Approach:

Part 1:

- Created a simulator to create a blank grid and initialized with start and goal cells.
- Designed simple iterative algorithm to reach the goal based on the difference between the coordinates of the current cell and the goal.
- Extended simulator to create obstacles at randomly assigned cells. The simple algorithm would not function for a grid with obstacles.
- Developed a new algorithm based on Breadth First Search to find the shortest, and hence most efficient, path to the goal.

Part 2:

- Extended simulator to contain cells with weights higher than other cells. Breadth First Search based algorithm would yield the shortest path, but not the minimum cost path, leading to it's failure in finding the most efficient path.
- Developed a new algorithm based on A* algorithm (or Dijkstra's with a Heuristic function).
- Wrote basic unit-tests for methods of the simulator and path-finding algorithm.
- Extended functionality to include command-line arguments.

Design Decisions:

- Chose BFS-based algorithm for Part 1, because in case of relatively trivial graphs, BFS outperforms most popular algorithms, like A*, and is relatively simple to implement.
- First algorithm was caused to fail by introducing a weighted grid, causing the BFS-based algorithm to provide sub-optimal solutions as it would ignore the weighted cells.
- Chose A*-based algorithm for Part 2, since the heuristic would minimize the number of paths that require to be explored, thereby reducing computation time.
- Wrote function to generate a grid with random start, goal, dimensions and obstacles.
- Performed validation checks on start and goal (start and goal cannot be the same, start and goal cannot be placed where there are obstacles). If failing to meet the required criterion, the start and goal cells are regenerated repeatedly until they comply.
- Pseudo-Randomized count of obstacles and weights by constraining them to the dimensions of the grid (So as to avoid over-population or under-population).
- Developed both algorithms by maintaining a constant grid, by entering hard-coded values, so as to be able to consistently observe, debug and tweak the algorithms.
- Once suitably mature, the grid was again randomized, and further changes were made to the algorithm as necessary.
- Used a min-heap to simulate priority queue functionality, so as to be able to retrieve lowest cost cells at low complexity.
- Implemented an admissible heuristic function in the algorithm for the second part of the problem, so as to be able to retrieve the cost to the goal from the current cell, and to use in

determination of the total cost in combination with the cost generated in reaching the current state.
- Performed check on whether or not goal state can be reached, in order to display suitable output.

Solution:

In order to build an effective simulator, first the function for building the grid was designed to generate random dimensions, and populate the grid with obstacles and start and goal cells at random. This would ensure that no over-fitted solution could perform.
Next, to solve the grid, an algorithm based on BFS was developed. The algorithm employs the "flood-fill" process, in which all the neighbors of the current cell are explored and expanded in an exponential manner, whilst checking if the newly encountered cells haven't already been visited, or are not present as obstacles. This technique works well for trivial graphs, and hence is employed in generating the first algorithm.

For the second task, the grid dynamics were upgraded to additionally populate the grid with weights for the cells, which essentially meant an additional cost to traverse through that cell, in contrast with the first simulator, where all cells other than the obstacles were equally weighted. In such a case, the BFS-based algorithm will fail since it has no consideration for the weights of the cells. It will disregard the weights and go on to find the shortest path solution, which could potentially turn out to be the sub-optimal solution in a weighted grid. The weights were also pseudo-randomized within a range of values.

In order to overcome the new dynamics, an A* based algorithm was used. A min-heap was adapted as a priority queue, in order to be able to retrieve the low cost elements at a reduced complexity. The algorithm works similarly to the first algorithm, except that it considers the cost taken to reach any cell from the start point, in addition to the cost to the goal which is given by the heuristic function. The next cell yielding the minimum total cost in included into the path, thereby leaving a much reduced number of path evaluations as compared to an algorithm without a heuristic. The A* algorithm is very efficient in finding the minimum cost path to the goal, and does so at a potentially reduced complexity than Dijkstra's algorithm.

In addition to the simulators and algorithms, basic unit-tests are written to check certain grid design related methods and parameters, as well as the message for solvability.

The functionality of the algorithm was also extended to include 2 optional command-line arguments for the dimensions (height and width) of the grid. In the event that no arguments are provided, or insufficient arguments are provided, or invalid arguments are provided, all 3 cases default to a randomly sized grid.

The algorithms are capable of solving large grids, but beyond a point (~400x400), the computations become excessive causing big delays.

NOTE: Please use Python3