Final project: delivery company Y

-Yixi Rao u6826541

1. Vision statement

X city has a delivery company Y, almost all the manufacturers in the X city collaborate with this company, therefore this company can deliver different kinds of products (e.g. furniture, electronic products, books…). The Y company also make a promise to the customers that they will try their best to deliver the parcel to the customer within one day if the customer submits the order before 23:00 Pm. The workflow of this company is:

1. first the customer submits an order. then the system will sort all orders that are submitted before 23:00 pm in an order of the soonest deadline. The order submitted after 23:00 pm will be given to next day to process.
2. Second, the location of that item's warehouse will be given to the Courier.
3. At last this system will calculate the shortest path between the location of the warehouse and the location of the customer residence, and then the Courier can deliver the parcel.

1.1 Description

X city is a square grid map, each vertex either represents a customer location or a warehouse location, and each vertex has 2,3 or 4 roads (with different distance) to other vertices. This program will help the Y company to collect all the orders and can extract the soonest deadline order in an instantly time. Also, it will help the Y company to build a memorandum about the warehouse information and the serial number of the product, and if the customer location and the warehouse location is known, then the shortest path will be calculated.

1.2 Assumption

The assumption will be made about the map, product, order.

1. The map size should not bigger than 2000 and should not less than 10. So, the maximum number of vertices is 4,000,000.
2. The edge (road) distance range should between 1 to 20km, each vertex must have 2, 3 or 4 roads, the vertex has 2 roads if and only if it is in the corner, the vertex will have 3 roads if and only if it is in the side.
3. The manufacturer name is same as the product name that it is manufactured in this factory and each manufacturer only can produce one product, each product will have an unique serial number, therefore each manufacturer have only one and unique serial number.
4. Each manufacturer only can have one warehouse and, the warehouse location cannot repeat or overlap with others warehouse location. One specific location only can have one warehouse located in it.
5. The maximum manufacturer number is 3000, the minimum number is 10.
6. The maximum number of orders can be handled in one day is 86,400, this is because one day only have 86,400 seconds, so this program maximum precision about the time

is second. And the timestamp is the (864,00 – the second have passed after the starting ordering time so the soonest deadline order will have relative larger timestamp.

2. Functionality

The overall functionality is divided into two part, the first part is initialization of all the manufacturers and the order, map as well. The second part is handling the orders, at this stage, three functionality will work together to deal with one order as the first functionality will extract an order, the second one will find the warehouse location, the third one will calculate the shortest path.

2.1 First Functionality

The data structure it used is Max-Heap

2.1.1 description

In the initialization stage of the whole program, the user will type in all information (time, name, location…) of each order to the system, and the system will store it in a list, and then add it to the Max-Heap using the timestamp to decide who should located at the top of the array. At the handling order stage, the functionality_1(OrderHeap OH) will return the top element of the array (the soonest deadline order).

2.1.2 Argument

In this functionality, I use the Max-Heap to implement it, as we just want the soonest deadline order, and the timestamp can be a comparable numeric type, so we can always get the order that has the largest timestamp (the larger timestamp means that the duration of this order is longer). Also, when the Maxheap extracts the “max” order, it will delete that order. This property fits the Application and assumption very well because we are dealing with lots of orders, and we do not want to keep the order when we finish the delivery, therefore the heap can help us not only extract the “Max” order quickly, but also help us to delete that one automatically. After deleting it, it also can adjust the structure automatically and a new top element will be elected automatically.

2.1.3 Theoretical time complexity analysis

1. T(n) of Max-heapify

Assumed that the time complexity of finding the maximum element in A[i], A[Left(i)], A[right(i)] is $\theta(1)$

and since a heap is a complete binary Tree, the largest imbalance, in which one one sub-problem is maximised happens when the last level is half full. When this happens, We have:

$n = 1 + \text{SizeLeftSubTree} + \text{SizeRightSubtree}$

$\text{SizeLeftSubTree} = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$

$\text{SizeRightSubTree} = \sum_{i=0}^{h-1} 2^i = 2^h - 1$ (h = height of the Tree)

