Group 8

Project 2: The Dijkstra’s Algorithm

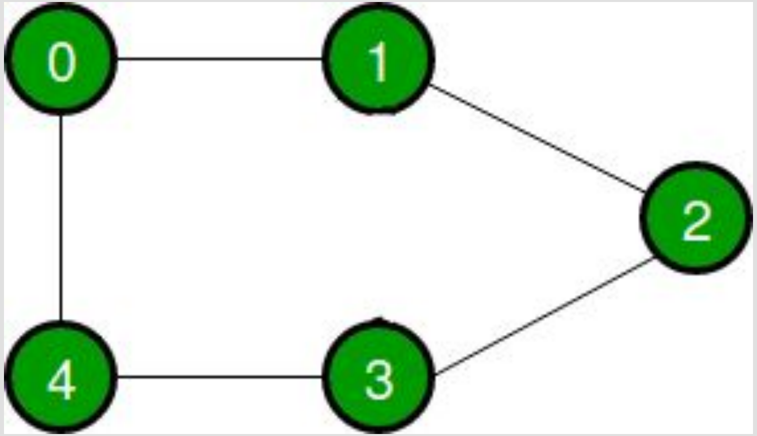
Question

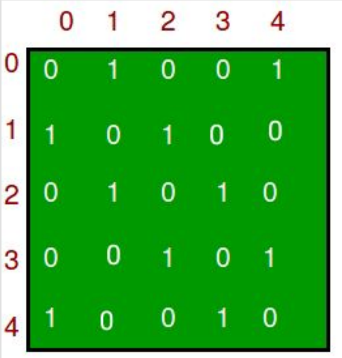
Using the Dijkstra's algorithm, we want to find out how the following affects it's time complexity:

1) Input graph stored in an adjacency matrix and using an array for the priority queue

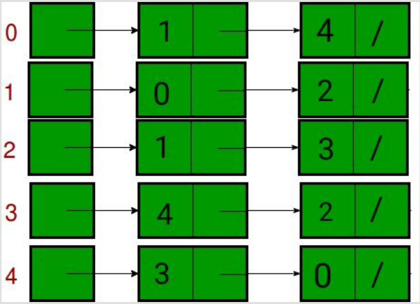
2) Input graph stored in an an array of adjacency lists and using a minimizing heap for the priority queue

Adj Matrix vs Adj List





Matrix



List

Part A: Graph stored in an adjacency matrix and we use an array for the priority queue

```
void IMPL1::dijkstra(int src)
{
    //Initialization: for each vertex, set pi to -1 and d to infinity
    fill(S.begin(),S.end(), 0);
    fill(d.begin(),d.end(), MAX);
    fill(pi.begin(),pi.end(), -1);
    //Change the source
    this->source = src;
    d[src] = 0;
    pi[src] = src;
    //Put all vertices in priority queue, Q, in d[v]'s increading order
    for (int j = 0; j < v; j++)
        Q.push_back(j);
    //extract from Q
    for (int iter = 0; iter < v - 1; iter++)
    {
        int u = findCheapest(); //the cheapist element should be in the first position of Q
        S[u] = 1; //Add u to S
        //for each vertex adjacent to u:
        for (int i = 0; i < v; i++)
        {
            if (S[i] != 1 && d[i] > adj_mtx[u][i] + d[u]) //if the vertex is unvisited and d[u]+wt[u,i] is shorter
            {//update i's parent and distance
                d[i] = adj_mtx[u][i] + d[u];
                pi[i] = u;
            }
        }
    }
}
```

```cpp
class IMPL2 {
public:
    IMPL2(int n);                   //constructor
    ~IMPL2();                       //destructor
    void dijkstra(int source);      //use the dijkstra algorithm to traverse through all the nodes, and obtain d & pi
    void printPath(int target);         //print out the path from given source to target
    void printSol();
    void makeEdge(int i, int j, int wt);    //make a edge from vertex i to vertex j with the given weight
private:
    //adjacency list
    struct AdjListNode* newAdjListNode(int dest, int weight);

    //heap
    struct MinHeapNode* newMinHeapNode(int v, int dist);
    void heapSwap(struct MinHeapNode** a, struct MinHeapNode** b);
    void fixHeap(int idx);  // A standard function to heapify at given idx(node). This function also updates position of nodes when they are swapped. Position is needed for updateDis()
    struct MinHeapNode* extractMin(); // Standard function to extract minimum node from heap
    void updateDis(int v, int dist);
    bool isInMinHeap(int v);

    //variables
    int v;                      //number of vertices
    int source;                 //source of a path
    struct AdjList* adj_list;   //adjacency list for graph
    std::vector<int> S;         //array for priority queue
    std::vector<int> d;         //distance from the source
    std::vector<int> pi;        //parent node
    struct MinHeap* minHeap;
};
```

