```python
from enum import Enum
from typing import List, NamedTuple, Callable, Optional
import random
from math import sqrt
# generic_search 가 포함된 폴더를 sources 로 설정
from generic_search import dfs, bfs, node_to_path, astar, Node

# Cell 과 MazeLocation 을 객체로 생성을 해둔다라?
class Cell(str, Enum):
    EMPTY   = " "
    BLOCKED = "X"
    START   = "S"
    GOAL    = "G"
    PATH    = "*"
# end of Cell


class MazeLocation(NamedTuple):
    row: int
    column: int
# end of MazeLocation


class Maze:
    def __init__(self, rows: int = 10, columns: int = 10,
                 sparseness: float = 0.2,
                 start: MazeLocation = MazeLocation(0, 0),
                 goal: MazeLocation = MazeLocation(9, 9)) -> None:
        # initialization
        self._rows: int = rows
        self._columns: int = columns
        self.start: MazeLocation = start
        self.goal: MazeLocation = goal
        # EMPTY
        self._grid: List[List[Cell]] = [[Cell.EMPTY for c in range(columns)] for r in range(rows)]
        # BLOCKED
        self._randomly_fill(rows, columns, sparseness)
        # START AND GOAL
        self._grid[start.row][start.column] = Cell.START
        self._grid[goal.row][goal.column] = Cell.GOAL
    # end of __init__

    def _randomly_fill(self, rows: int, columns: int, sparseness: float):
        for row in range(rows):
            for column in range(columns):
                if random.uniform(0, 1.0) < sparseness:
                    self._grid[row][column] = Cell.BLOCKED
    # end of _randomly_fill

    def goal_test(self, ml: MazeLocation) -> bool:
        return ml == self.goal
    # end of goal_test

    def successors(self, ml: MazeLocation) -> List[MazeLocation]:
        locations: List[MazeLocation] = []
        if ml.row + 1 < self._rows and self._grid[ml.row + 1][ml.column] != Cell.BLOCKED:
            locations.append(MazeLocation(ml.row + 1, ml.column))
        if ml.row - 1 >= 0 and self._grid[ml.row - 1][ml.column] != Cell.BLOCKED:
            locations.append(MazeLocation(ml.row - 1, ml.column))
        if ml.column + 1 < self._columns and self._grid[ml.row][ml.column + 1] != Cell.BLOCKED:
            locations.append((MazeLocation(ml.row, ml.column + 1)))
        if ml.column - 1 >= 0 and self._grid[ml.row][ml.column - 1] != Cell.BLOCKED:
            locations.append(MazeLocation(ml.row, ml.column - 1))
        return locations
    # end of successors

    def mark(self, path: List[MazeLocation]) -> None:
        for maze_location in path:
            self._grid[maze_location.row][maze_location.column] = Cell.PATH
        self._grid[self.start.row][self.start.column] = Cell.START
        self._grid[self.goal.row][self.goal.column] = Cell.GOAL
    # end of mark
```

```python
    def clear(self, path: List[MazeLocation]) -> None:
        for maze_location in path:
            self._grid[maze_location.row][maze_location.column] = Cell.EMPTY
        self._grid[self.start.row][self.start.column] = Cell.START
        self._grid[self.goal.row][self.goal.column] = Cell.GOAL
    # end of clear

    def __str__(self) -> str:
        output: str = ""
        for row in self._grid:
            output += "".join([c.value for c in row]) + "\n"
        return output
    # end of __str__
# end of Maze


# 작동 방법에 대한 이해
# Capturing?
def euclidean_distance(goal: MazeLocation) -> Callable[[MazeLocation], float]:
    def distance(ml: MazeLocation) -> float:
        xdist: int = ml.column - goal.column
        ydist: int = ml.row - goal.row
        return sqrt((xdist * xdist) + (ydist * ydist))
    return distance
# end of euclidean_distance


def manhattan_distance(goal: MazeLocation) -> Callable[[MazeLocation], float]:
    def distance(ml: MazeLocation) -> float:
        xdist: int = abs(ml.column - goal.column)
        ydist: int = abs(ml.row - goal.row)
        return xdist + ydist
    return distance
# end of manhattan_distance


if __name__ == "__main__":
    m: Maze = Maze()
    print("MAP")
    print(m)

    # dfs
    solution1: Optional[Node[MazeLocation]] = dfs(
        m.start, m.goal_test, m.successors)

    if solution1 is None:
        print("깊이 우선 탐색으로 길을 찾을 수 없습니다!")
    else:
        path1: List[MazeLocation] = node_to_path(solution1)
        m.mark(path1)
        print("MARK")
        print(m)
        m.clear(path1)

    # bfs
    solution2: Optional[Node[MazeLocation]] = bfs(
        m.start, m.goal_test, m.successors)

    if solution2 is None:
        print("너비 우선 탐색으로 길을 찾을 수 없습니다!")
    else:
        path2: List[MazeLocation] = node_to_path(solution2)
        m.mark(path2)
        print("MARK")
        print(m)
        m.clear(path2)

    # A* algorithm
    distance: Callable[[MazeLocation], float] = manhattan_distance(m.goal)
    solution3: Optional[Node[MazeLocation]] = astar(
```

```
    m.start, m.goal_test, m.successors, distance)

if solution3 is None:
    print("A* 알고리즘으로 길을 찾을 수 없습니다!")
else:
    path3: List[MazeLocation] = node_to_path(solution3)
    m.mark(path3)
    print("MARK")
    print(m)
    m.clear(path3)
```