2021/22 Frist Term
INM702 Programming and Mathematics for Artificial Intelligence
Report – Task 1
Finding the Fastest Path in a Grid

By Suen Chi Hang
Chi.Suen@city.ac.uk

Introduction

Given a rectangular grid of random numbers representing the time cost of each cell, we are required to find the fastest path from top left corner to the destination at the bottom right. One can only move right, left, top or down by one cell each step.

Coding architecture of the game

Using Python, a class “grid” is built to initialize, populate and display each cell of the grid with random numbers. A sub-class for populating numbers with different probability distribution might be built as they won’t share the same numbers, but alternatively I built a class that accepts “random_function” as input, allowing different probability distributions.

A class “path” is built to initialize, populate and display the fastest path using certain algorithm. A sub-class for populating the path might be built as different algorithm might create different results, but alternatively I built a class that accepts “path_function” as input to allow different algorithms to be used. Using functions as input is more convenient and it avoids creating sub-classes. Code: https://github.com/chihangs/suen_INM702-Individual-task1

Naïve approach

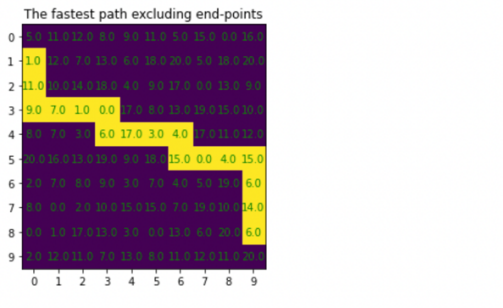
Imagine a grid to turn 45 degrees with starting point at the top. Its first half looks like a tree diagram and its second half an inverted one. For each step one just needs to go down left or right – equivalent to right and down in the original grid. This is also like a Pascal triangle, which can be used to calculate the no. of combinations of right and down for passing each cell. All combinations of right and down are calculated to get the minimum path, with total no. of steps = (no. of rows -1) + (no. of columns -1). This takes too much computation time and seems to increase exponentially with the size.

To improve this, naïve_split_path is created by splitting the grid into two triangles along a diagonal, as the path must pass through at least one cell of the diagonal, which is each treated as end-point of first part and start-point of the second part. Run the naïve algorithm for each diagonal cell twice for each part (which is much smaller than the whole triangle) and sum up the result, then we can choose the diagonal cell through which the combined path is the shortest. For non-square grid, shorter side is used to choose “diagonal” cells. This improved algorithm is ~100 times faster than original naïve one for 12x12 grid.

Dijkstra Algorithm

The same classes “grid” and “path” are used as those in naïve approach, but Dijkstra function is built as input function to calculate the path. This algorithm focuses on all and only adjacent nodes to calculate distance (time in our case) for each step, and then ignore those already calculated and visited. It calculates and stores the distance travelled to the current node and check if neighbour nodes can be reached from current node with shorter distance/time. It moves forward each step to an adjacent node that is the nearest and update which preceding node leading to shortest distance, until it finally reaches the destination. Compared with my naïve approach, although it gives the same result for the same 10x10 grid shown below, it doesn’t waste time computing all combinations, and it is

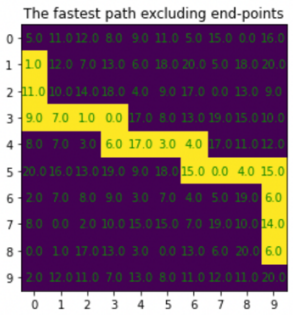
```
Using naive_split_path algorithm, the time cost is 119.0
```




```
Computation time by naive_split_path: 0.03287196159362793
```

about 5 times faster than my improved version of naïve algorithm, and it can readily generate results for the shortest path to all other nodes, not just the destination. Another advantage is that as it will keep checking unvisited neighbour that may be behind it, unlike my naïve approach which only checks path with down and right moves, it is able to find the shortest path that may need turning around long consecutive high value block of cells – very rare in random simulation, but it may happen in real life (such as mountains).

```
Using Dijkstra algorithm, the time cost is 119.0
```

