SCSA SC2001 Lab Example Class Project 2 Team 7

PU FANYI (U2220175K)

PUSHPARAJAN ROSHINI (U2222546A)

QIAN JIANHENG OSCAR (U2220109K)

RHEA SUSAN GEORGE (U2220116B)

Part A:

1) Using array for priority queue
2) Using Adjacency Matrix

Implementation of Dijkstra’s algorithm

```
distances[0] = 0;
for (int i = 1; i < numVertices; ++i) {
    int min = INT_MAX;
    int min_index = -1;
    for (int j = 0; j < numVertices; ++j) {
        ++(*compare_count);
        if (!visited[j] && distances[j] < min) {
            min = distances[j];
            min_index = j;
        }
    }
    if (min == INT_MAX) break;
    visited[min_index] = 1;
    for (int j = 0; j < numVertices; ++j) {
        if (!visited[j] && (++(*compare_count)) &&
            adjMatrix->adjMatrix[min_index][j] != INT_MAX &&
            (distances[j] == INT_MAX ||
             distances[min_index] + adjMatrix->adjMatrix[min_index][j] < distances[j])) {
            distances[j] = distances[min_index] + adjMatrix->adjMatrix[min_index][j];
        }
    }
}
```

Theoretical Analysis

The Dijkstra algorithm time complexity is O(V^2), where V is the graph's vertex count.
The following is an explanation:

1. Finding the unvisited vertex with the shortest path. This requires **O(V)** time.
2. We now need to relax the neighbours of each vertex chosen. Every relaxation time complexity is O(1).
3. We need to relax every vertex's neighbours, it takes [O(V) * O(1)] = O(V) time to update every vertex's neighbours.

Processing one vertex in O(V) seconds
O(V)*O(V) = O(V^2) is the time required to visit and process all of the vertices.

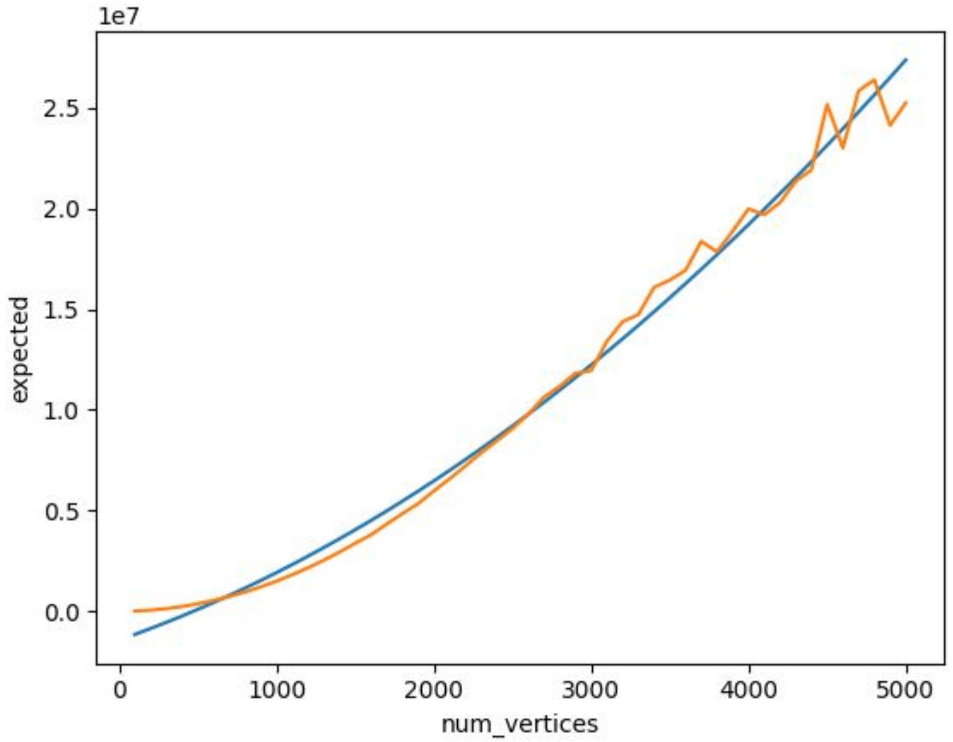
Empirical Analysis: Fixed |E|, Vary |V|

Our approach:

1. Generate random graphs with the following settings
   - Fixed |E| = 10,000 and Starting |V| = 100 , until 5,000
   - Step = 100
2. Each step, the algo is ran 100 times with a newly generated graph.
3. Average key comparison is then taken

Empirical Analysis

Fixed Edges, Vary Vertices



$T(|V|, |E|) = k_1 * |V|^2 + k_2 * |V|$

$k_1 = 6.03343824e\text{-}01$
$k_2 = 2.74286781e\text{+}03$

Empirical Analysis: Fixed |V|, Vary |E|

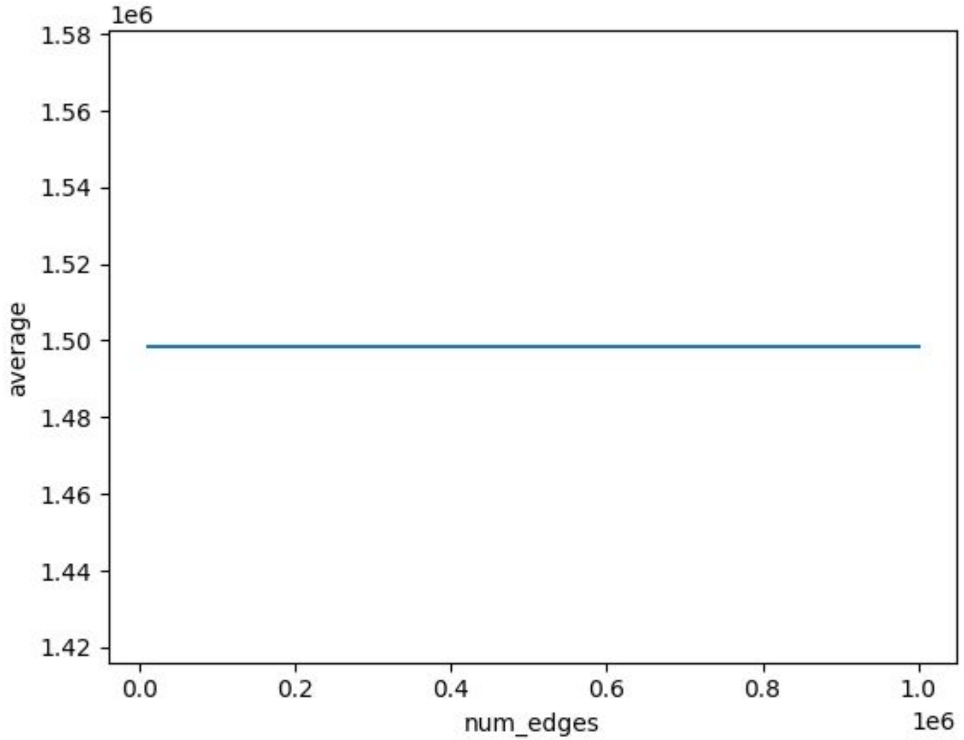
Our approach:

1. Generate random graphs with the following settings
   - Fixed |V| = 1000 and Starting |E| = 10,000 , until 1,000,000
   - Step = 10,000
2. Each step, the algo is ran 10 times with a newly generated graph.
3. Average key comparison is then taken

