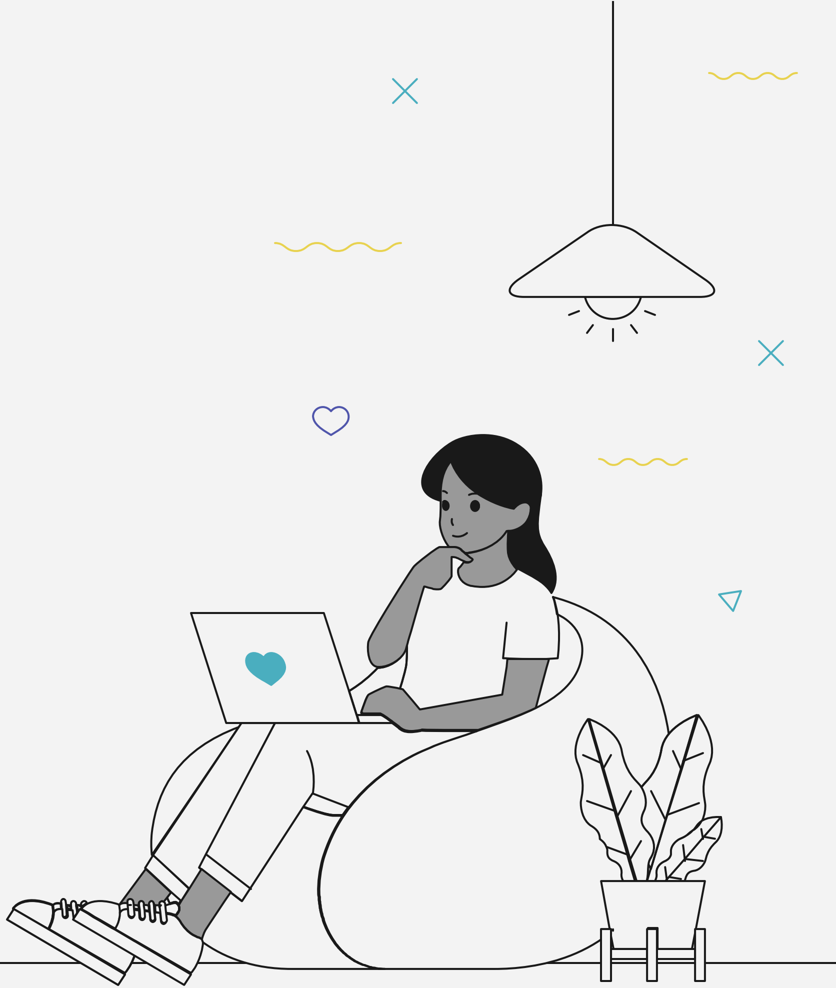
Dijkstra's Algorithm

By Team 3 SCSF:

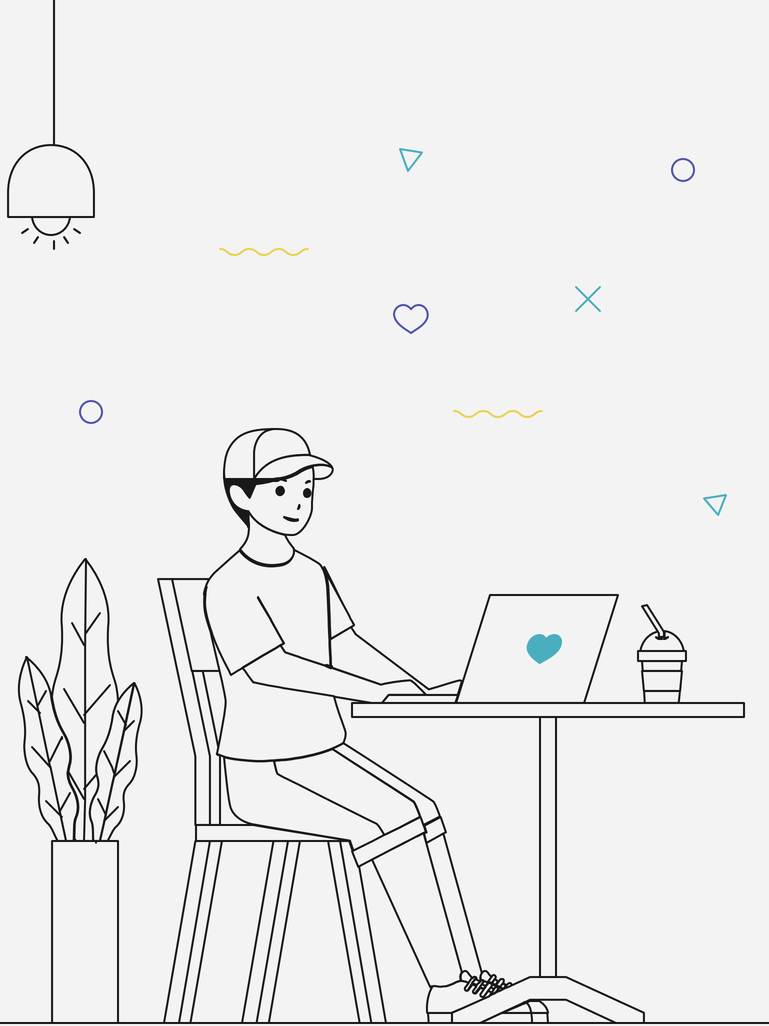
- Matthew Heng
- Siah Yee Long
- Wang Anqi



- **(a) Matthew**
- Code overview
- Theoretical time complexity
- Empirical time complexity
-
- **(b) Yee Long**
- Code overview
- Theoretical time complexity
- Empirical time complexity
-
- **C) An Qi**
- Comparison of performance between the two
- Use graph to plot
- SWOT analysis

(a)

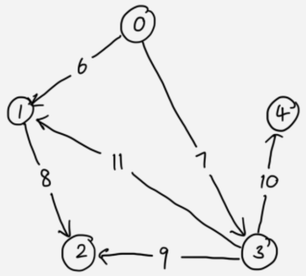
Graph representation	Adjacency matrix
Data structure for traversal	Priority queue using unsorted array



(a) Data structures

<u>Graph</u>
adjacency
matrix

`g.adj_matrix`



	0	1	2	3	4
0	0	6	-	7	-
1	-	0	8	-	-
2	-	-	0	-	-
3	-	11	9	0	10
4	-	-	-	-	0

<u>Priority Queue</u>
array

```
int visited[g->V]; // S
int d[g->V];
int prev_vertex[g->V];
// pi
```

```
          0   1   2   3   4
S  =  [                       ]
d  =  [                       ]
pi =  [                       ]
```

(a) Theoretical Time Complexity

```
        0   1   2   3   4
S =  [                    ]
d =  [                    ]
pi = [                    ]
```