$\because n = 1 + \text{SizeLeftSubTree} + \text{SizeRightSubtree}$

$= 1 + 2^{h+1} - 1 + 2^h - 1$

$= 3 \cdot 2^h - 1$

$\therefore 2^h = \frac{n+1}{3}$

so $\text{SizeLeftSubTree} = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1 = 2\frac{n+1}{3} - 1 \leq \frac{2}{3}n$

And $T(n) \leq T(\frac{2}{3}n) + \theta(1)$, According to master theorem, $T(n) = O(\log n)$

In the worst case

Assumed that A[1] is the smallest element in the Tree, and using the Max-heapify(1) will let the A[1] be swapped through each level of the heap until it is a leaf node. Since the heap has height $\lfloor \log n \rfloor$, so the worst time will have $\Omega(\log n)$

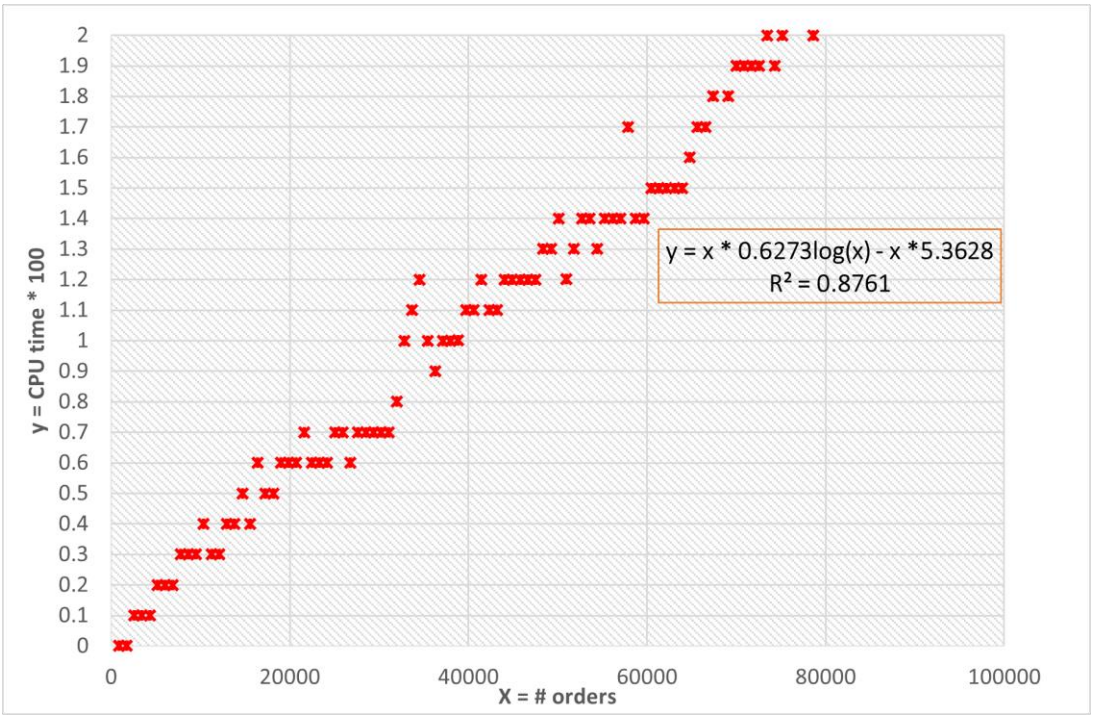
2. T(n) of Max_heap_Insert

Since We want to insert the order dynamically, so We have to insert into the heap one by one by moving the new element to the rightmost of the Tree, and then find a suitable position of the node. In the Worst time, this will take $O(n \log n)$, and also for the average case.

