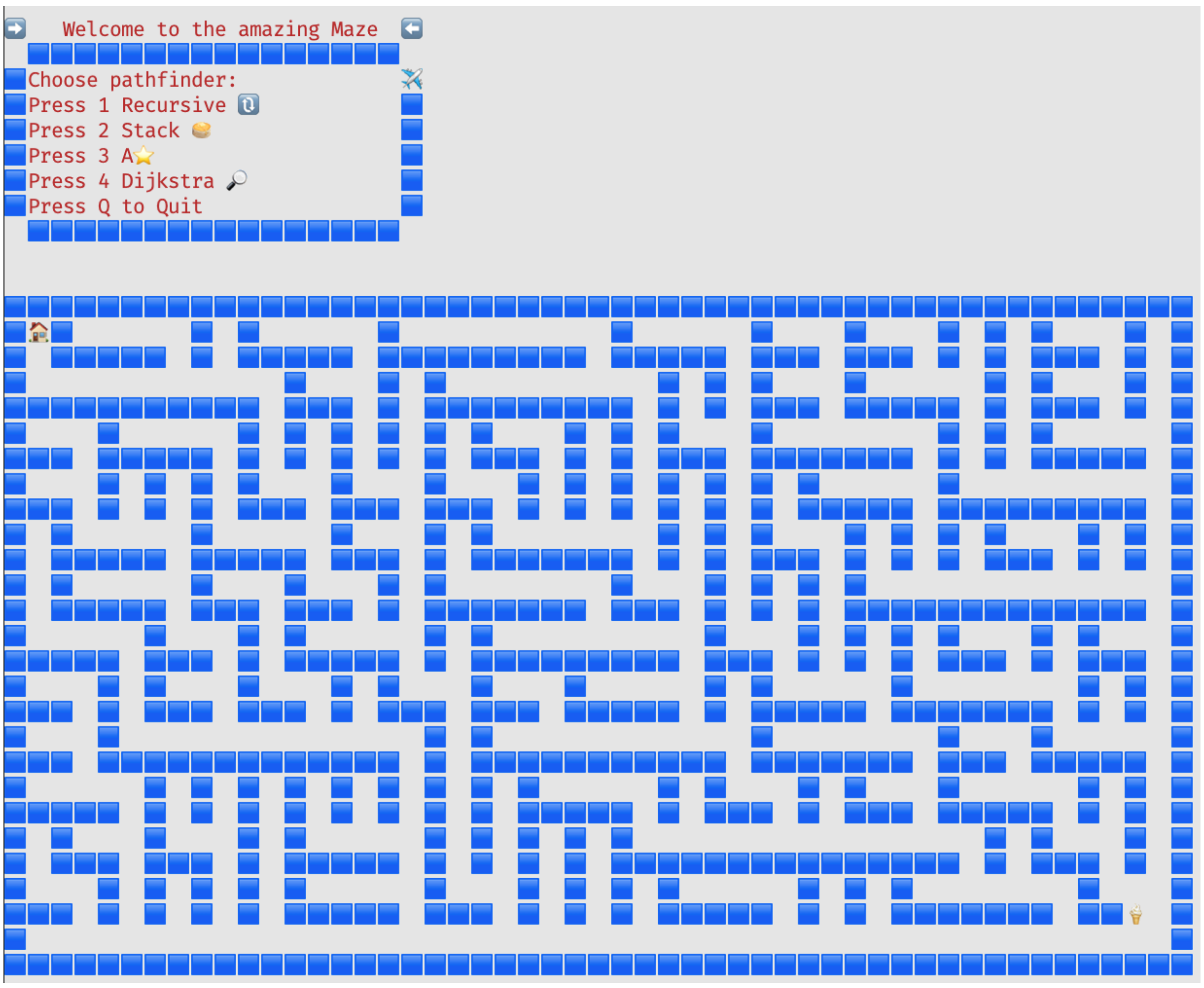
Project second part

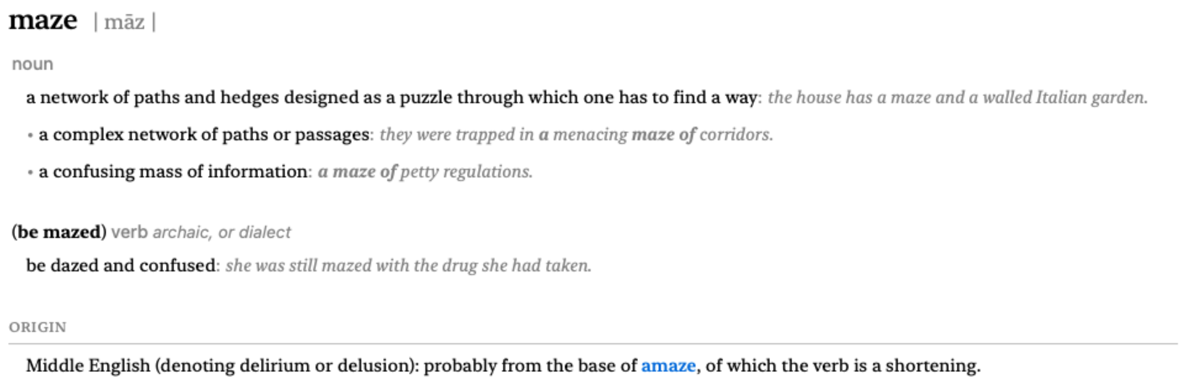
Maze: Maze Generator, and Solver



```
➡️   Welcome to the amazing Maze  ⬅️
  🟦🟦🟦🟦🟦🟦🟦🟦🟦🟦🟦🟦🟦🟦🟦🟦
🟦Choose pathfinder:              ✈️
🟦Press 1 Recursive 🔃             🟦
🟦Press 2 Stack 🥞                 🟦
🟦Press 3 A⭐                      🟦
🟦Press 4 Dijkstra 🔎              🟦
🟦Press Q to Quit                 🟦
  🟦🟦🟦🟦🟦🟦🟦🟦🟦🟦🟦🟦🟦🟦🟦🟦
```

Project description

The project's tasks revolves around the concept of Maze or Labyrinth:



maze | māz |

noun

a network of paths and hedges designed as a puzzle through which one has to find a way: *the house has a maze and a walled Italian garden.*

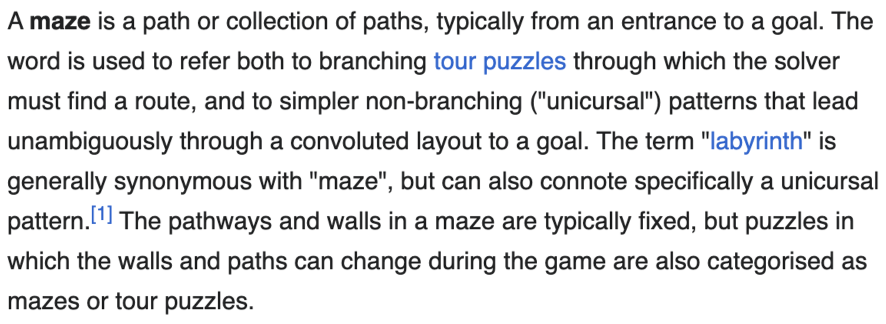
- a complex network of paths or passages: *they were trapped in **a** menacing **maze of** corridors.*
- a confusing mass of information: ***a maze of** petty regulations.*

(be mazed) verb *archaic, or dialect*

be dazed and confused: *she was still mazed with the drug she had taken.*

ORIGIN

Middle English (denoting delirium or delusion): probably from the base of amaze, of which the verb is a shortening.



A **maze** is a path or collection of paths, typically from an entrance to a goal. The word is used to refer both to branching tour puzzles through which the solver must find a route, and to simpler non-branching ("unicursal") patterns that lead unambiguously through a convoluted layout to a goal. The term "labyrinth" is generally synonymous with "maze", but can also connote specifically a unicursal pattern.[1] The pathways and walls in a maze are typically fixed, but puzzles in which the walls and paths can change during the game are also categorised as mazes or tour puzzles.

Brief description of the Starter-Kit.