Aim: Fill in S

- Using while loop
- Looping V times

(a) Theoretical Time Complexity

```c
// explore the rest of the vertices
while(count<g->V-1){
    int min_distance = INFINITY;
    int next_vertex;

    // find the smallest cost neighbour to pursue
    for(i=0;i<g->V;i++){
        if(d[i] < min_distance && !visited[i]){
            next_vertex = i;
            min_distance = d[i];
        }
    }
    visited[next_vertex] = 1;
    if(count==1) prev_vertex[next_vertex] = start; // NO I CANT JUST DO THIS

    // update the new d reachable from this new node and update the prev_vertex accordingly
    for(i=0;i<g->V;i++){
        if (!visited[i]){
            if (min_distance + g->adj.matrix[next_vertex][i] < d[i]) {
                d[i] = min_distance + g->adj.matrix[next_vertex][i];
                prev_vertex[i] = next_vertex;
            }
        }
    }
    count++;
}
```

V times

Find the frontier node to be explored

- Smallest d[i]
- Never visited before

V times

Update the d and pi of d[i] neighbors

- Loop through the row Matrix[i]

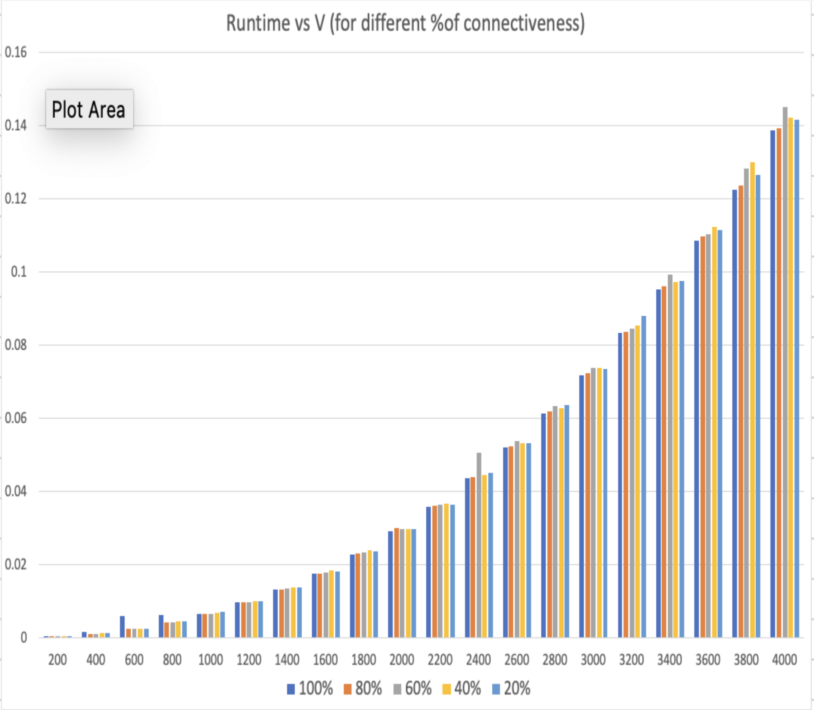
(a) Theoretical time complexity analysis

$$T(V,E) = V\,(V + V) = O(V^2)$$

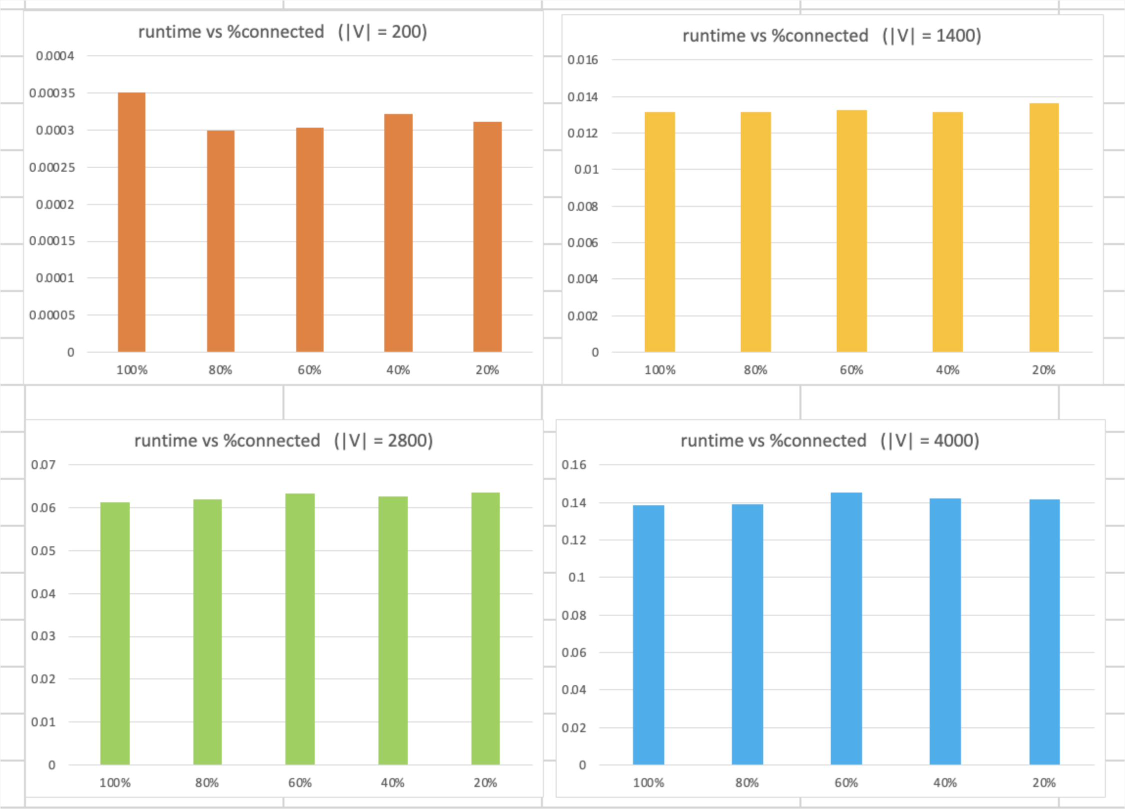
Find out best, average, worse

a) Empirical (runtime vs %of connectiveness)