3. T(n) of Heap_Extract_Max()

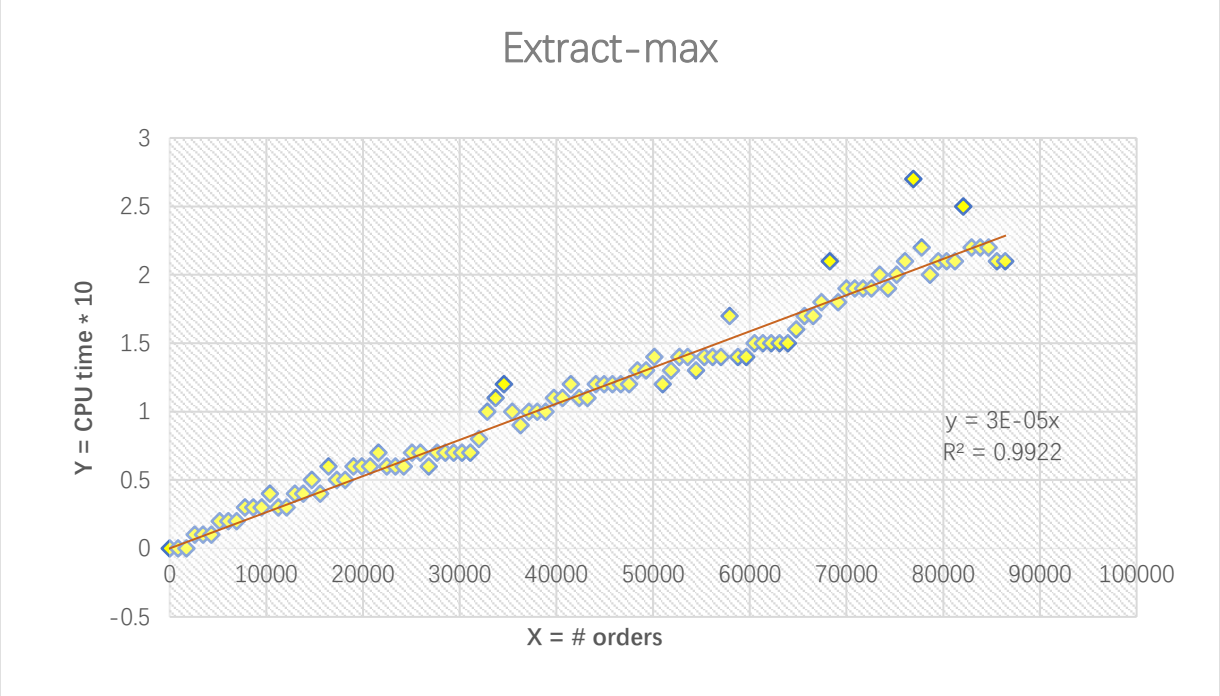
Actually this is the core function of the first functionality, Assumed that taking the first element in the heap uses $\theta(1)$. This function will extract the first one and move the last one to the top of heap, and call Max-Heapify(1). The time complexity of max-heapify is already discussed above therefore the T(n) is $\theta(1) + O(\log n) = O(\log n)$, and in the worst time, it is $\Omega(\log n)$

2.1.4 Empirical time complexity analysis



This is the graph of build a heap, and its time complexity should be O(n*log(n) and the data set

it uses to test is 864 orders to 86400 orders, the total test cases is 100. Empirical time complexity is more like y = x * 0.6273log(x) - x *5.3628 and it is belonging to the O(n*log(n)), the R square is 0.87 very close to 1, so the time complexity of building a heap is corresponded to the Theoretical time complexity analysis.



However, the Extract-Max function have some deviation to the Theoretical time complexity analysis, it fits O(n) very well rather than O(log(n)) in the Theoretical time complexity analysis. I think the reason may is spend some time to build the Order since it will call constructor method of the Order.

2.2 Second Functionality

The data structure it used is Hash Table. And it is a perfect hash table.

2.2.1 description

In the initialization stage, all the warehouse objects will be added into a warehouse hash table, the user will need to type the name of the manufacturer and the warehouse location, when the user type in a name the system will allocate a serial number which will only correspond to this warehouse and the product, the serial number will be the key to search the hash table. The value of the hash table it stored is the warehouse class object which contains the location, name, serial number attribute. At the handling stage, the system will use the order that is extracted from the heap by using functionality_1 to read the serial number the product, and use that number to search the hash table to get the warehouse object and then get the location.