Part B: Graph stored in an array of adjacency list and we use an minimising heap for the priority queue

```
void IMPL2::dijkstra(int src)
{
    //initialize everything
    fill(S.begin(), S.end(), 0);
    fill(d.begin(), d.end(), MAX);
    fill(pi.begin(), pi.end(), -1);

    // Initialize min heap with all vertices. dist value of all vertices
    for (int i = 0; i < v; i++)
    {
        minHeap->array[i] = newMinHeapNode(i, d[i]);
        minHeap->pos[i] = i;
    }
    // Make distance value of src vertex as 0 so that it is extracted first
    //minHeap->array[src] = newMinHeapNode(src, d[src]);
    //minHeap->pos[src] = src;
    d[src] = 0;
    pi[src] = src;
    updateDis(src, d[src]);

    // In the followin loop, min heap contains all nodes whose shortest distance is not yet finalized.
    while (minHeap->size > 0)
    {
        // Extract the vertex with minimum distance value
        struct MinHeapNode* minHeapNode = extractMin();
        // Store the extracted vertex number
        int u = minHeapNode->v;
        S[u] = 1;
        // Traverse through all adjacent vertices of u (the extracted vertex) and update their distance values
        struct AdjListNode* neighbours = adj_list->array[u].head;
        while (neighbours != NULL)
        {
            int next_v = neighbours->dst;

            // If shortest distance to v is not finalized yet, and distance to v through u is less than its previously calculated distance
            if (isInMinHeap(next_v) && S[next_v] != 1 && neighbours->wt + d[u] < d[next_v])
            {
                d[next_v] = d[u] + neighbours->wt;
                pi[next_v] = u;
                // update distance value in minHeap also
                updateDis(next_v, d[next_v]);
            }
            neighbours = neighbours->nxt;
        }
    }
}
```

Theoretical Time Complexities (Pseudocode)

```
Dijkstra_ShortestPath ( Graph G, Node source ) {
  for each vertex v {          O(n) to initialise arrays
  d[v] = infinity;
  pi[v] = null pointer;
  S[v] = 0;
   }
  d[source] = 0;
  put all vertices in priority queue, Q, in d[v]'s increasing order;
  while not Empty(Q) {         O(n) loop to termination
  u = ExtractCheapest(Q);      O(n) to find the cheapest in array
  S[u] = 1; /* Add u to S */
  for each vertex v adjacent to u
    if (S[v] ≠ 1 and d[v] > d[u] + w[u, v]) {
    remove v from Q;
    d[v] = d[u] + w[u, v];
    pi[v] = u;
    insert v into Q according to its d[v];
       }
   } // end of while loop
}
```

O(n) if priority queue is directly constructed from d. O(nlogn) if heap is created by inserting vertices 1 by 1

Adjacency Matrix (array): **O(|V²|)**

Adjacency List (minimizing heap): **O((|V|+|E|) log|V|)**

Theoretical Time Complexities

Adj Matrix + Array

Time taken to select vertex with minimum distance → $O(|V|)$

Loop through total number of vertices = $|V|$

Time taken for each iteration of loop= $O(|V|)$

Iterate (1) through all vertices → $|V|+|V| * |V|$ = $\mathbf{O(|V^2|)}$

Theoretical Time Complexities

Adj List + Heap

|V| extractions from the priority queue and |E| updates to the priority queue → O(|E| + |V|)

Time taken for each iteration of loop= O(|V|)

Finding & updating 1 adjacent vertex weight → O(log(|V|))

Iterate through all vertices & edges→ $O(|V|) + O(|E| \times log|V|) + O(|V| \times log|V|)$