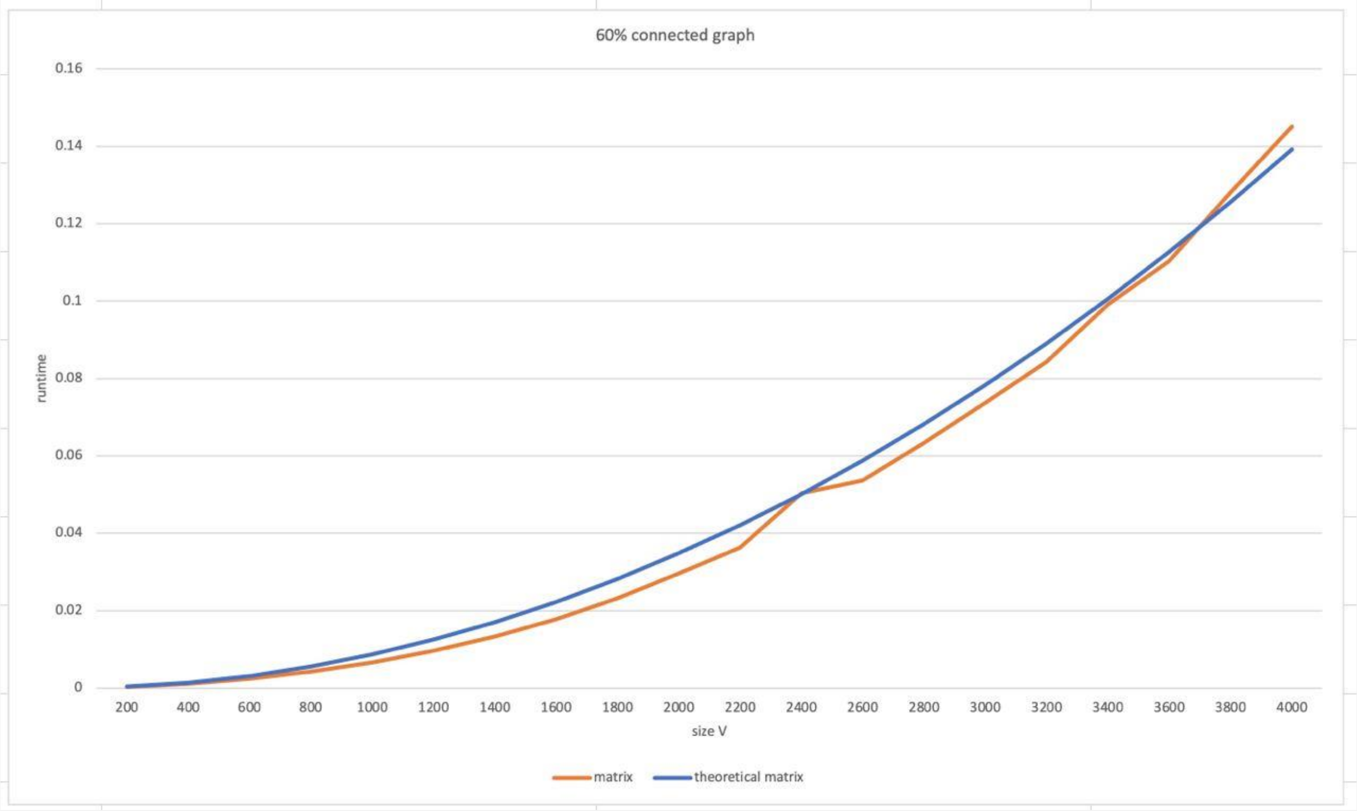
Size of graph, V	% of connected graph				
	100%	80%	60%	40%	20%
	runtime (s)	runtime (s)	runtime (s)	runtime (s)	runtime (s)
200	0.000351	0.000299	0.000304	0.000322	0.000312
400	0.001486	0.001084	0.0011	0.001117	0.001125
600	0.006012	0.002365	0.002377	0.002426	0.002441
800	0.006206	0.004213	0.004206	0.004303	0.00431
1000	0.006545	0.006581	0.006597	0.006791	0.00707
1200	0.009523	0.009525	0.009605	0.009928	0.009933
1400	0.013138	0.013178	0.013281	0.013636	0.013645
1600	0.017443	0.017602	0.017801	0.018285	0.018195
1800	0.022808	0.022926	0.023149	0.023893	0.023473
2000	0.029113	0.029994	0.02956	0.029641	0.029703
2200	0.035624	0.036088	0.036247	0.036621	0.036461
2400	0.043432	0.043814	0.050391	0.044477	0.04498
2600	0.051982	0.052352	0.053574	0.053249	0.053163
2800	0.061299	0.061939	0.063356	0.062728	0.063651
3000	0.07162	0.072241	0.073745	0.073567	0.073554
3200	0.083397	0.083636	0.08434	0.085197	0.087928
3400	0.095156	0.095922	0.099128	0.097163	0.097464
3600	0.10845	0.109505	0.110335	0.11222	0.111405
3800	0.122346	0.123558	0.128157	0.13002	0.126584
4000	0.138674	0.139326	0.145105	0.142221	0.141585



Empirical runtime vs % connected

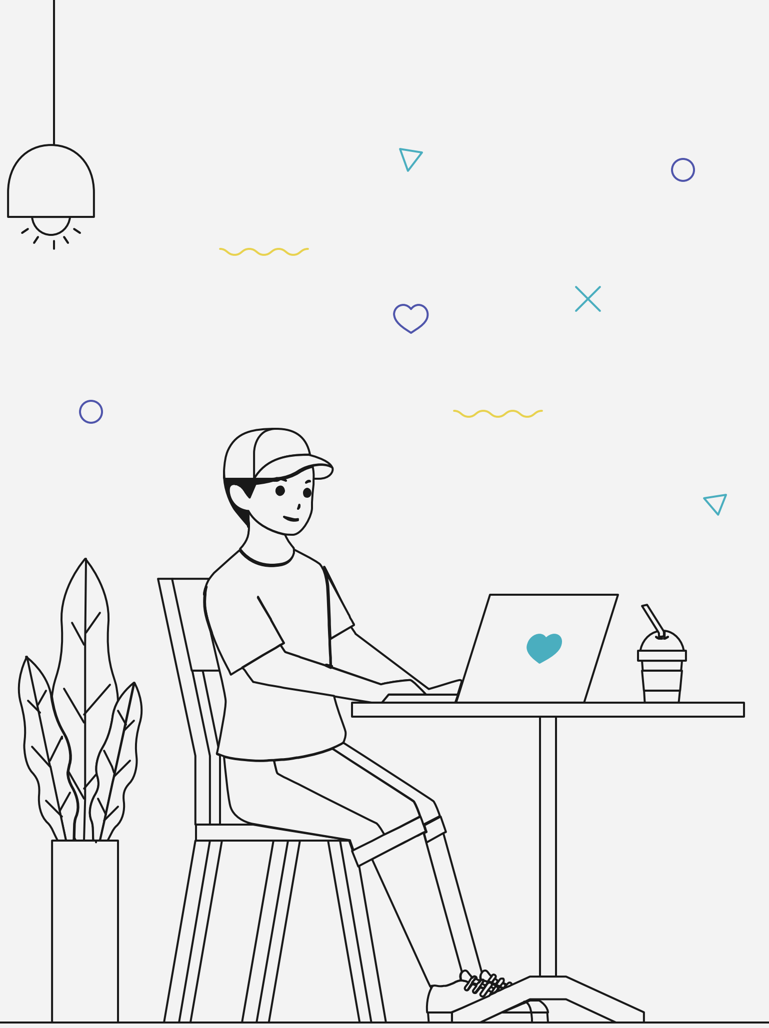


a) Empirical vs Theoretical runtime



(b)