2.2.2 Argument

For the HashMap, I use the perfect hashing, in the first and second level of this HashMap are both universal hashing. The reason why I choose HashMap to store all the key-value pair is that the serial number can only have one correspond product, so it would be best to use a key-value pair map which will provide a very efficient searching. The reason why I chose the universal hash function is that it will distribute all elements to the slot uniformly, and the probability of collision is less than 1/m (see written proof 1). The reason why I choose the perfect hashing is that it will have a constant searching time even in the worst case, not like the link list method that will have O(n) time complexity in the worst case, another reason why I choose the perfect hashing is that the manufacturer list will not be updated after we finish the initialization of it, so it is a static list, this fits the requirement of the perfect hashing.

1. Why it is universal

In the functionality-2, the H hash function set is $H\{(a_i \cdot key + b_i) \bmod p \bmod m\}$ where $a_i \in [1, p]$ $b_i \in [0, p]$, $p > m$

So now assume we have k, l and a random hash $h_{ab} \in H$, so we have

$r = (ak+b) \bmod p \Rightarrow r - s \equiv a(k-l) \bmod p$

$s = (al+b) \bmod p$

$\because k \neq l$ and $k-l \neq 0$, $a \neq 0$

$\therefore r \neq s$

Which tells us that $r \neq s$ at the modular p level, but they still can be equivalent at modular m level, namely $r \equiv s \bmod m$. for a given s, the maximum number of r that $s \neq r$ but $s \equiv r \bmod m$ is $\lceil \frac{p}{m} \rceil - 1$

$\lceil \frac{p}{m} \rceil - 1 \leq (\frac{p+m-1}{m}) - 1 = \frac{p-1}{m}$

so the PCs collides with r) $= \frac{p-1}{m} \cdot p - 1 = \frac{1}{m}$

so $P(h_{ab}(k) = h_{ab}(l)) \leq \frac{1}{m}$

2. Why there is no collision

An important property of perfect hashing is the second level hash table will have n_j^2 slots (n_j is the number of elements stored in this table.) So it will have $C_{n_j}^2$ pairs of elements may have a collision, since the probability of collision of a hash function hab is less than $\frac{1}{m}$, we assume X = random variable of collision, when $m = n_j^2$

$E(x) = C_{n_j}^2 \cdot \frac{1}{n_j^2} = \frac{n_j(n_j-1)}{2} \frac{1}{n_j^2} < \frac{1}{2}$

(written proof 1)

2.2.3 Theoretical time complexity analysis

1. T(n) of building a hash map

First assumed that using the hash function $((ak+b) \bmod p) \bmod m$ takes $\theta(1)$ time.

To build a Warehouse hash map, it will contain three steps, the first step is counting How many elements collide in the first universal hash function, This should be less than $P-m$, the second step is resizing the second level hash table, it should not change to n_j^2 (n_j is the number of collision in slot j), The last step is building all elements to the hash table, using the first hash function to decide which slot in the first table it should be sitted, and the second hash function to decide the second one.

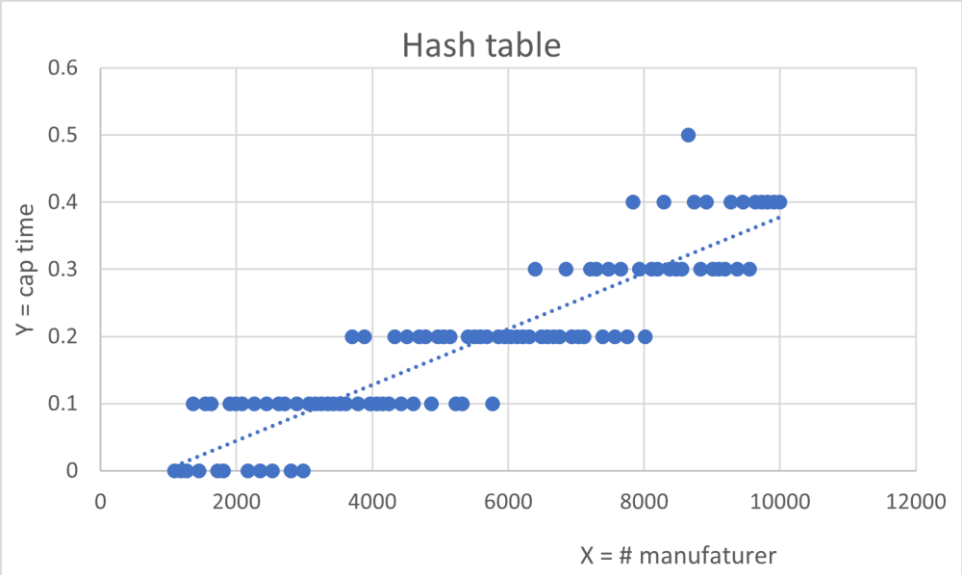
The $T(n) = n \cdot \theta(1) + n\theta(1) + n\theta(1) = 3n\theta(1) = \theta(n)$

