CMPE 130 Project Report

Luke Dillon, Matthew Spadaro, Nathan Luc

5/12/18

Introduction

This project is attempting to generate the most random key possible. Random key generation is interesting because it is used heavily for encryption in data security and also for creating random passwords. The most immediate application of this code would be as a random password generator or a random key generator that can be used to encrypt data. The people who would be using our program are people that encrypt data using a key and what our program would do for that would generate a key that is the source of the encryption. Our code would be difficult to break the encryption as the outcomes would be completely random with a variety of sizes. We used Dijkstra's algorithm because of the purpose of finding the shortest path within a graph. Compared to other encryption methods our work is not based off of a table that has ciphers within in it such as AES. Our work generates a random key from a random graph that has a very low chance of producing a duplicate key. Emulating true randomness is challenging because the "random" keys may seem random but they might be following some pattern that the user is unaware of.

Related Work

Dijkstra's algorithm use in the real world currently in predominantly used in routing for GPS, and commonly used in routing. This algorithm has not been used for encryption that we have seen. Our work does not relate to many other works using this algorithm. As mentioned before other works use this algorithm in order to find the shortest path to a destination and use that shortest path in order to travel. Our program finds the shortest path in a graph and uses the nodes as a key, giving a random key to a user, while actually storing a graph.

Furthermore, we pulled a lot of inspiration from random level generation in game design. While each level may never truly be random, you want each one to feel unique to the player. The idea was, if we replaced the floor of a level with characters and found the shortest path from start to finish, what would the result be. In our reference there is a github linked of a project that helped spur our idea into reality.

Method Description

The code generates a graph that is connected to random keys with random weights. The graph initially contains A-Z, a-z, and 0-9. There is a function (rando_gen) that creates the connected nodes and their weights. You then select two random nodes and find the shortest path between them. The code is implemented using Python. The random keys and weights are generated using a built-in python random library. It is important to not that the random library python is not entirely secure, but we decided to use it to create a proof of concept. Likewise, this program adds another layer of randomness by finding the shortest path in the graph to generate a random key. As noted above, the shortest path is found using Dijkstra's algorithm. These data structures and algorithms were chosen because they provide the most simple and efficient way to generate a random graph and to find the shortest path through the random graph. Each execution of code is almost unique as there is a very low chance of generating the same key because of all the variables in play. There a total of 62 nodes that are created and each connection is random and the start and ending node will be different. The link below will take you to the github repository for the code: https://github.com/Ordo-S/CMPE130/blob/master/RandomKeyGenerator

Analysis of Algorithms

Dijkstra's algorithm is really the star of the show here. We chose this algorithm because we knew it was fast and efficient. Although, the function of our application doesn't necessarily require speed and efficiency, it adds a good touch to not keep users waiting too long. Dijkstra finds the shortest path in the graph and returns all of the nodes that it touches. Our Dijkstra's algorithm takes in a graph, a source, and a destination. It returns the path of nodes that were connected and it returns the total cost that it took to reach the destination. The input size is always the same because we are not using a dynamic graph. In future work, we would like to make this a dynamic graph. We did try to see if there was a correlation between runtime and cost of the path. However, we did not see a correlation as is evidenced in Figure 2. Note that the runtimes will not match what is seen in Figure 1 because we added a print() function so we could see the cost in this test case. We used a Python function called getsizeof() to measure the size of the graph we used. It turns out that the graph is 6416 bytes. This is a decent size of memory. That is one reason why we would like to make a dynamic graph as opposed to our static graph.

Pseudo code

```
Function random_gen ( size is 1, chars is  ascii_letter +  digits)
 // used to create random number or letter
Graph is  random_gen // creates a random graph using that is either A-Z or a-z or 0-9
source= random_gen; //creates a random source node
```

```
destination =random_gen; // creates a random destination node
Function Dijkstra(graph, source, destination) // use to find the shortest path in the graph
Return key

Function Dijsktra (graph,source,destination)

Path is an empty set

Previous is destination

       If source is destination
              While prev is not none
                     Add previous onto path
                     Previous is  predecessor get previous and none
                     Print shortest path
       If source is not destination
              For neighbor of the graph source
                     If neighbor is not visited
                            New distance is graph source neighbor  + distance source
                     If new distance is less than distance of neighbor
                            Neighbor distance is new distance
                            Neighbor predecessors is source

Source is added onto visited
Unvisted is an empty array

For k in the graph
       If k is not visited
              Unvisited k is the distance from k

X is the min of unvisited and key of the unvisited

 Dijsktra (graph,source,destination)
```