Graph representation	Adjacency list
Data structure for traversal	Priority queue using <u>minimizing heap</u>

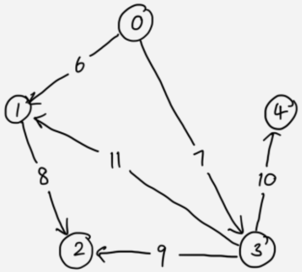


(b) Data structures

<u>Graph</u>
adjacency list

```
g.adj.list[v]

g.adj.list[v]->vertex
g.adj.list[v]->cost
g.adj.list[v]->next
```




```
0:  -(7)->3 -(6)->1
1:  -(8)->2
2:
3:  -(10)->4 -(9)->2 -(11)->1
4:
```

<u>Priority Queue</u>
heap

```
pq

pq.qvertex
pq.qcost
pq.qprev
pq.qsize
```

```
void pq_swap(PQueue *pq, int a, int b);
void pq_minHeapify(PQueue *pq, int i);
void pq_insert(PQueue *pq, int vertex, int cost, int prev);
int pq_pop(PQueue *pq);
void pq_updateCost(PQueue *pq, int vertex, int old_cost, int new_cost, int new_prev);
int pq_findVertex(PQueue *pq, int vertex, int cost, int index);
void pq_printPQ(PQueue *pq);
void pq_checkViolation(PQueue *pq);
```

(b) Pseudo-code

```
Initialise pq                of size |V|
Initialise d[]               of size |V|, all INFINITY
Initialise visited[]         of size |V|, all 0
Initialise prev_vertex[]     of size |V|, all -1


d[start] = 1
visited[start] = 1
For each vertex adjacent to start:
          d[vertex] = cost of start → vertex)
          prev_vertex[vertex] = start
          pq_insert(pq, vertex, d[vertex])

While pq is not empty:
          new = pq_pop(pq)
          visited[new] = 1
```

(b) Pseudo-code

```
While pq is not empty:
        new = pq_pop(pq)
        visited[new] = 1

        For each vertex adjacent to new:
                if vertex not visited, AND new shortest path found for vertex:
                        d[vertex] = new cost
                        pq_updateCost(pq, vertex, old cost, new cost)
```

(b) Theoretical time complexity analysis

```c
void pq_swap(PQueue *pq, int a, int b)
{
    // this function simply swaps the data of vertex at index a and vertex at index b
    int temp_vertex = pq->qvertex[a];
    int temp_cost = pq->qcost[a];
    int temp_prev = pq->qprev[a];

    pq->qvertex[a] = pq->qvertex[b];
    pq->qcost[a] = pq->qcost[b];
    pq->qprev[a] = pq->qprev[b];

    pq->qvertex[b] = temp_vertex;
    pq->qcost[b] = temp_cost;
    pq->qprev[b] = temp_prev;
}
```

pq_swap = O(1)

```c
void pq_minHeapify(PQueue *pq, int i)
{
    // this function floats the smallest cost item to the top
    int smallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < pq->size && pq->qcost[left] < pq->qcost[smallest])
        smallest = left;

    if (right < pq->size && pq->qcost[right] < pq->qcost[smallest])
        smallest = right;

    if (smallest != i) {
        pq_swap(pq, i, smallest);
        pq_minHeapify(pq, smallest);
    }
}
```

pq_minHeapify = O(lg[n])

(b) Theoretical time complexity analysis

```c
void pq_insert(PQueue *pq, int vertex, int cost, int prev)
{
    // at every insertion, float the smallest cost vertex to the top
    int i = pq->size++;
    pq->qvertex[i] = vertex;
    pq->qcost[i] = cost;
    pq->qprev[i] = prev;

    while (i != 0 && pq->qcost[i] < pq->qcost[(i - 1) / 2]) {
        pq_swap(pq, i, (i - 1) / 2);
        i = (i - 1) / 2;
    }
}
```

pq_insert = O(lg[n])

```c
int pq_pop(PQueue *pq)
{
    // this function pops the vertex with the smallest cost and minHeapifies to fix the heap
    if (pq->size == 0) {
        printf("Queue is empty!\n");
        return -1;
    }
    int smallest = pq->qvertex[0];

    pq->qvertex[0] = pq->qvertex[pq->size-1];
    pq->qcost[0] = pq->qcost[pq->size-1];
    pq->qprev[0] = pq->qprev[pq->size-1];
    pq->size--;
    pq_minHeapify(pq, 0);

    return smallest;
}
```

pq_pop = pq_minHeapify = O(lg[n])

(b) Theoretical time complexity analysis

```c
int pq_findVertex(PQueue *pq, int vertex, int cost, int index)
{
    // this function returns the index of the found vertex
    if(pq->qvertex[index] == vertex) return index;

    int found;
    int left = index*2+1;
    if(left < pq->size && pq->qcost[left] <= cost){
        found = pq_findVertex(pq, vertex, cost, left);
        if(found!=-1 && pq->qvertex[found] == vertex) return found;
    }

    int right = index*2+2;
    if(right < pq->size && pq->qcost[right] <= cost){
        found = pq_findVertex(pq, vertex, cost, right);
        if(found!=-1 && pq->qvertex[found] == vertex) return found;
    }

    return -1;
}
```

pq_findVertex = O(lg[n])

```c
void pq_updateCost(PQueue *pq, int vertex, int old_cost, int new_cost, int new_prev)
{
    // this function will search for the vertex and update its cost, then fix it

    int index = pq_findVertex(pq, vertex, old_cost, 0); // find the index of the target vertex

    // update the new cost and prev
    pq->qcost[index] = new_cost;
    pq->qprev[index] = new_prev;

    // float the vertex up due to the new cost potentially being smaller than the parent
    while (index != 0 && pq->qcost[index] < pq->qcost[(index - 1) / 2]) {
        pq_swap(pq, index, (index - 1) / 2);
        index = (index - 1) / 2;
    }
}
```

pq_updateCost = pq_findVertex + O(lg[n])
= O(lg[n])

