Tilt Game!

Contents

[Problem Definition 2](#_Toc163361781)

[Problem Specifications 2](#_Toc163361782)

[Project requirements 3](#_Toc163361783)

[**Required Implementation** 3](#_Toc163361784)

[**Input** 3](#_Toc163361785)

[**Output** 4](#_Toc163361786)

[**Test Cases (TBD…)** 4](#_Toc163361787)

[**Sample Test:** 4](#_Toc163361788)

[**Complete Test:** 5](#_Toc163361789)

[Deliverables 5](#_Toc163361790)

[**Implementation (60%)** 5](#_Toc163361791)

[**Document (40%)** 5](#_Toc163361792)

[**Allowed Codes** 5](#_Toc163361793)

[Milestones 5](#_Toc163361794)

[BONUSES 6](#_Toc163361795)

# **Problem Definition**

Tilt is a puzzle game played on a Tilt **board**: an square grid, where grid square contains either a fixed **obstacle**, a movable **slider**, or neither (the grid square is **empty**).

A Tilt **move** consists of tilting a board in any of the four cardinal directions (up, left, down or right), causing each slider to slide maximally in that direction until it hits either: the edge of the board, an obstacle, or another slider that cannot slide any further. A Tilt move results in a new Tilt board **configuration**.

Given a Tilt board with one grid square *t* labeled as the **target**, a sequence of Tilt moves solves Tilt puzzle if applying the moves in sequence to results in a Tilt board configuration that contains a slider in the target square *t*.

The figure below shows a small 6.006 Tilt puzzle and a solution using the fewest possible moves . Obstacles are shown in black, movable sliders are circles, and the target square is shaded gray.

A grid of squares with arrows and symbols

Description automatically generated with medium confidence

# **Problem Specifications**

We represent an board configuration using 2D grid matrix, where the grid square in row y (from the top) and column x (from the left) is B[y][x], equal to either character ’#’ (an obstacle), ’o’ (a slider), or ’.’ (neither). We represent the target square by a tuple t = (xt, yt) where the puzzle is solved when B[yt][xt] = ’o’.

It’s required to find the **minimum number of moves** that solve this game (if any).

Your task is **to read from a file** containing an square grid with the sliders ‘o’, the obstacles ‘#’ and the empty cells ‘.’ as well as the target square represented in a tuple t = (xt, yt).

1. **First, try to detect whether the board is solvable or not.**
2. **If solvable, then describe an algorithm to return a move sequence that solves the puzzle in the fewest moves. If not solvable return that no such move sequence exists.**

If the puzzle’s shortest solution uses moves (if the puzzle is not solvable), your algorithm should run in .

Where is the number of successors for each configuration (i.e., up, down, left and right), so we have at most successors for each board configuration. If is not solvable, every non-solved configuration may need to be searched which is at most configurations. Where , as each of the s sliders can be on any of the empty cells. Where b is the number of fixed obstacles.

**NOTE:** the memory footprint of the board-state should be minimized to avoid out-of-memory and to easily check for duplicate states

# **Project requirements**

## **Required Implementation**

|  |  |
| --- | --- |
| **Requirement** | **Performance** |
| 1. **Read** a **file** containing the initial board configuration and the target location | **Time:** should be **bounded by O(N2)**, N is a one dimension of board |
| 1. **Implement** the move function that generates the next board state after tilting the board in a given direction | **Time:** should be **bounded by O(N2)**, N is a one dimension of board |
| 1. **Solve** the game by constructing graph/tree till you get the goal state with the shortest path from the initial state with the lowest number of moves. | **Time:**  **solvable:** **bounded by O(N2 r2)**, r (=4) is the max number of successors for a board state  **Not solvable:** **bounded by** , b, s are the number of fixed obstacles and movable sliders, respectively |
| 1. **Output file:**    1. Statement that indicates if the given board is solvable or not.    2. If solvable, write down the sequence of moves   **In Sample Cases:** write down the STEP-by-STEP puzzle configuration till reaching the goal. | **Time:**  **Sample:** should be **bounded by O(N2 x k)**  **Complete:** should be **bounded by O(k)** |

## **Input**

**Game\_board File:**

The file contains a 2d-array of size representing the board game, also it contains a tuple holding the target location (i.e., column\_num, row\_num).