2. T(n) of finding an element

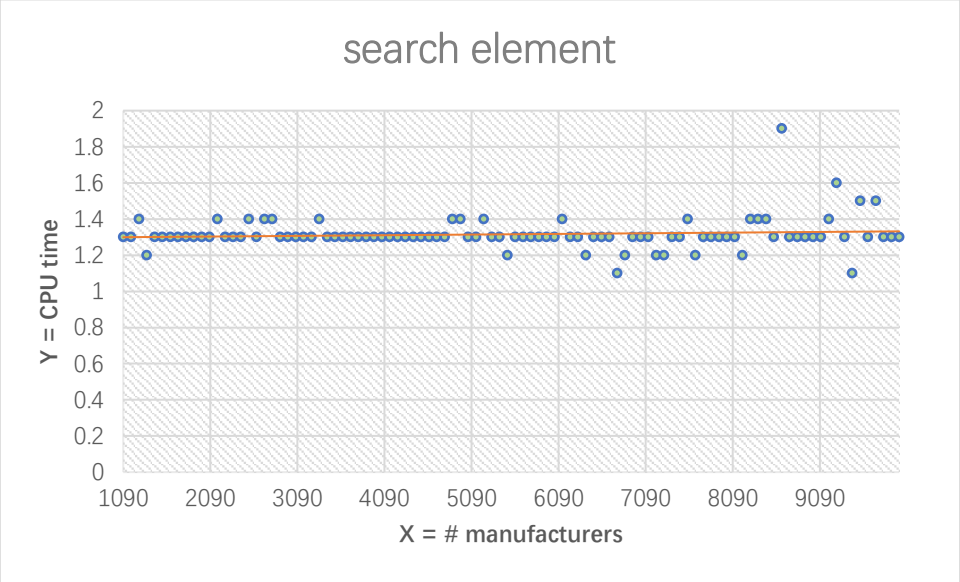
The core function in this step is search_warehouse(int key), it just return HashTable[Hash_i(key)][Hash_j(key)]

As the assumption stated above, the hash function hash_i, hash_j will take $\theta(1)$ time. So, no matter it is a worst situation or a average situation, the search time are always $\theta(1)$, Not like using the linking list, will have a bad time complexity at worst case.

2.2.4 Empirical time complexity analysis



The Theoretical time complexity analysis of creating the hash table is the O(n), and we can check this graph that this is indeed linear time in some degree, as there are multiple test cases that they have the same CPU time, this is because the time precision is concise enough therefore the CPU time will be round to a specific precision. However, we can observe that at different CPU time value level, the number of X is similar.



Note that for each test case the search function will run 200000 time therefore the Y (CPU time is actually recording the searching 200000 time of an element), this is because the time of searching one time is so small that it cannot be recorded. The Theoretical time complexity analysis of searching function should be O (1) and in this graph there is indeed a trend line Y = 1.3 fits the Theoretical time complexity.

