0, 1],
            [1, 0, 1, 1, 1],
            [1, 1, 1, 0, 1],
            [1, 0, 1, 0, 1]]
    solver.solve(maze, start=(0, 0), finish=(4, 4))
    print()

    # Example 4
    maze = [[1, 1, 1, 1, 1],
            [1, 0, 1, 0, 1],
            [1, 1, 1, 1, 1],
            [1, 0, 1, 0, 1],
            [1, 1, 1, 1, 1]]
    solver.solve(maze, start=(3, 0), finish=(1, 4))
```

Output :

```
Maze
start :  (1, 1)
finish :  (2, 3)

. # # #
. . # .
# . # #
. . . .

No solution found

Maze
start :  (0, 0)
finish :  (3, 3)

. # # #
. . . .
# . # #
. . # .

No solution found

Maze
start :  (0, 0)
finish :  (4, 4)

. # . . .
. # . # .
. # . . .
. . . # .
. # . # .

Solution (Total Steps: 11)
. # # # #
. # # # #
. # . . .
. . . # .
# # # # .
```

```
Maze
start :  (3, 0)
finish :  (1, 4)

. . . . .
. # . # .
. . . . .
. # . # .
. . . . .

Solution (Total Steps: 7)
# # # # #
# # # # .
. . . . .
. # # # #
# # # # #
```

Penjelasan :

Berbeda dengan sebelumnya, pada kode ini terdapat perubahan yang cukup signifikan dimana terdapat tambahan struktur data berupa Deque dimana Deque ini berfungsi untuk membantu proses Breadth First Search, dengan menggunakan algoritma ini akan didapatkan jalur terpendek dari maze tersebut. Maksud dari algoritma ini adalah dengan mencoba segala kemungkinan yang ada dengan memperhatikan cost atau harga yang diperlukan untuk mencapai titik akhir. Nantinya, cost atau harga yang paling sedikit akan menjadi jalur terpendek dari rute tersebut.