Results

The result of the program was that a random key was generated but some of the time we would encounter an error. Each run created a random key of varying size but there was about a 25% chance that the program will run into an error. The error came from the process in which the node would hook onto other nodes in a random fashion. There was a flaw in the code as the source node would also be the end node causing it to hook onto itself causing the shortest path to the destination to become just looping to itself. For example the node A would hook onto itself and when we used Dijkstra's algorithm then the shortest path would be just A.

We believe that the approach we took in order to create a random key generator was a fairly unique solution. We have never heard nor seen this type of method being implemented to generate a random key. In every successful run that we did our program would generate a unique key that varied in size and used a combination of capital letters, lower case letters, and numbers. We compared Dijkstra algorithm run time with Bellman-Ford’s algorithm runtime and

that can be seen in Figure 1. Within the twenty runs of both Dijkstra and Bellman-Ford, Dijkstra would always produce a lower runtime and Bellman-Ford would always have a constant runtime. Based off the results, we believe that our program was successful in creating a random key generator that is fairly secure.

As evidenced by Figure 2, there was not necessarily a correlation between cost and runtime. I predicted that the smaller the cost the shorter the runtime. However, I was incorrect. I assume that when the program is run that it has to go through the entire graph and the runtime is not affected by the length of the key nor the cost.

The runtime for Dijkstra is O(E+VlgV) and the runtime for Bellman-Ford is O(VE). Now, we have these runtimes to refer to. However, it is hard to relate the runtimes we recorded with the Big-O notation runtimes. But we can see that the Dijkstra algorithm did run faster each time compared to Bellman-Ford (Figure 1).

Details regarding future improvements can be found in the Conclusion.