(b) Theoretical time complexity analysis

```c
void dijkstra_list(Graph *g, int start)
.
.
.
// mark start as visited and explore it
visited[start] = 1;
d[start] = 0;
while(vertex != NULL){
    d[vertex->vertex] = vertex->cost;
    prev_vertex[vertex->vertex] = start;
    vertex = vertex->next;
}

for(i=0;i<g->V;i++)
    pq_insert(pq, i, d[i], prev_vertex[i]);

// while pq is not empty, extract cheapest vertex and explore it
while(pq->size){
    // extract cheapest vertex
    new_vertex = pq_pop(pq);
    visited[new_vertex] = 1;

    // explore new vertex
    vertex = g->adj.list[new_vertex];
    while(vertex!=NULL){
        // if there is a new shorter path found, update the pq and d
        if(!visited[vertex->vertex] && d[vertex->vertex] > d[new_vertex]+vertex->cost){
            int old_cost = d[vertex->vertex];
            d[vertex->vertex] = d[new_vertex]+vertex->cost;
            pq_updateCost(pq, vertex->vertex, old_cost,d[vertex->vertex], new_vertex);
            prev_vertex[vertex->vertex] = new_vertex;
            // pq_checkViolation(pq);
        }
        vertex = vertex->next;
    }
}
```

O(V) + O(V*lg[V]) = **O(V*lg[V])**

V

O(lg[V])

E/V

O(lg[V])

(b) Theoretical time complexity analysis

```c
void dijkstra_list(Graph *g, int start)
.
.
.
// mark start as visited and explore it
visited[start] = 1;
d[start] = 0;
while(vertex != NULL){
    d[vertex->vertex] = vertex->cost;
    prev_vertex[vertex->vertex] = start;
    vertex = vertex->next;
}

for(i=0;i<g->V;i++)
    pq_insert(pq, i, d[i], prev_vertex[i]);

// while pq is not empty, extract cheapest vertex and explore it
while(pq->size){
    // extract cheapest vertex
    new_vertex = pq_pop(pq);
    visited[new_vertex] = 1;

    // explore new vertex
    vertex = g->adj.list[new_vertex];
    while(vertex!=NULL){
        // if there is a new shorter path found, update the pq and d
        if(!visited[vertex->vertex] && d[vertex->vertex] > d[new_vertex]+vertex->cost){
            int old_cost = d[vertex->vertex];
            d[vertex->vertex] = d[new_vertex]+vertex->cost;
            pq_updateCost(pq, vertex->vertex, old_cost,d[vertex->vertex], new_vertex);
            prev_vertex[vertex->vertex] = new_vertex;
            // pq_checkViolation(pq);
        }
        vertex = vertex->next;
    }
}
```

O(V) + O(V*lg[V]) = **O(V*lg[V])**

V

O(lg[V])

E/V

O(lg[V])

V (O(lg[V]) + E/V (O(lg[V])))

= O(V*lg([V])) + O(E*lg([V]))

= **O([V+E]*lg([V]))**

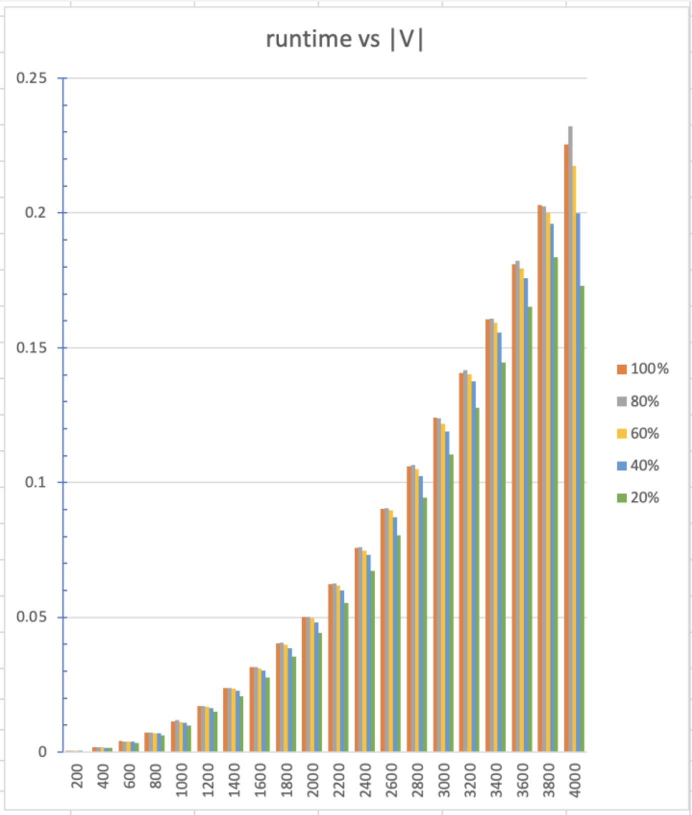
(b) Theoretical time complexity analysis