Empirical Analysis

$T(|V|, |E|) = O(|V|^2)$

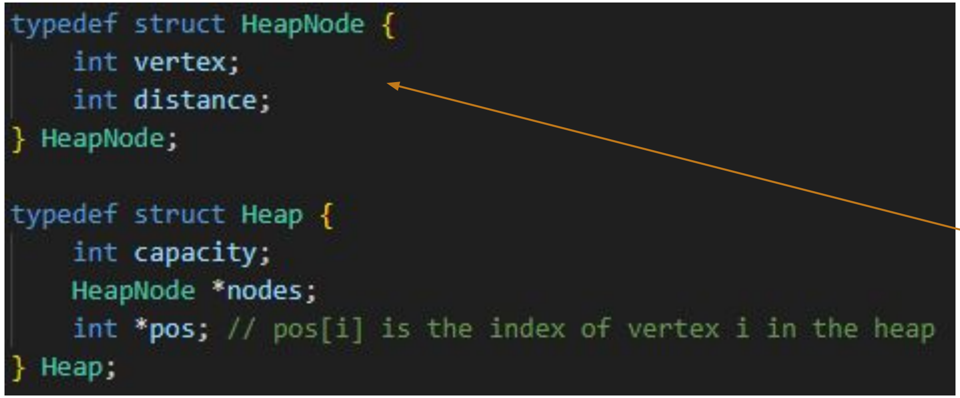
Fixed Vertices Vary Edges



Part B:

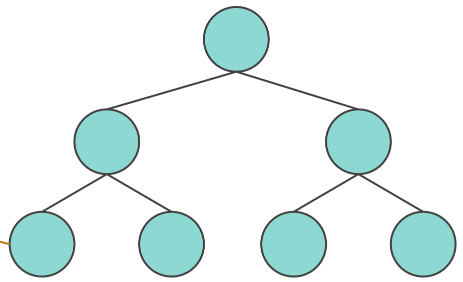
1) Using Minimising Heap for priority queue
2) Using Adjacency List

Implementation

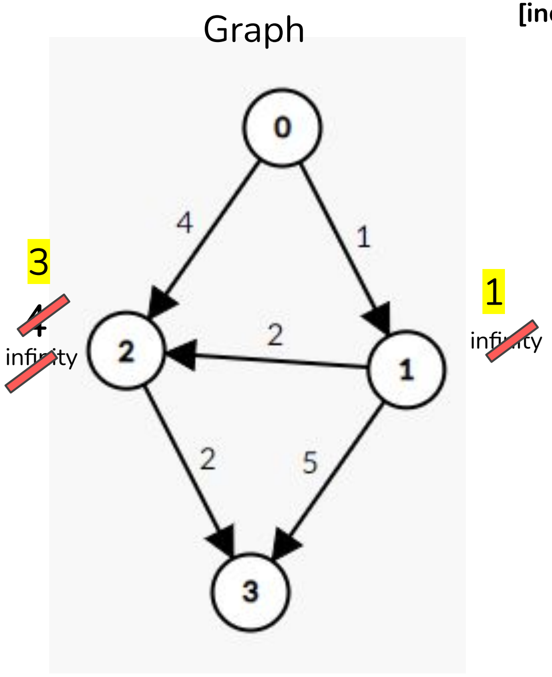


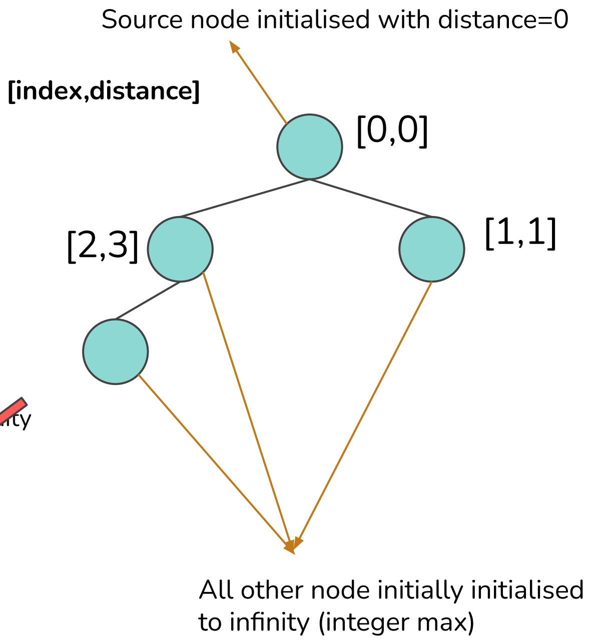
```c
typedef struct HeapNode {
    int vertex;
    int distance;
} HeapNode;

typedef struct Heap {
    int capacity;
    HeapNode *nodes;
    int *pos; // pos[i] is the index of vertex i in the heap
} Heap;
```