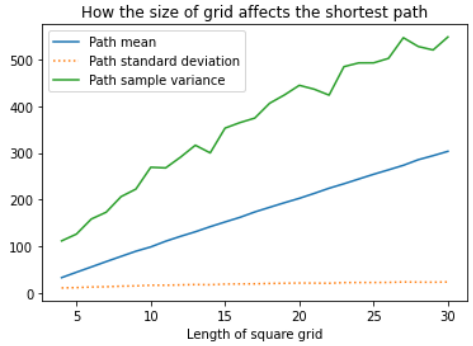



```
Computation time by Dijkstra: 0.0060901641845703125
```

Factors affecting the path

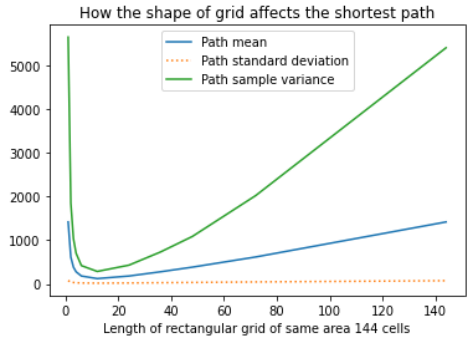
1. Size of the grid

For simplicity, we use square grid, using the uniform integer distribution (from 0 to 20 inclusive) with mean 10. The length (no. of cells for each side) is varied and for each length the simulation is run 1000 times. The path mean increases almost linearly with the length of the grid, while the path variance increases more than linearly (looks like quadratically) and with more noises.



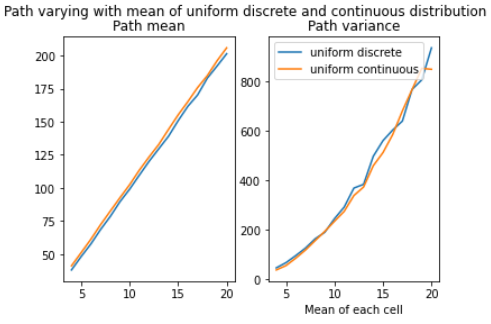
2. Shape of the grid

Simulation is done with the same setting except that the shape impact is studied by varying both length and width, but holding area unchanged at 144 cells. The path mean and variance are minimum at the square shape of 12x12 grid, and increases when the shape becomes flatter – in extreme case, when it becomes 1 x 144 grid, no of cells involved is more and the path has only one choice and it cannot benefit from choosing smaller random numbers.



3. Discrete or continuous distribution

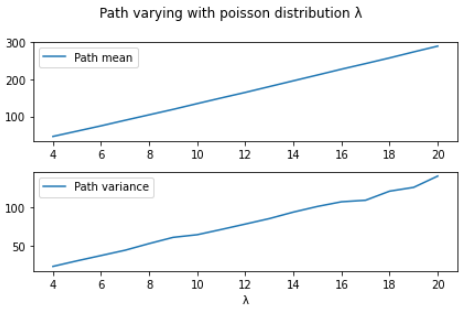
The path mean is proportional to both means of uniform discrete and continuous distribution, with the discrete distribution giving slightly smaller path mean, which can be explained by its slightly higher variance giving more chance to choose smaller random number in the path. The path variance seems to increase quadratically with the means of both uniform distributions, with the discrete one giving slightly larger path variance.



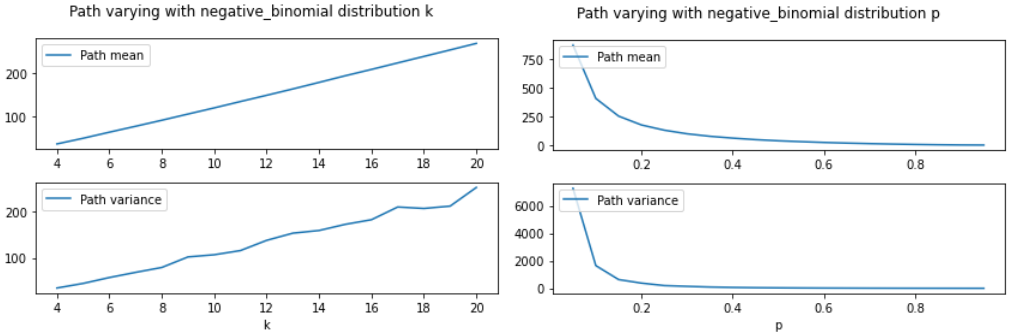
4. Different probability distributions

Distributions are studied by the created function "distribution_factor", simulating 1000 times for 10x10 grid as default unless otherwise stated.