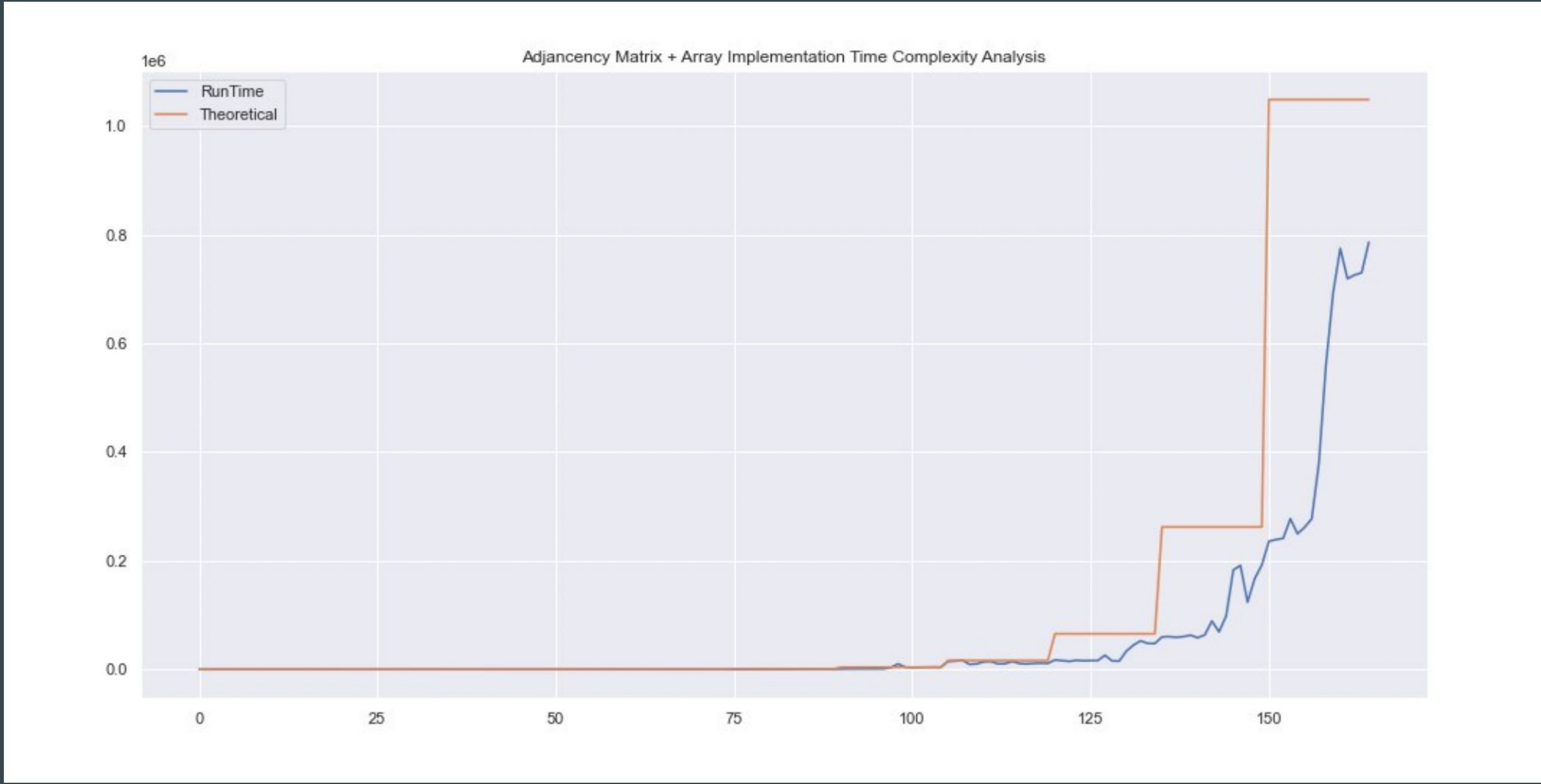
$$O((|E| + |V|) \times log|V|) = O(|E| \times log|V|)$$