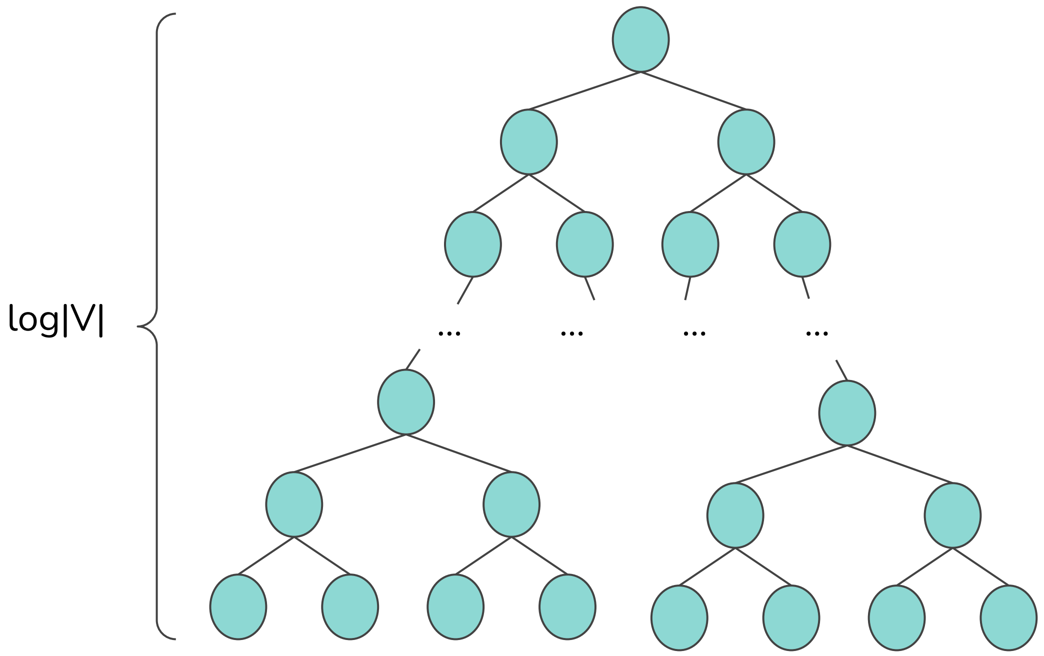
Example




Source node initialised with distance=0
[index,distance]
[0,0]
[2,3]
[1,1]
All other node initially initialised to infinity (integer max)

1. Create a minimising heap and initialise with source vertex - distance to be 0, other vertices : infinity
2. While minimum heap is not empty, pop the root vertex with minimum distance/highest priority
3. For each adjacent vertex of popped vertex, calculate the new distance by adding the weight of edge between the 2 vertices
   a. If new distance is < current, update the distance array
   b. And update the heap
4. Repeat until all vertices are popped or destination vertex attained
5. Obtain shortest distances of vertices from distance d array

Theoretical Analysis



2 Key factors which affect time complexity:

1. Popping/Extracting the minimum element from the minimising heap
   a. The operation to fix the heap after popping each vertex takes **O(log |V|)**
   b. We do this for all **V** vertices so **O(|V| log |V|)**
2. Relaxation of outgoing edges
   a. For **V** vertices, **|E|** : max number of times where distance can be adjusted
   b. **O (log |V|) :** Swapping to fixheap
   c. Combining both, **O(|E| log |V|)**