$$\underbrace{O(V*lg[V]) \quad + \quad O([V+E]*lg([V]))}$$

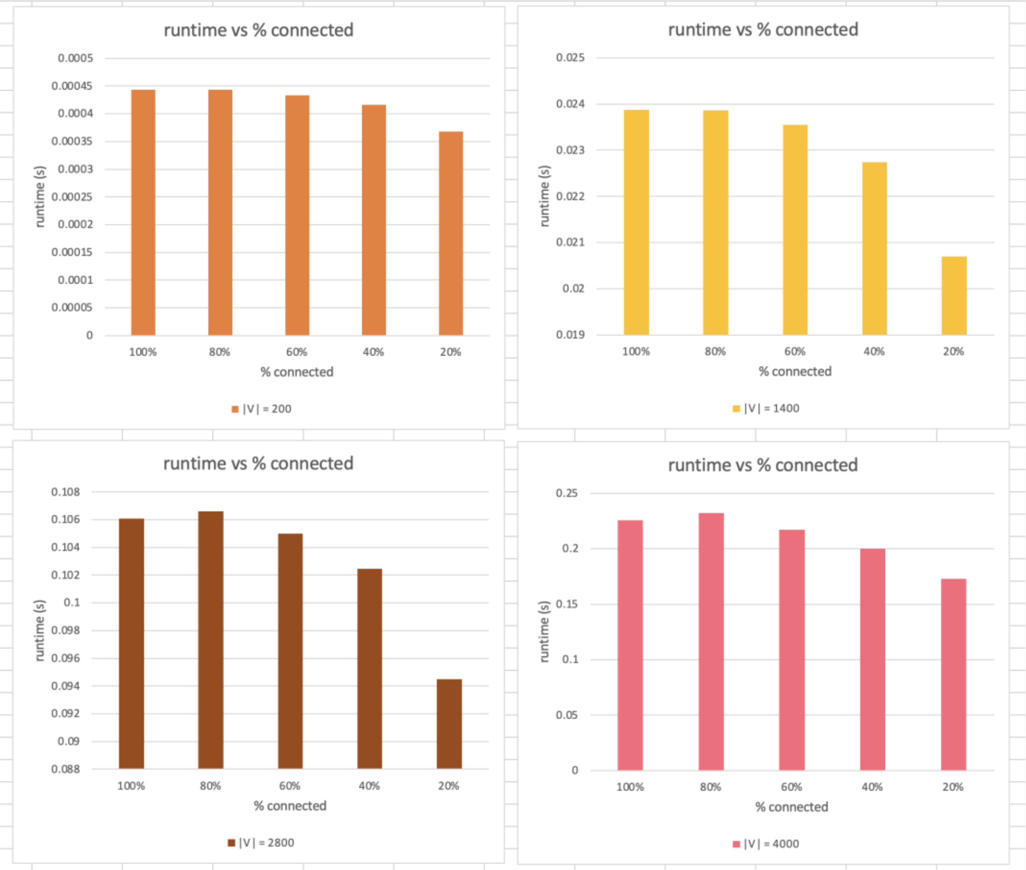
$$O([V+E]*lg([V]))$$

(b) Empirical time complexity

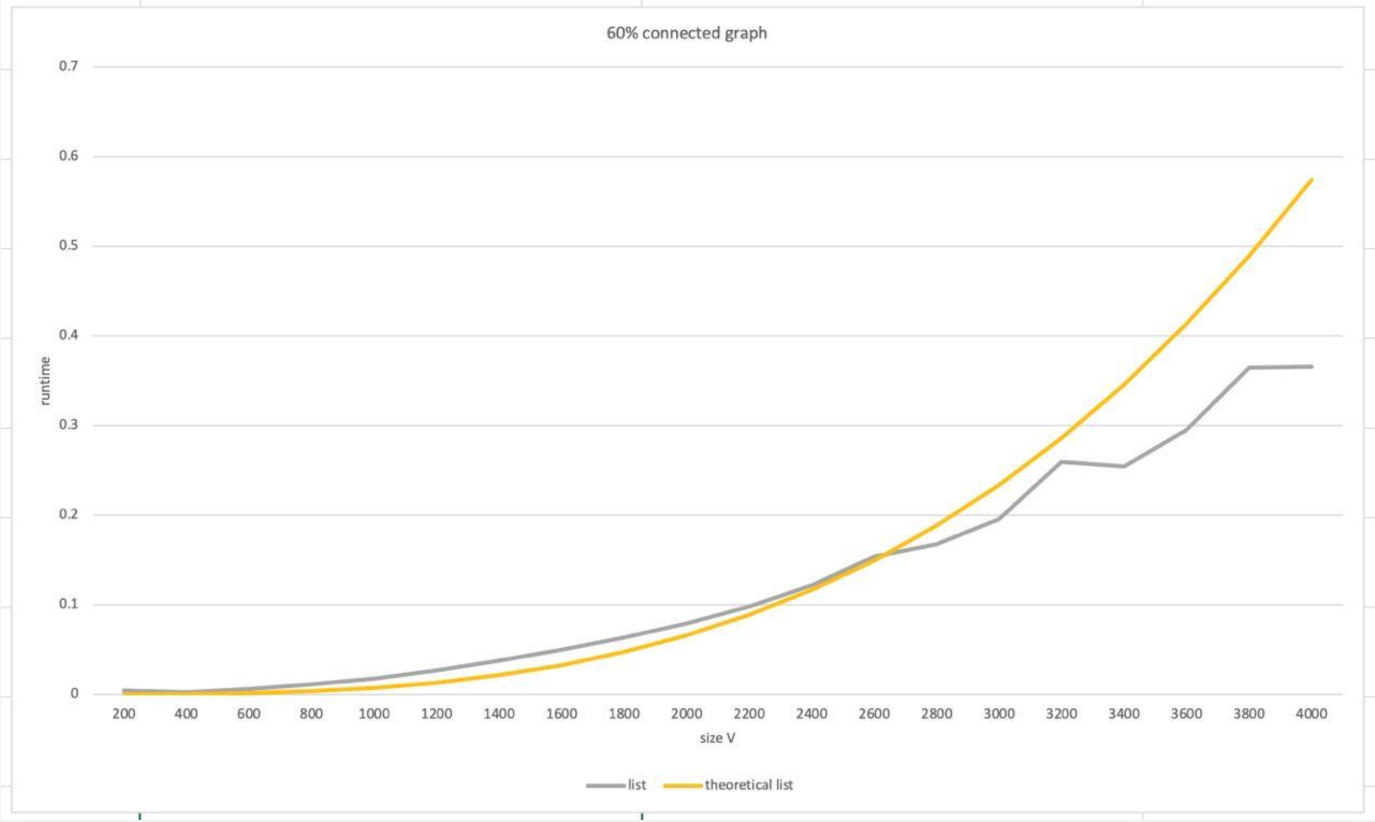
		% connected graph				
		100%	80%	60%	40%	20%
size of graph, V	200	0.000443	0.000443	0.000433	0.000416	0.000368
	400	0.001732	0.001743	0.00171	0.001648	0.001471
	600	0.004042	0.004007	0.003947	0.003805	0.003414
	800	0.007294	0.007313	0.007119	0.006891	0.006198
	1000	0.011448	0.011846	0.011204	0.010839	0.009819
	1200	0.017182	0.017186	0.0169	0.016329	0.014883
	1400	0.023874	0.023859	0.023548	0.022739	0.020697
	1600	0.031436	0.031619	0.031153	0.030183	0.027592
	1800	0.0404	0.040736	0.039817	0.038522	0.035358
	2000	0.050289	0.050286	0.049606	0.048134	0.044344
	2200	0.062401	0.062634	0.0618	0.060048	0.055439
	2400	0.075864	0.075935	0.074836	0.073214	0.067367
	2600	0.090346	0.090448	0.089688	0.087094	0.080458
	2800	0.106087	0.106623	0.104979	0.102447	0.094485
	3000	0.124031	0.123915	0.121682	0.118881	0.110291
	3200	0.140675	0.141581	0.140082	0.13766	0.127851
	3400	0.160557	0.160852	0.159331	0.155558	0.144458
	3600	0.180906	0.18218	0.179347	0.17574	0.165109
	3800	0.202853	0.202438	0.2002	0.196016	0.183532
	4000	0.225478	0.232061	0.217394	0.199833	0.173005