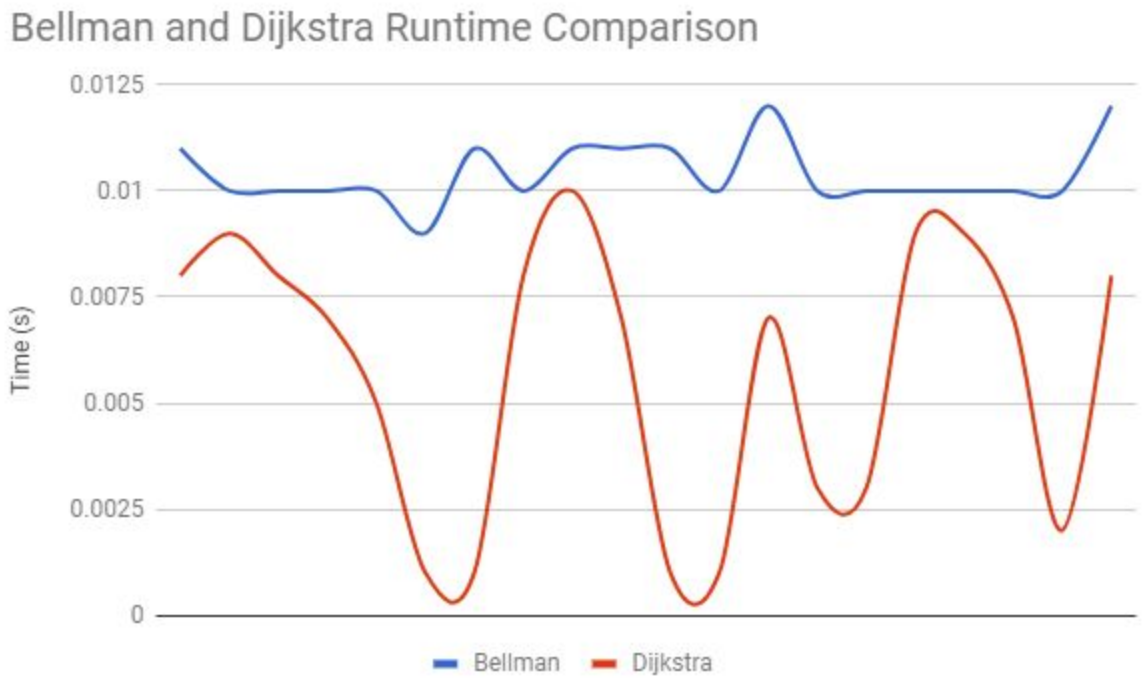

Figure 1

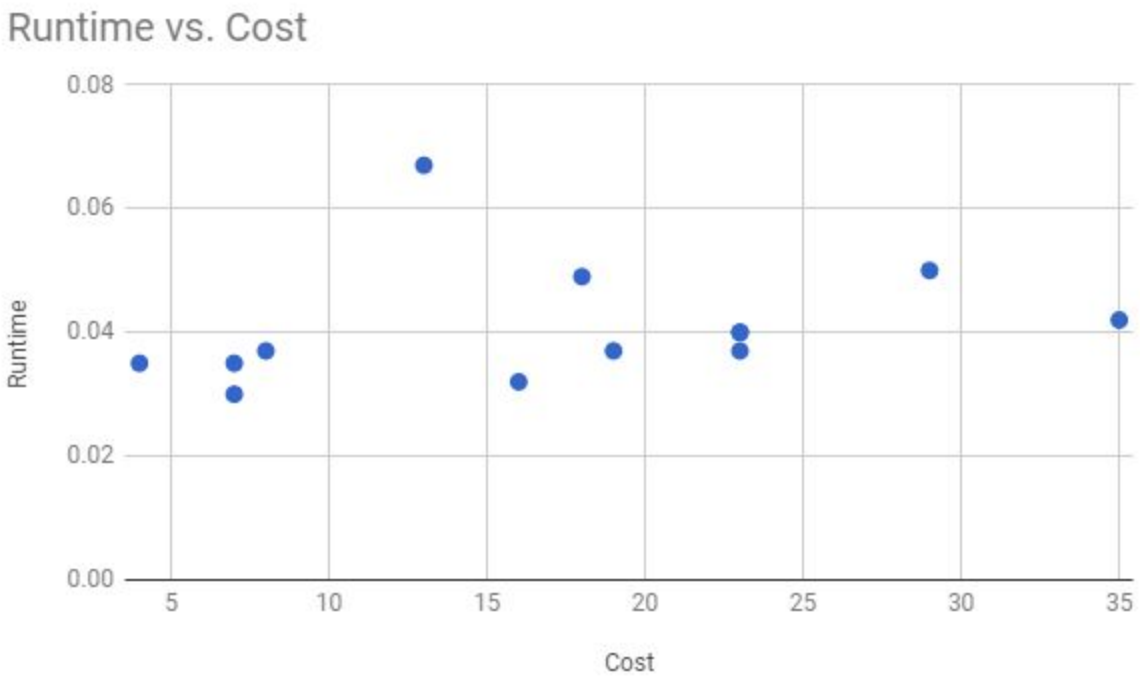

Figure 2

Conclusions and Lessons Learned

In conclusion we created a way to create a random key that can be used for security purposes. This was done in python and we used python built in random number/letter generator in order to make graphs that were random. When creating the code we ran into the problem of there being a chance where there was a chance that a node would connect to itself causing an error and creating a key with one letter. In the future if we had more time we could have implemented a minimum for the amount of characters that our key have contained. By doing this if the node looped back to itself then the source node would change to another node that did not loop to itself.

Another way we could improve the code is by changing the way we randomize nodes and their connection. Currently we use the built in random library in python, but that can easily be exploited so rather than using that library we could find a better and more secure way to randomize the nodes and their connection. Along with this, the graph itself takes up considerable size in memory, a little over 6KB, this is something that would need to be improved upon in a future revision, especially in a database applications where every bit is valuable.

References

https://github.com/Mizipzor/roguelike-dungeon-generator

- This was the inspiration for creating a graph of characters and finding the shortest path between them.

Introduction to Algorithm, 3rd Edition, by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, ISBN-10(0262033844), ISBN-13(978-0262033848) MIT Press, 2009.

- Pseudocode and information about Dijkstra and Bellman-Ford.

https://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/

- Some class definitions and visuals to describe the Dijkstra algorithm.

Contributions

Matt Spadaro: Research and Development on Algorithm

Luke Dillon: Lead Coder

Nathan Luc: Debugging and Optimization

Equal contributions were made by each team member.