- Sparse: **O(|V| log(|V|))**
- Dense: **O(|V|^2 log(|V|))**

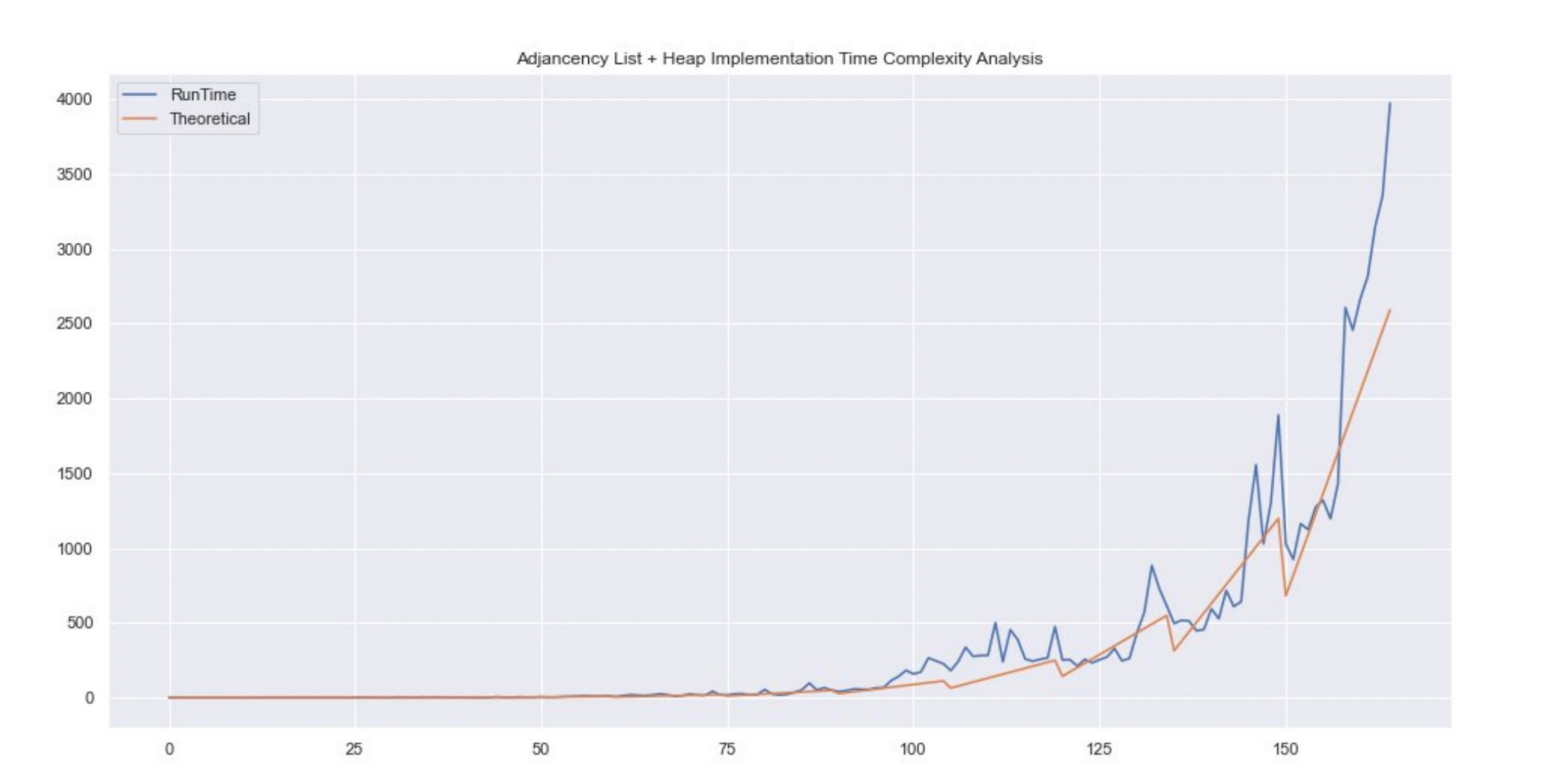
Compare implementation against theoretical analysis

	NumOfVertices	NumOfEdge	Source	Mtx+Arr runtime(ms)	List+Heap runtime(ms)	Mtx+Arr In Theory	List+Heap In Theory
0	10	5	3.0	2.8	2.6	1.0	0.166096
1	10	6	3.0	1.8	1.2	1.0	0.199316
2	10	7	3.0	1.5	0.7	1.0	0.232535
3	10	8	3.0	1.0	1.4	1.0	0.265754
4	10	9	3.0	1.6	1.1	1.0	0.298974
...	...	...	...	...	...	...	...
160	10240	15360	301.8	774852.8	2666.2	1048576.0	2046.248155
161	10240	16384	340.3	719065.1	2814.0	1048576.0	2182.664699
162	10240	17408	378.5	725792.6	3147.3	1048576.0	2319.081243
163	10240	18432	416.7	729920.6	3354.1	1048576.0	2455.497786
164	10240	19456	455.5	785988.9	3972.5	1048576.0	2591.914330