(b) Empirical time complexity

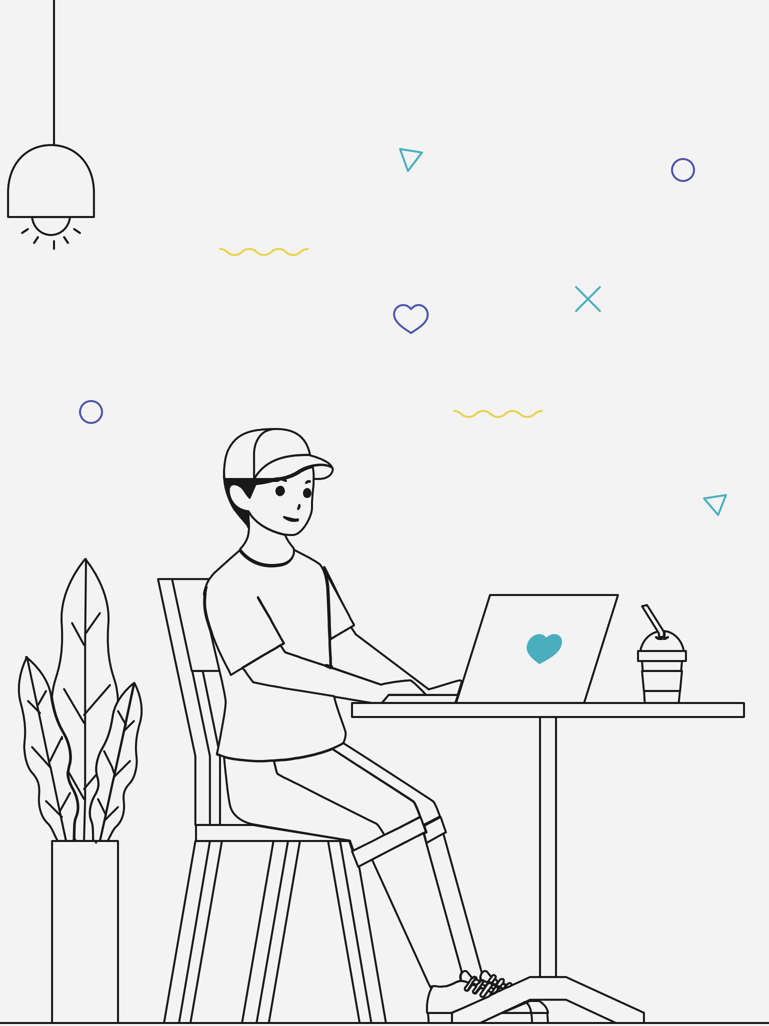


b) Empirical vs Theoretical runtime



(c)

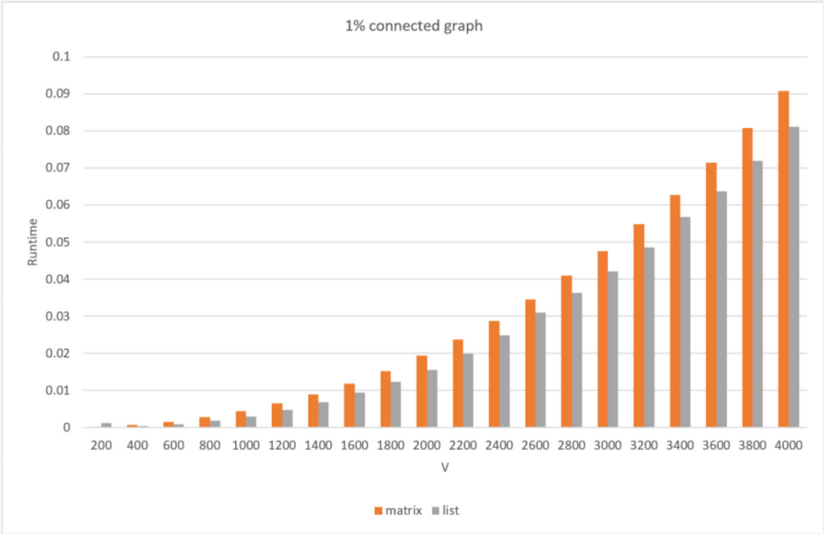
Comparison between (a) and (b)