At the beginning of the course the students will be provided access to a codebase to start the project and complete the given assignments. This codebase, also known as the Starter-Kit, is structured following the Model View Controller design pattern (from now on MVC) which separates those three concerns.
To briefly summarize, the Model in this project attains at the definition of the data structures used to represent the maze and the algorithms employed to automatically find a path between the starting and the end points, as well as the method responsible for manual playing of the maze.

For the maze data structures, we can think of a multi-dimensional (table) or a jagged array (containing only sub-arrays of equal lengths) as the most suitable data structures to encapsulate the characteristics of a maze. Where each element in this table will be a tile in the grid of the maze.

Using integers we can describe the possible tiles of the maze, by using four different values, namely: **1** for the **Begin**; **2** for the **End**; **-1** for a **Wall**; **0** for a **Passage** (*empty tile*).

In the provided Maze class a number of useful constructors and auxiliary methods is provided.
Auxiliary methods:

- Two methods that allows generating the maze grid from a string (input example can be found in `Program.cs`): `ToMazeArray`, `ToMazeMDArray` are provided and used in the relevant constructor (`Maze(string lines)`).
- `static int CountNotVisited(int[][] maze)`
  returns the total number of empty spaces (passage/path tiles, value: 0) in the maze.
- `static bool IsValidPos(int[][] array, int newRow, int newColumn)`
  ensures position (newRow, newColumn) is within the array/grid bounds.
- `public bool IsValidMove(int newRow, int newColumn)`
  makes sure the position is within the maze array bounds, and does not correspond to a wall.

It is obviously possible to use a number of algorithms in order to generate the maze. This is one of the programming assignments of this project part:
Implement different versions of method `GenerateMaze`.

The Path Finding algorithms (not provided) will be added to the Model section by implementing the given `IPathFinder` interface.
The provided `ManualPathFinder` which also implements the `IPathFinder` interface handles the addition of new visited tiles/positions when the manual mode is chosen: manual playing.

Based on the input from console the Controllers will take care of running the chosen algorithm and visualizing the maze (demo video).

The View namespace contains two classes for the Menu and for the Maze display/visualization.
An improvement of this visualization can be achieved by using the data structures we have already seen in this course (visualization assignment).

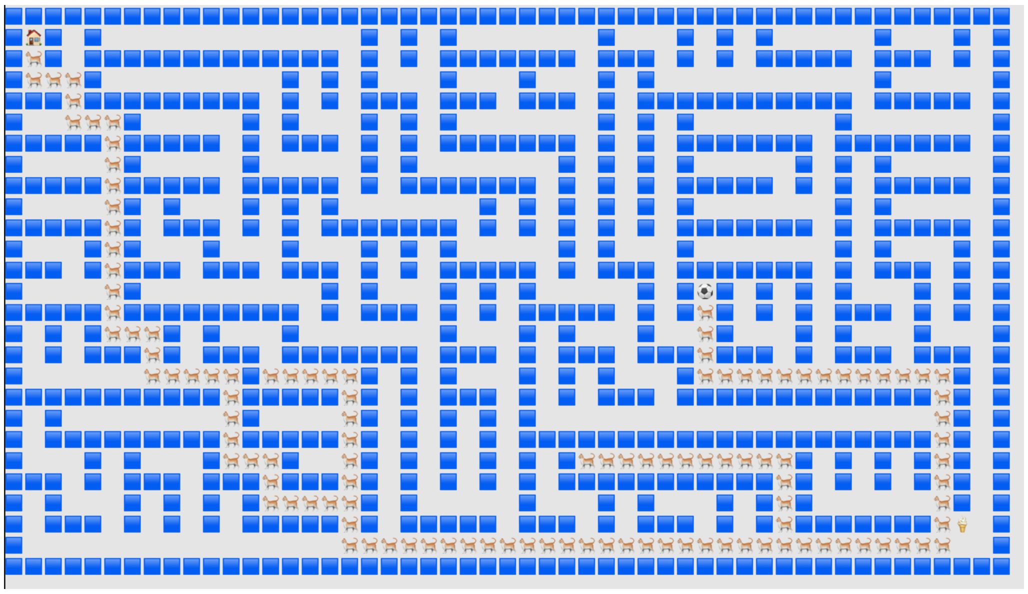
Requirements

The programming assignments belongs to *three* different categories.
The final deliverables covering these three sub-parts must contain:

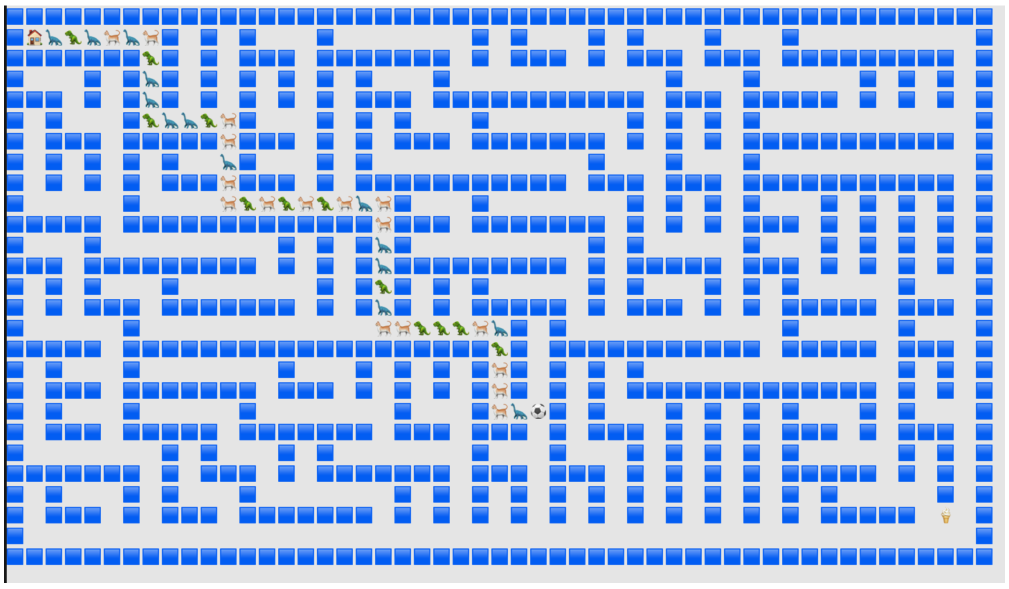
- Source code with clear algorithm implementation.
- Short report (≤ 2 pages) including:
  - Problem definition and algorithm chosen.
  - Complexity analysis.
  - Results summary (code / screenshots / videos / output logs).

Visualization part

Using the starter-kit the current (valid) position in the maze is marked using just one character/symbol (🐕):



We can improve this visualization by using a number of symbols and display them in such a way that the behavior of a queue is shown, namely a new symbol appears at the end of the queue (which corresponds the Start/home) and all the already shown symbols are shifted forward towards the beginning of this queue (last visited position). Please have a look at the picture below and to the provided video.
The `DisplayMaze` method overloads are responsible for the visualization of the maze and how the maze is visited from the start until the end is reached. Therefore the necessary changes to perform this task will involve these methods.
Hint: In order to print this sets of characters at a particular position in the grid it is necessary to map each position belonging to the collection of visited positions to a specific character in the set of characters. Think about data structures that might help to achieve this goal. Moreover, since we want to simulate the behavior of an actual queue the head and all the other element will not change, they will just be shifted forward, at the end of the queue (start of the maze/end of the queue) a new element will be added. **Hint**: a *shift* in the character' set or a change in the collection of visited points might help.



Maze solver:

This task is related to the implementation of the `IPathFinder` interface.

The method:

`void FindPath(Maze maze, int[] pos, Queue<int[]> visitedPositions);`

takes as input an instance of a maze, an initial position, and a queue containing the visited positions, initially empty. This queue will eventually contain all the positions from the Start until the End, or with other words the path.

The head/front of this queue will contain the first added position (Start) and the back will be the last visited position, eventually when a correct path is found: the End of the Maze.

Implementing the simple backtracking solution both recursive and iterative (using a stack), together with at least another pathfinding implementation (for instance Dijkstra) constitutes the minimum requirement.