2.3 Third Functionality

The algorithm it used is the Dijkstra algorithm of finding the shortest path

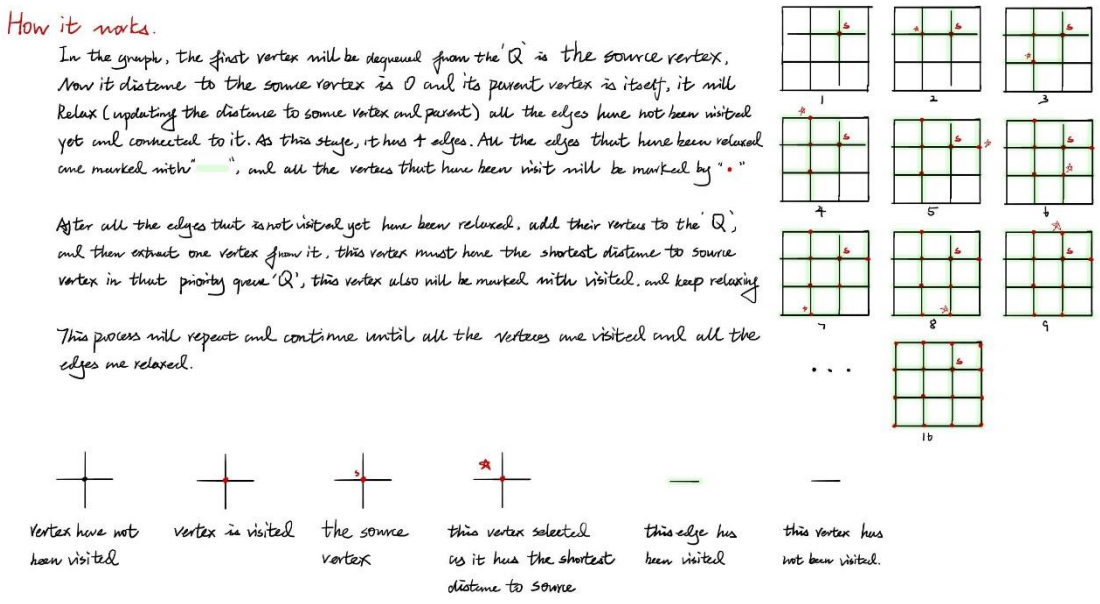
2.3.1 description

In the initialization stage, in the program, a static globe two-dimension vector variable "Dijkstra_Maps" will be initialized, every cell in this vector will represent a distance-vector table and the source vertex is that cell. This functionality will only work in the handling stage, as the first and second functionality provides the customer location (in the order class), and the warehouse location, the third functionality will first check whether there is already a distance-vector table in the "Dijkstra_Maps" of the warehouse location, if it is true, it will use it directly without running the Dijkstra again, otherwise it will run the algorithm to fill in that cell of the "Dijkstra_Maps". At last the shortest path is found by first check the destination vertex's parent vertex and use that vertex to get its parent vertex until it reaches the source vertex. The shortest path will be shown in the terminal as "S <- (x,y) …(xn,yn) <- D". (S = source, D = destination)