$$\boxed{O((|V| + |E|)\log |V|)}$$

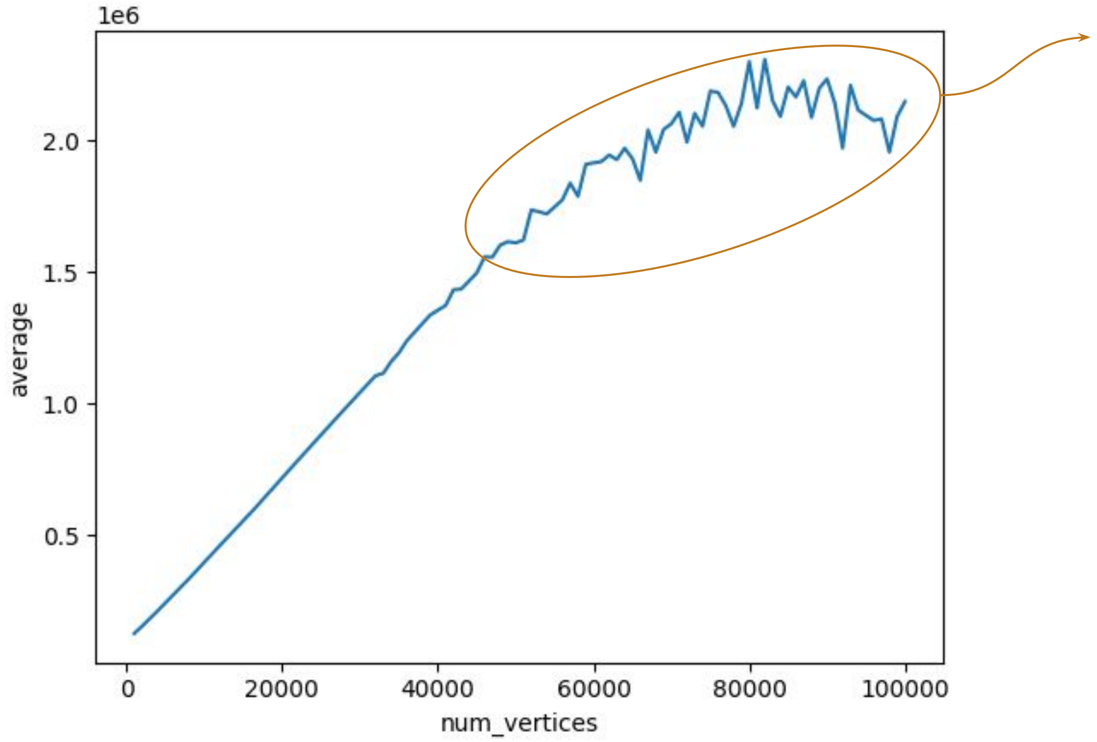
Empirical Analysis: Fixed |E|, Vary |V|

Our approach:

1. Generate random graphs with the following settings
   - Fixed |E| = 200,000 and Starting |V| = 1000 , until 100,000
   - Step = 1000
2. Each step, the algo is ran 100 times with a newly generated graph.
3. Average key comparison is then taken

Empirical Analysis

Fixed Edges, Vary Vertices

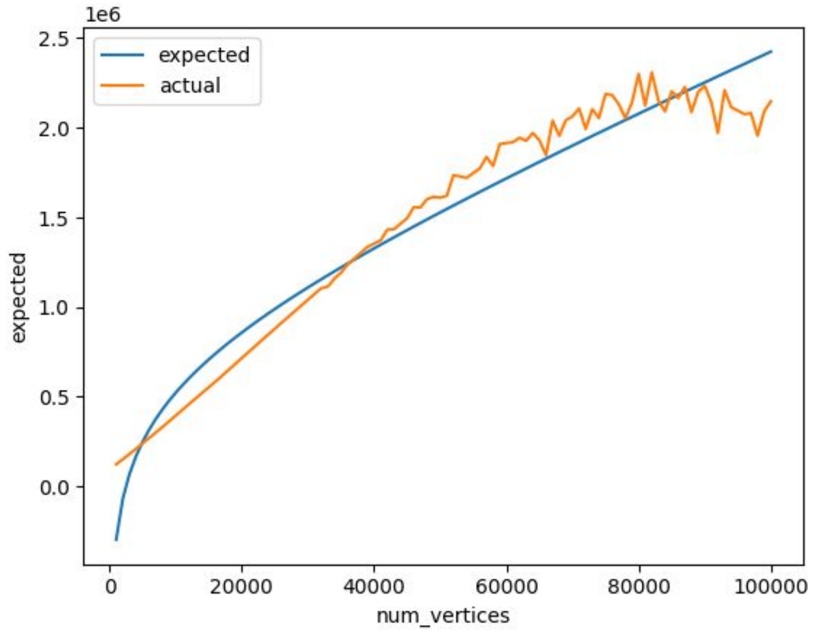


As |E| gets nearer to |V|, Since our graph are randomly generated, there are increasing chance of disconnected components appearing.