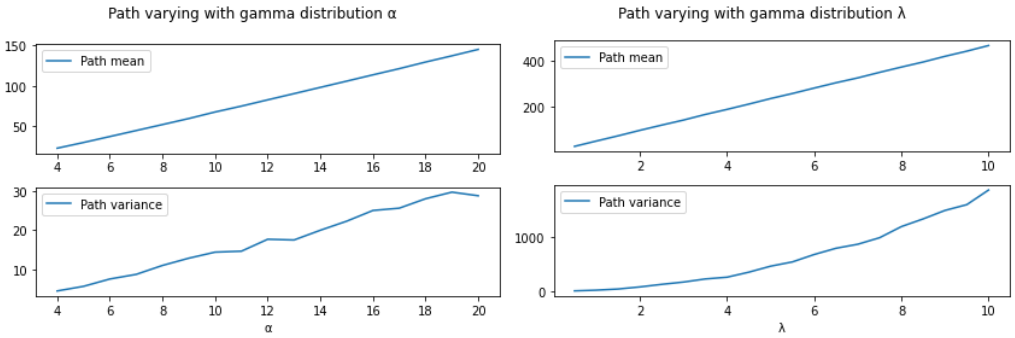
Poisson distribution – the path mean and variance both look linear and roughly proportional to λ which is the mean and variance of Poisson distribution.



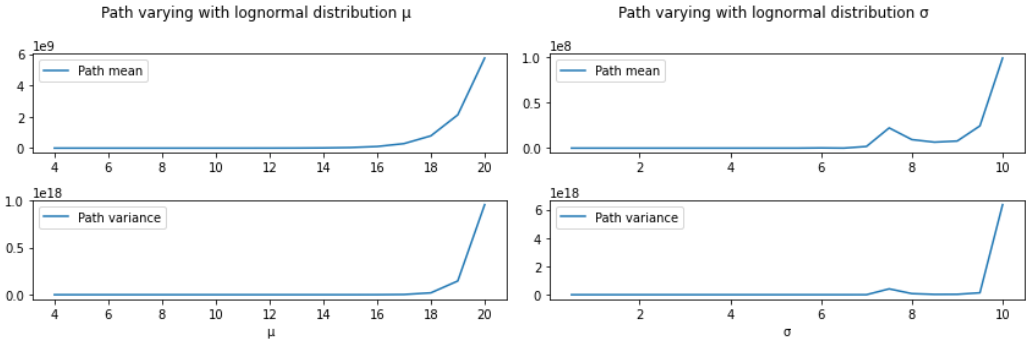
Negative binomial distribution – both the path mean and variance increase linearly and almost proportionally with k, but decrease exponentially with p.



Gamma distribution - both the path mean and variance increase linearly and almost proportionally with α. The path mean increases linearly and almost proportionally with λ while the path variance increases exponentially with λ.



Lognormal distribution – both the path mean and variance increases exponentially with μ, σ.



Characteristics of the path mean and variance

The path mean looks proportional to the mean of each cell, increases almost linearly with the length of the grid, and increases when the shape of grid becomes flatter (less square).

The path variance is roughly proportional to the variance of each cell with some noises, increases with length of the grid and when the shape of grid becomes flatter. For equal mean, the distribution with larger variance produces shorter path but bigger path variance.

Reference

Wikipedia editors, 2002-2021, Wikipedia, Accessed Nov 2021,
<https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm>

Barngrader, 2013, Youtube, accessed Nov 2021,
<https://www.youtube.com/watch?v=0nVYi3o161A>

Hunter J., Dale D., Firing E., Droettboom M. and the Matplotlib development team, ©2002 – 2012, Matplotlib, accessed Nov 2021,
<https://matplotlib.org/stable/gallery/images_contours_and_fields/image_annotated_heatmap.html>