Compare implementation against Theoretical analysis for part (a)

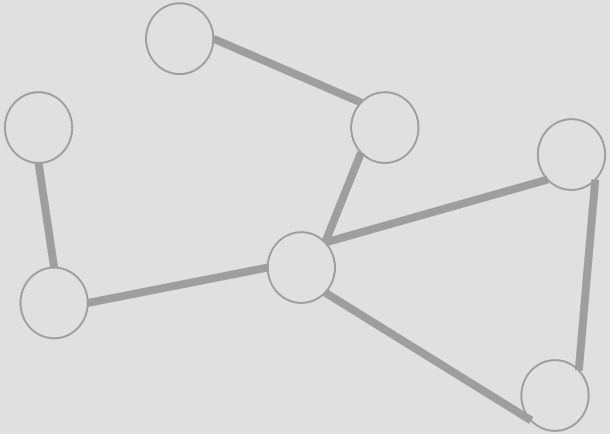


Compare implementation against Theoretical analysis for part (b)

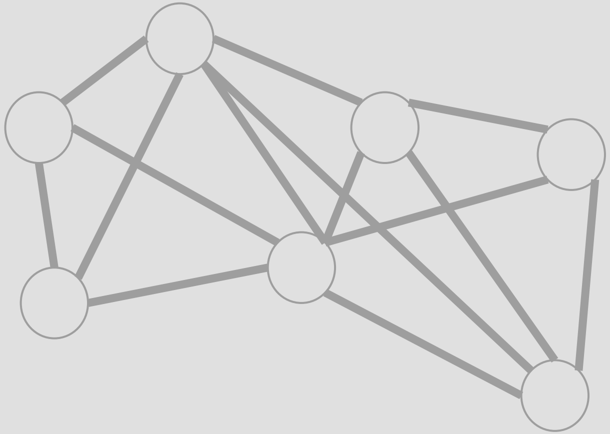


Compare implementations against $|V|$ and $|E|$

Sparse Graph vs Dense Graph

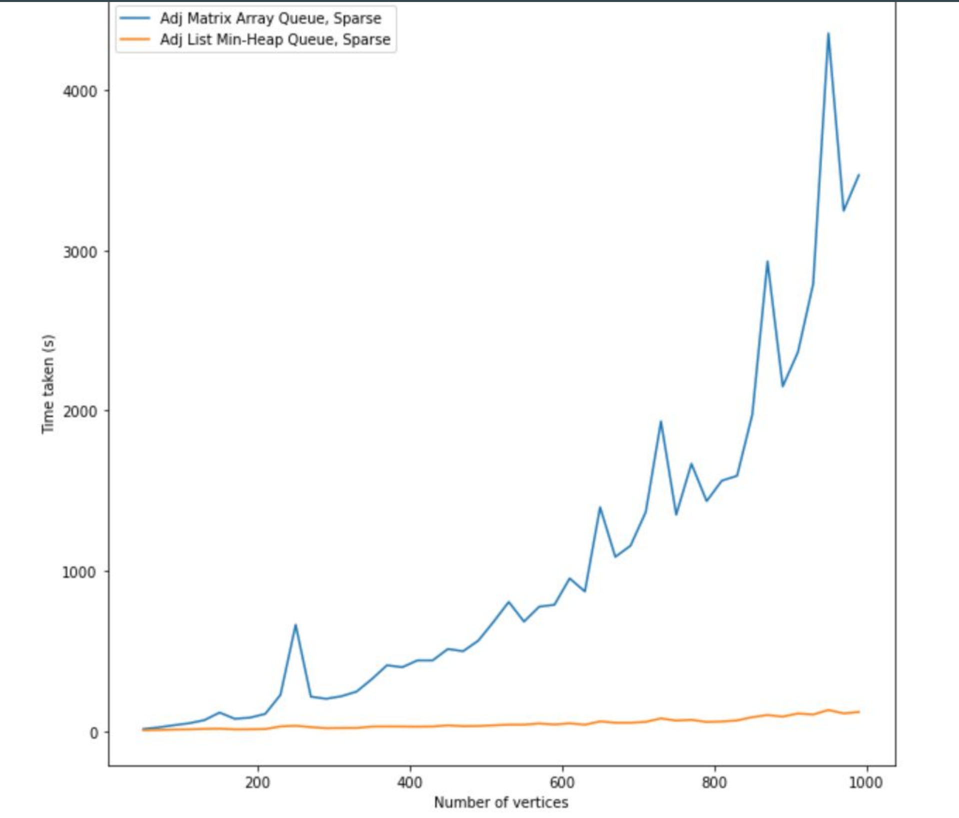


<u>Sparse</u>
Number of edges is close to the **minimum** number of edges for a given number of vertices