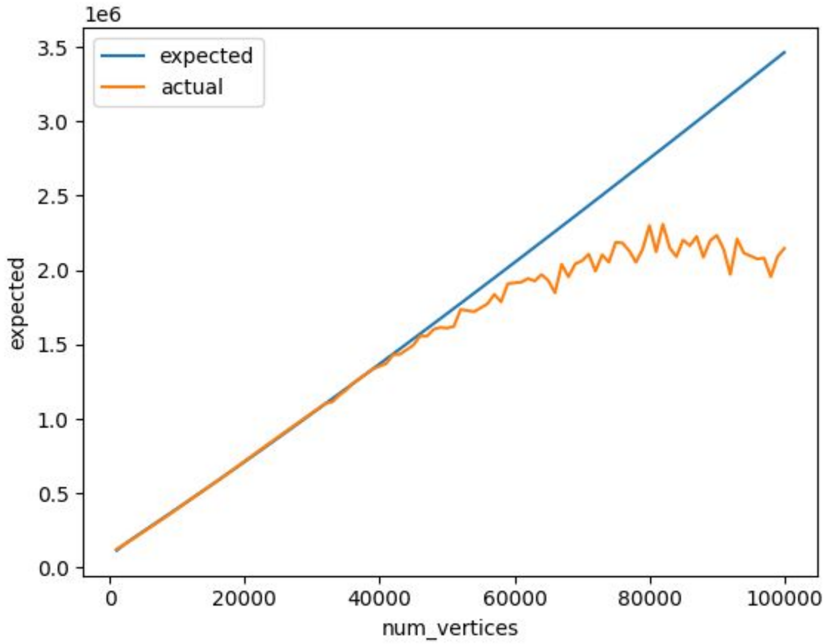
Resulting in lesser than expected key comparison in some case -> leading to fluctuations