Maze Generator:

This task is related to this constructor defined in the Maze class:

`public Maze(int rows, int cols) => GenerateMaze(rows, cols);`

Implement the method `GenerateMaze` which takes two integers as input: the number of rows and columns of the maze grid.

Make use of Binary Tree or Depth/Breadth First search algorithms and/or other algorithms in order to carve the grid of the maze that will need to allow for at least a path connecting the Start to the End.

Set the right values (walls, passages, start, end) in the jagged array and multidimensional array describing the grid. Finally set the Start and End properties of the Maze.

For each algorithm provide a different version of the generate algorithm, using constructor overload and a flag for the method's choice is also possible.

Evaluation criteria and rubrics:

This project consists of a number of programming assignments covering the following topics of the course, which the student is suppose to master and be able to proficiently **apply**:

- Data structures: Arrays, Jagged arrays, Multi-Dimensional arrays, Dictionary (Hash-Tables), Dynamic arrays (Lists in C#), Queue, Stack.
- Data Backtracking: Recursion, Stack (iterative in place of a recursive solution).
- Data Binary Trees, Breadth/Depth-First Search.
- Data Graphs, Single Source Shortest Path algorithms: Dijkstra, A*

The learning goals of this project part entails

- the application of a number of data structures and algorithms learnt during this course.
- the ability to generalize and adapt techniques and algorithms learnt/used during the course in different settings to the proposed problem.
- the ability to research, understand, and apply new algorithms based on those taught in the course (excellent).

The deliverables will be evaluated using the following rubrics. To obtain a *sufficient* grade *all* the sub parts must be evaluated as sufficient. An overall *excellent* evaluation can be given if at least *two* of the three sections are evaluated as excellent.

Visualization CLO5

Level	**Criteria**
Not Sufficient <5.5	Proposed implementation of the given task is incomplete or incorrect. Code fails to produce the visualization of the queue as described in the assignment's description. The student shows little ability to understand and apply the course concepts.
Sufficient 5.5:8.4	Proposed implementation of the given task is correct. Code produces the visualization of the queue as described in the assignment's description. The student shows sufficient ability to undertand and apply the course concepts.
Excellent >=8.5	Multiple possible correct implementations of the given task are provided and correctly explained. All implementations produce the visualization of the queue as described in the assignment's description. The student shows outstanding ability to understand and apply the course concepts.

Maze Solver CLO6, CLO7

Level	Criteria
Not Sufficient <5.5	Maze solving is incomplete or incorrect. Code fails to find a valid path or contains major errors. The student shows little ability to understand and apply the course concepts.
Sufficient 5.5:8.4	Implements a solver using the backtracking (recursive/iterative) algorithms and another (based on learnt algorithms) pathfinding algorithm. Produces correct solutions for solvable mazes. The student shows sufficient ability to understand and apply the course concepts.
Excellent >=8.5	Researches, understands and implements multiple solving strategies correctly. Code is efficient, handles edge cases, and demonstrates strong application of graph traversal concepts. Solutions are optimized and adaptable. The student shows outstanding ability to understand and apply the course concepts, and to independently reserach, understand and apply other algorithms. The student shows outstanding ability to understand and apply the course concepts, and to independently research, understand and apply other algorithms.

Maze Generator CLO6, CLO7

Level	Criteria
Not Sufficient <5.5	Maze generation is incomplete or incorrect. Code fails to produce a valid maze or ignores algorithmic principles (e.g., paths are disconnected, walls not consistent). No clear application of a maze generation algorithm. The student shows little ability to understand and apply the course concepts.
Sufficient 5.5:8.4	Implements a maze generator using at least one algorithm. Maze is solvable and correctly reflects algorithmic logic, though efficiency, randomness, or variety may be limited. The student shows sufficient ability to undertand and apply the course concepts.
Excellent >=8.5	Researches, understands and implements multiple maze generation algorithms correctly. Code is efficient, modular, and produces varied, complex mazes. Demonstrates strong application of algorithmic knowledge, with clear justification for algorithm choice and design. The student shows outstanding ability to understand and apply the course concepts, and to independently research, understand and apply other algorithms.