Here is an example of the input:

5

#, #, ., ., .

., o, #, ., .

., ., o, ., .

., ., ., ., .

#, #, #, ., .

4, 3

## **Output**

**Game Solution File containing:**

1. Statement that indicates if the given board is solvable or not.
2. If solvable, write down
   * The min number of moves that solves the puzzle
   * the sequence of moves
   * **In Sample Cases:** write down the puzzle configuration after each move from the solution.

Here is the output for the given input sample.

**Solvable**

**Min number of moves: 2**

**Sequence of moves: down, right**

**Initial**

{{**'#'**, **'#'**, **'.'**, **'.'**, **'.'**},   
{**'.'**, **'o'**, **'#'**, **'.'**, **'.'**},  
{**'.'**, **'.'**, **'o'**, **'.'**, **'.'**},  
{**'.'**, **'.'**, **'.'**, **'.'**, **'.'**},  
{**'#'**, **'#'**, **'#'**, **'.'**, **'.'**}}

**Down**

{{**'#'**, **'#'**, **'.'**, **'.'**, **'.'**},  
{**'.'**, **'.'**, **'#'**, **'.'**, **'.'**},  
{**'.'**, **'.'**, **'.'**, **'.'**, **'.'**},  
{**'.'**, **'o'**, **'o'**, **'.'**, **'.'**},  
{**'#'**, **'#'**, **'#'**, **'.'**, **'.'**}}

**Right**

{{**'#'**, **'#'**, **'.'**, **'.'**, **'.'**},  
{**'.'**, **'.'**, **'#'**, **'.'**, **'.'**},  
{**'.'**, **'.'**, **'.'**, **'.'**, **'.'**},  
{**'.'**, **'.'**, **'.'**, **'o'**, **'o'**},  
{**'#'**, **'#'**, **'#'**, **'.'**, **'.'**}}

**Reached goal since one slider reached the target (4,3).**

## **Test Cases**

### **Sample Test:**

* **Goal:** test the correctness
* **Given:** set of small boards with the expected O/P
* **Cases: check** [**attached folder**](Sample%20Tests)

### **Complete Test: (TBD…)**

* **Goal:** test the efficiency (beside correctness)
* **Given:** 3 levels
  1. Small: board size O(10’s)
  2. Medium: board size O(100’s)
  3. Large: board size O(1000’s)

# **Deliverables**

## **Implementation (60%)**

1. **Read** data from file containing the nxn board and the target location.
2. **Determine** whether the given board is solvable or not?
3. **Implement** an efficient algorithm to solve the game and provide step by step movements till you reach the final solvable board from the initial given board state.

## **Document (40%)**

1. Entire source code
2. Detailed analysis of your code
3. Execution time of "**Complete Test**" cases

## **Allowed Codes**

Nothing

# **Milestones**

|  |  |  |
| --- | --- | --- |
|  | **Deliverables** | **Due to** |
| **Milestone1** | 1. **Read** the input file 2. **Implement** the move function that generates the next board state after tilting the board in a given direction 3. **Solve** the game by constructing graph/tree till you get the goal state with the lowest number of moves. 4. **Output file:**    1. Statement that indicates if the given board is solvable or not.    2. If solvable, write down the sequence of moves       * **In Sample Cases:** write down the STEP-by-STEP puzzle configuration till reaching the goal. | Final Delivery  [LAB EXAM WEEK] |
| * + **MUST** deliver the required tasks and **ENSURE** it’s worked correctly   + **MUST** deliver in your scheduled time (TO BE ANNOUNCED) | | |

# **BONUSES**

1. **Simulation user friendly GUI** which allows you to rewind the search one step (movement) at a time over a generic board configuration. (e.g. generate (NxN) matrix of picture boxes and swap them according to each movement to simulate the solution step by step).