Compare Theoretical and Empirical

$T(|V|, |E|) = k_1 * |V| \log |V| + k_2 * |E| \log |V|$



Regression on the whole range
$k_1 = 0.770432$, $k_2 = 1.08882654$



Regression on $|V| < 40000$
$k_1 = 1.98047116$, $k_2 = 0.05876973$

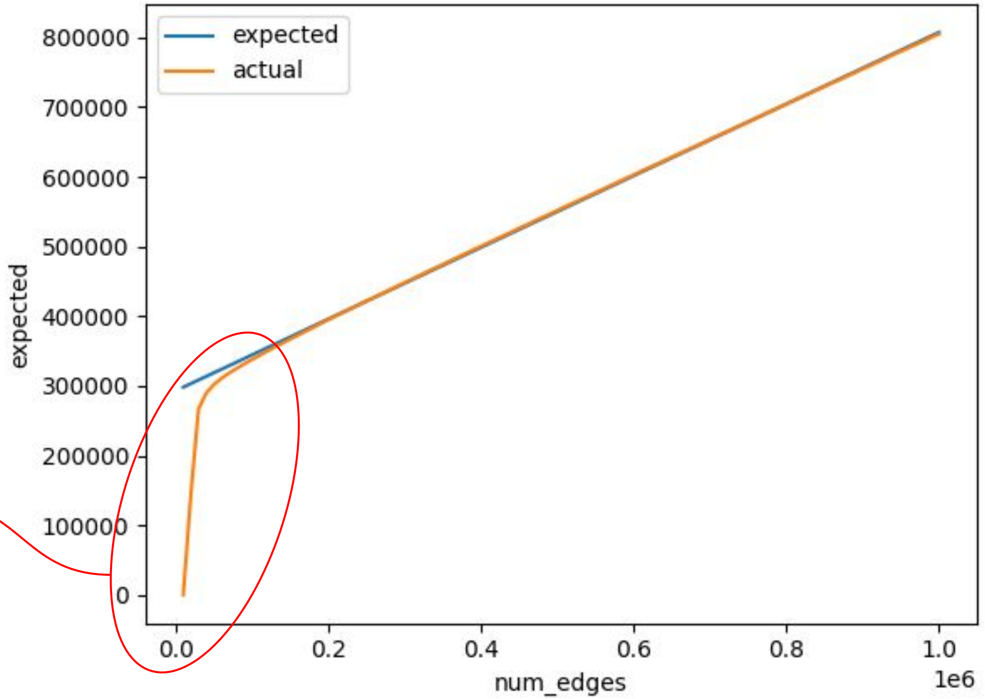
Empirical Analysis: Fixed |V|, Vary |E|

Our approach:

1. Generate random graphs with the following settings
    - Fixed |V| = 10,000 and Starting |E| = 10,000 , until 1,000,000
    - Step = 10,000
2. Each step, the algo is ran 10 times with a newly generated graph.
3. Average key comparison is then taken

Empirical Analysis Fixed Vertices Vary Edges

$T(|V|, |E|) = k * |E| \log |V| + C$ $\rightarrow$ $k * |V| \log |V|$



Same reason as before, When |V| is similar to |E|,

There is high chance for disconnected components to appear

$|V| = 1e4$
Regression on $|E| > 1e5$
$k = 3.87104105e-02$

Conclusion

Using linear regression we have proven that the time complexity is indeed **$O((|V|+|E|) \log |V|)$**

Part C: Comparison of 2 Implementation

Time complexity of part a dijkstra: **$O(|V|^2)$**

Time complexity of part b dijkstra: **$O((|V|+|E|) \log |V|)$**

When $|E|$ is small, **(like social networks, web page connectivity)**

$$(|V| + |E|) \log |V| << |V|^2$$

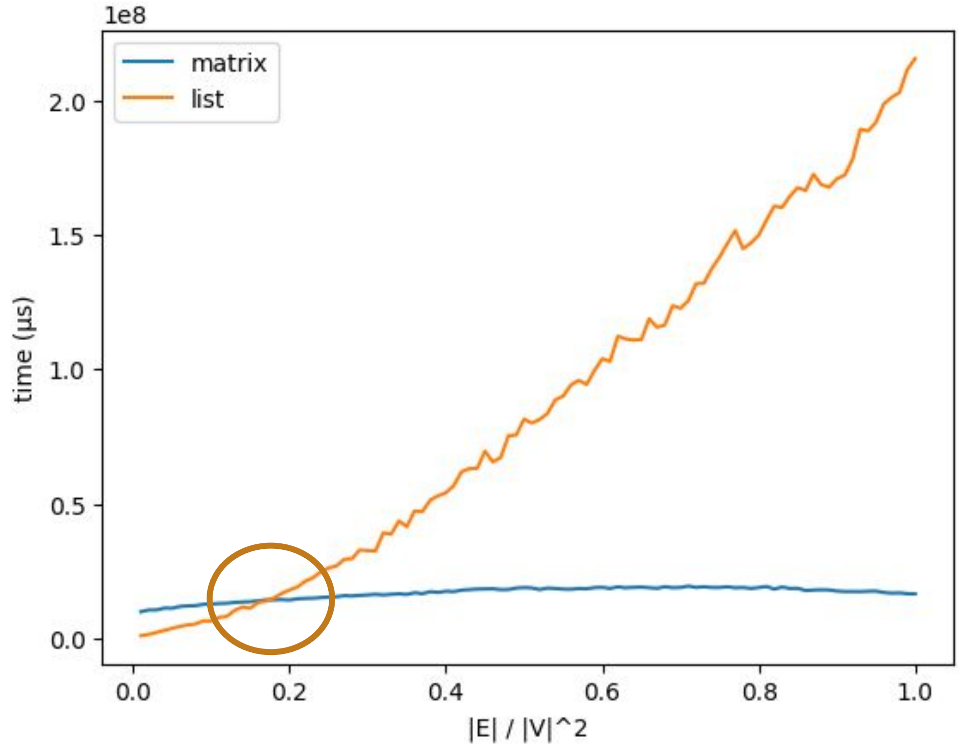
(so part b, list + min Heap is better)