<u>Dense</u>
Number of edges is close to the **maximal** number of edges for a given number of vertices

Sparse Graph

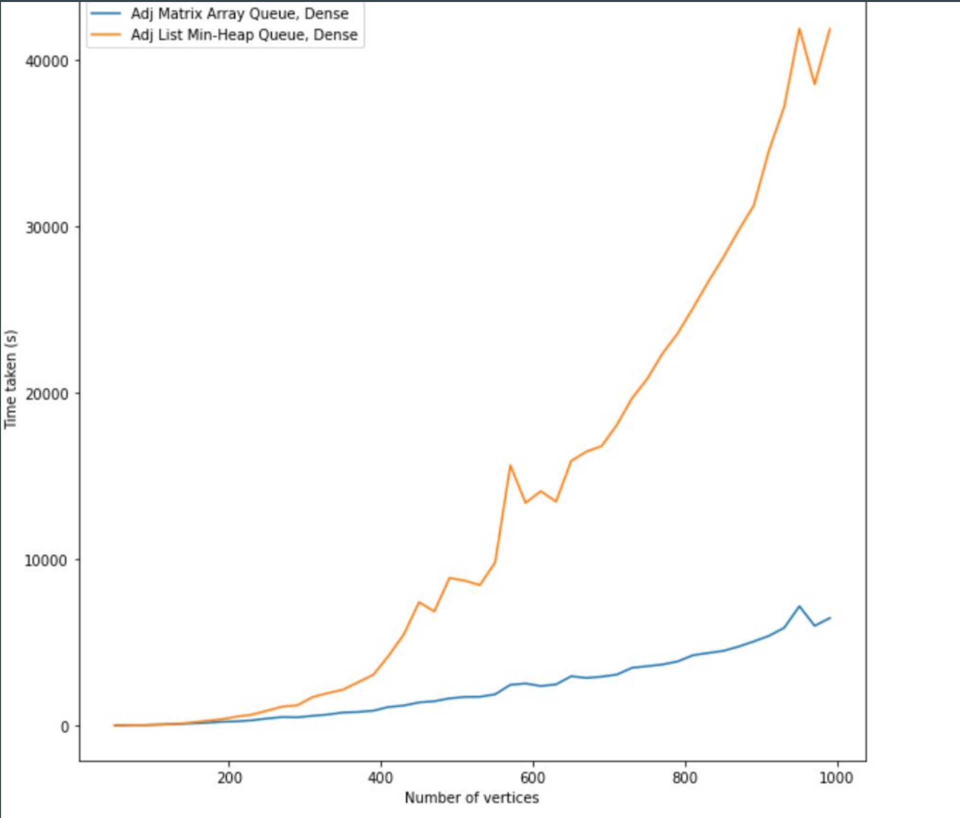


Adjacency Matrix + Array

Adjacency List + Heap ☆

$|E| = |V|$

Dense Graph



Adjacency Matrix + Array ★

Adjacency List + Heap

$$|E| = |V^2| - |V|$$

Which implementation is better?

Adjacency Matrix + Array is better for Dense Graphs

- Graphs with a relatively small number of vertices
- Graphs with a relatively large number of edges

Adjacency List + Minimizing Heap is better for Sparse Graphs

- Graphs with a relatively large number of vertices
- Graphs with a relatively small number of edges

Conclusion

Time Complexity

- Adjacent Matrix + Array: **$O(|V^2|)$**
- Adjacency List + Heap: **$O((|V| + |E|) \log(|V|))$**
  - Sparse: **$O(|V| \log(|V|))$**
  - Dense: **$O(|V|^2 \log(|V|))$**

Performance

- Adjacency Matrix + Array is better for Dense Graphs
- Adjacency List + Heap is better for Sparse Graphs