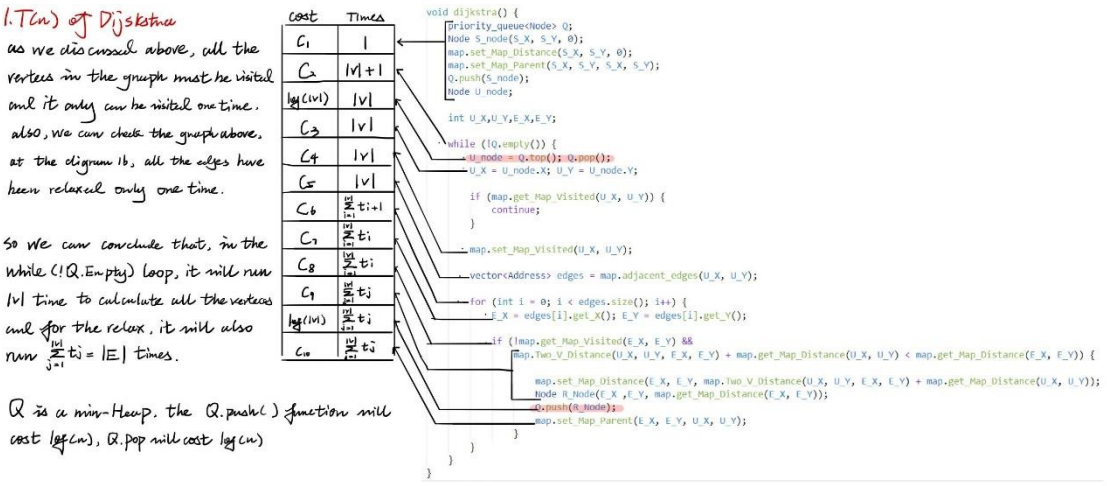
2.3.2 Argument

In this map, each road in the city, will have different distance, so we cannot naively think that a straight line path is the shortest path between two location because sometime the total weight of the curve line will have shorter distance than straight line. To find the shortest path, we must find some efficient algorithm to do it, clearly the recursively find the path for each vertex is

impractical as the time complexity is huge. The Dijkstra algorithm is suitable for many reasons, the first reason is it is very efficient and its time complexity can be O ((|V| + |E|) * log(|V|)), and the time complexity of searching the path can be O (n) where n is the map length, the second reason is that one of the Dijkstra algorithm requirement is all the edges' weight must be positive, and my program can satisfy this because in the reality, a road with negative distance cannot exist. The last reason is that we can store all the distance-vector maps where its source locations are all the warehouse location in some place, so the next time we want to find the path that has the same source location, we can just retrieve it in the "Dijkstra_Maps" and no need to run the algorithm again.



2.3.3 Theoretical time complexity analysis



$$\text{So } T(n) = C_1 + C_2(|V|+1) + \log(|V|)\cdot|V| + (C_3+C_4+C_5)|V| + C_6\left(\sum_{i=1}^{|V|} t_i + 1\right) + (C_7+C_8)\sum_{i=1}^{|V|} t_i + (C_9+C_{10})\sum_{j=1}^{|V|} t_j + \log(|V|)\sum_{j=1}^{|V|} t_j$$

$$\because |V| = n^2,\ |E| = 2n(n+1),\ \sum_{j=1}^{|V|} t_j = |E|,\ \sum_{i=1}^{|V|} t_i = 4n^2 - 4n,\ n \text{ is the number of vertex in each row or column}$$

$$\therefore T(n) = \log(|V|)(|E| + |V|) + C'|V| + C''|E| + C'''$$

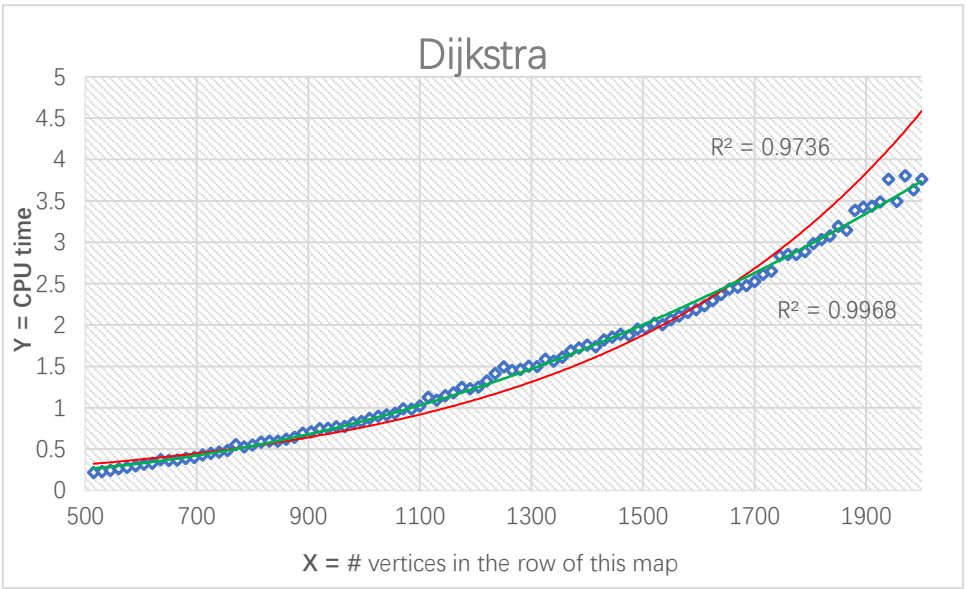
$$\therefore T(n) = O(\log(|V|)\cdot(|E|+|V|)) = O(\log(n^2)(2n^2-2n+n^2)) = O(2\log(n)(3n^2-2n)) = O(\log n(n^2))$$