When $|E|$ is large, and it can at most be $|V|^2$, **(like mesh network)**

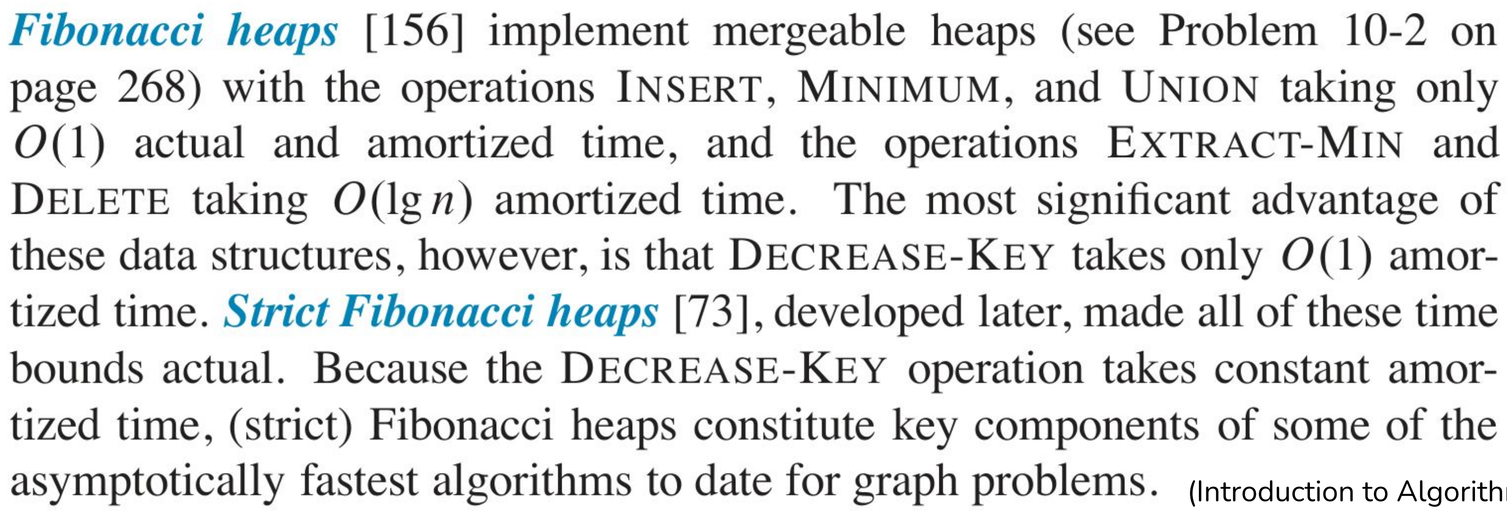
$$O((|V|+|E|) \log |V|) \approx O(|V|^2 \log |V|) > O(|V|^2)$$

(so part a, matrix + array is better)

Part C: Comparison of 2 Implementation



Fibonacci heaps



- ***Fibonacci heaps*** [156] implement mergeable heaps (see Problem 10-2 on page 268) with the operations INSERT, MINIMUM, and UNION taking only $O(1)$ actual and amortized time, and the operations EXTRACT-MIN and DELETE taking $O(\lg n)$ amortized time. The most significant advantage of these data structures, however, is that DECREASE-KEY takes only $O(1)$ amortized time. ***Strict Fibonacci heaps*** [73], developed later, made all of these time bounds actual. Because the DECREASE-KEY operation takes constant amortized time, (strict) Fibonacci heaps constitute key components of some of the asymptotically fastest algorithms to date for graph problems.

(Introduction to Algorithm, 4th)

Every update is a Decrease-Key operation, takes **O(1)** time (amortized)

So the time complexity takes **O(|E| + |V| log |V|)** time.