2. T(n) of finding the shortest path

When the Dijkstra algorithm is finished, we can now find the shortest path between two distance, namely the source vertex and the destination vertex. This is done by getting the parent vertex of the destination vertex, and using the destination parent vertex to get its parent vetex, this vertex will repeat until the parent vertex is the source vertex. Obviously, the time complexity of this function is dependent on the number of intermediate vertexes between the destination and source.

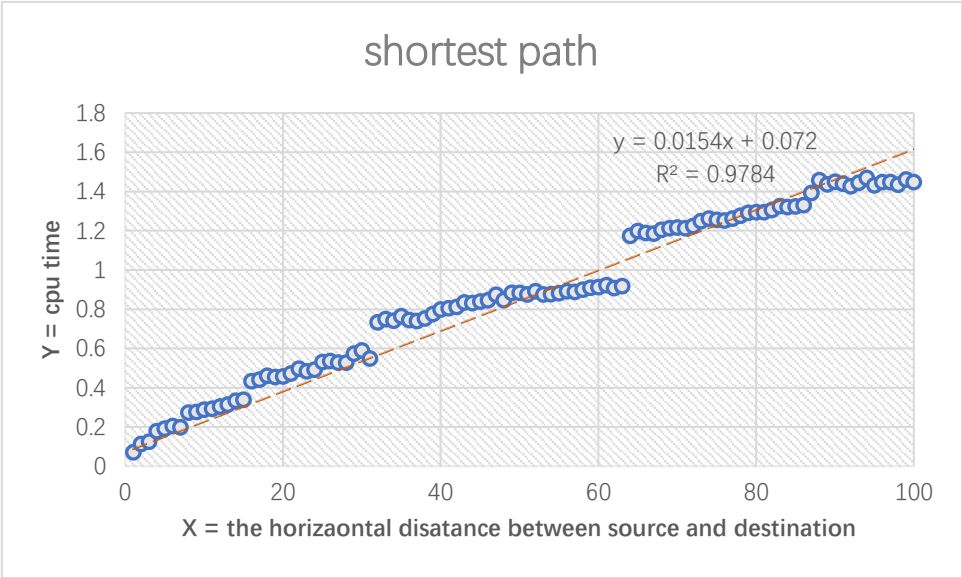
So in the worst case, it may run through all the vertexes in the graphs, $T(n) = \theta(|V|)$. and in the average case it should be $\theta(|V|) = O(n^2)$

2.3.4 Empirical time complexity analysis



Theoretical time complexity analysis of the Dijkstra () is O (log(n) * n^2), and the green line in this graph is the asymptotic line of t = n^2, and the orange line is the t = n^2 * log(n), it is obvious that the R^2 is very close to 1, namely 0.97. So, we can say that the time complexity

of Dijkstra algorithm is Ω (n^2) and O (n^2 * log(n)), so the Theoretical time complexity analysis is correct. (NOTE: n is the number of vertices in a row not the total number of vertices in the graph, the |V| = n^2.



Theoretical time complexity analysis of the find_path () function is O(|V|), the time is dependent to the source vertex and destination vertex location, and in the empirical analyse, the source vertex is the origin point (0,0), and the destination is vertex 1 to vertex 100. As the graph show it is a linear line, fits the Theoretical time complexity analysis very well. The reason why there are some segments is that as the destination vertices is 1 to 100, some periodic vertices will have the different row, as this is a square grid map, and this will increase the number of vertices between source vertex and the destination vertex.