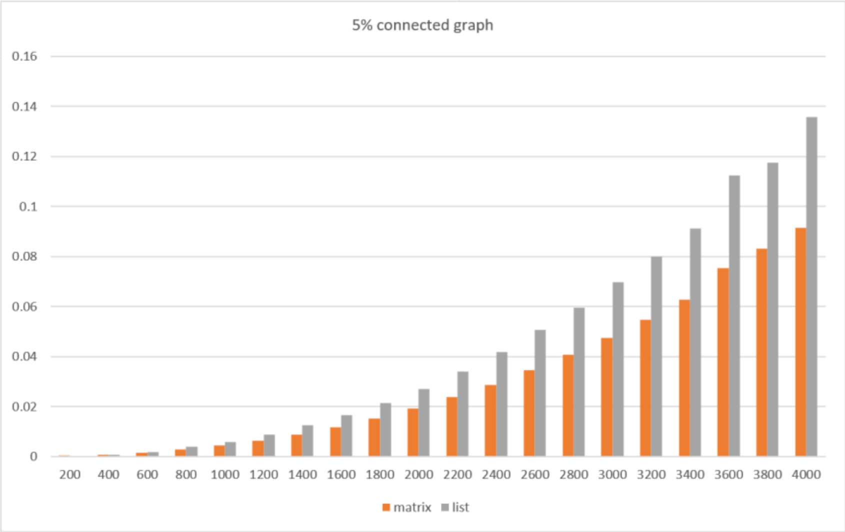


(c) Time complexity

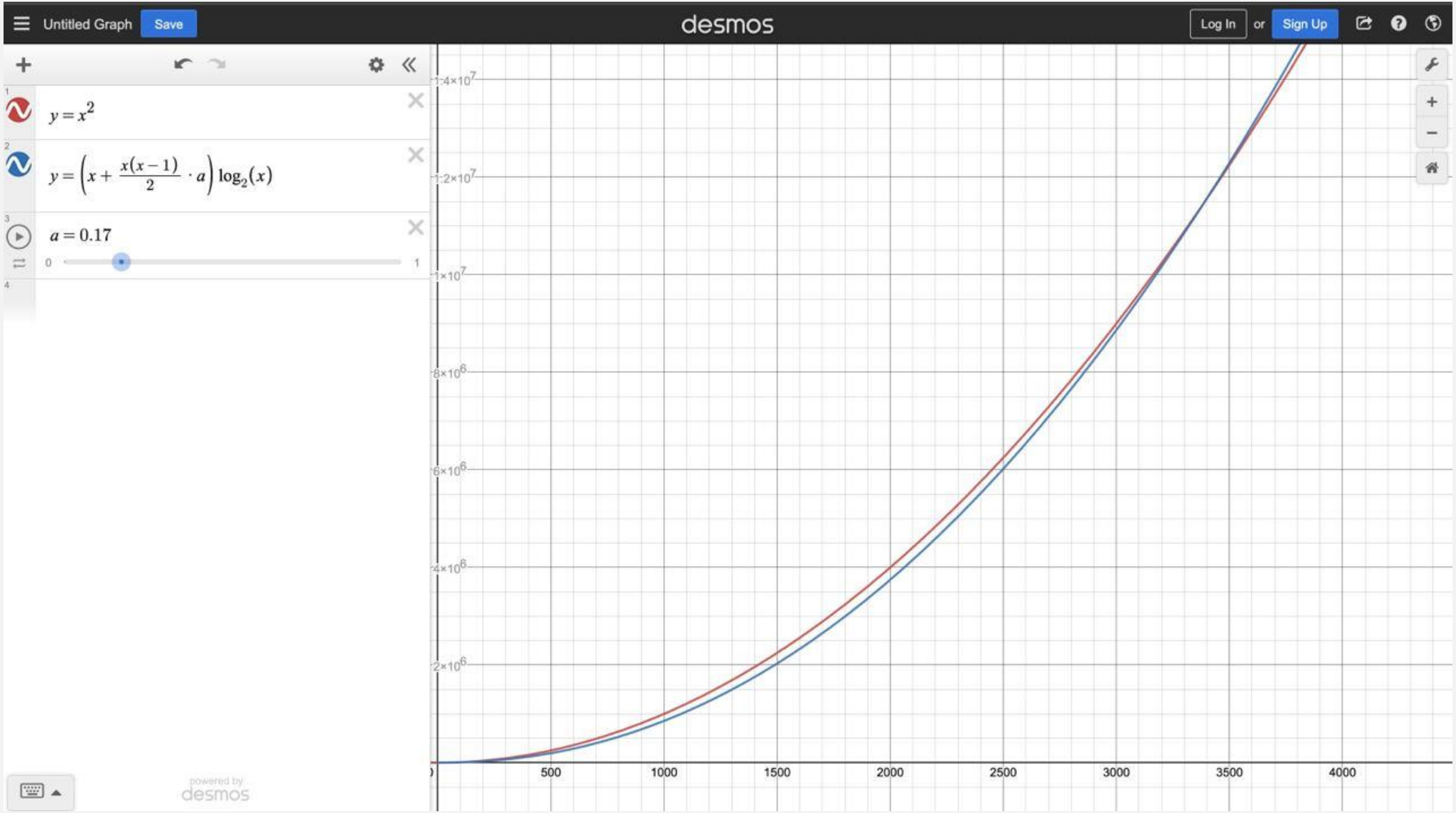
	Adjacency matrix	Adjacency list
Theoretical time complexity	O(V^2)	O([V+E]*lg([V]))

(c) Empirical time complexity

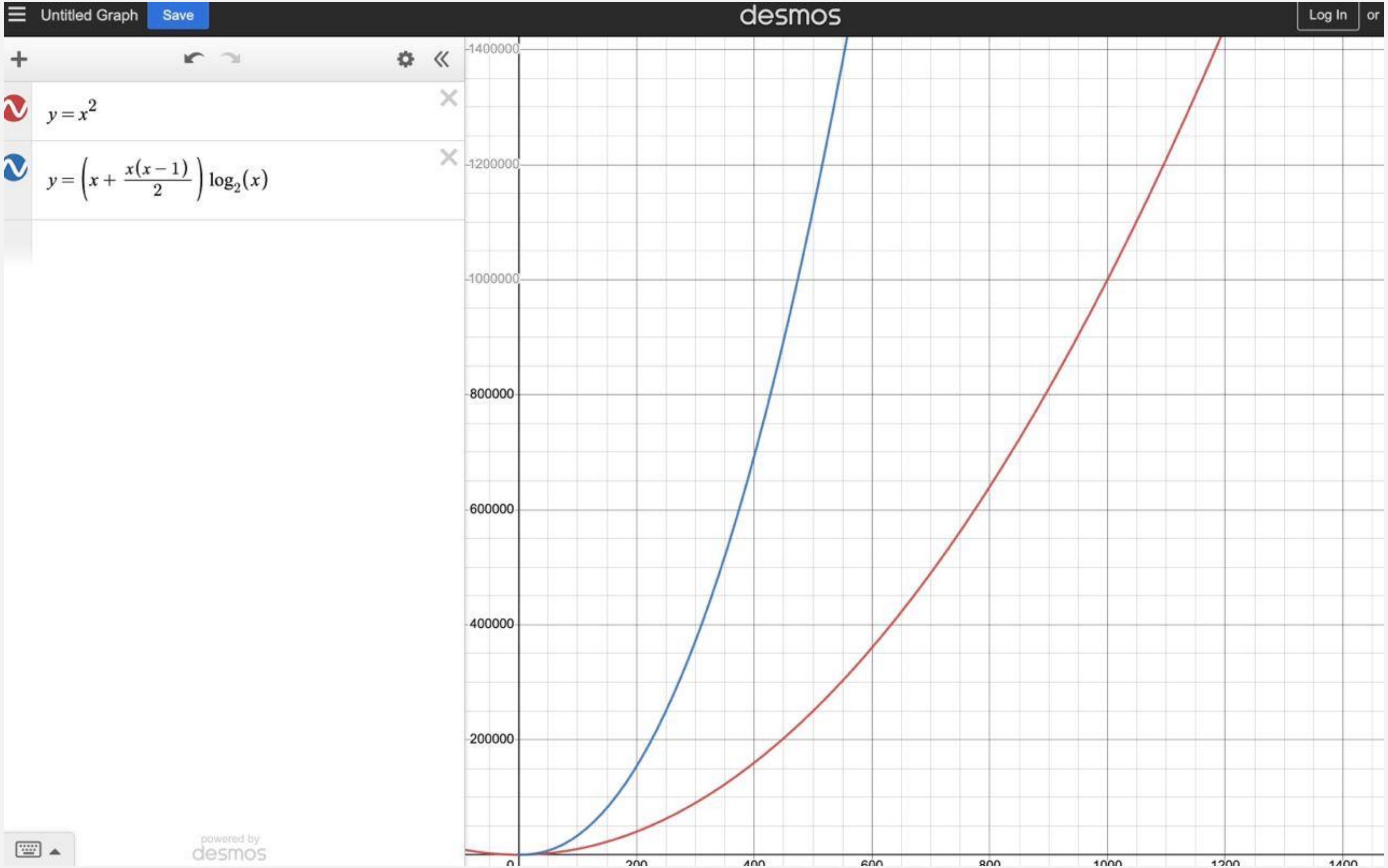




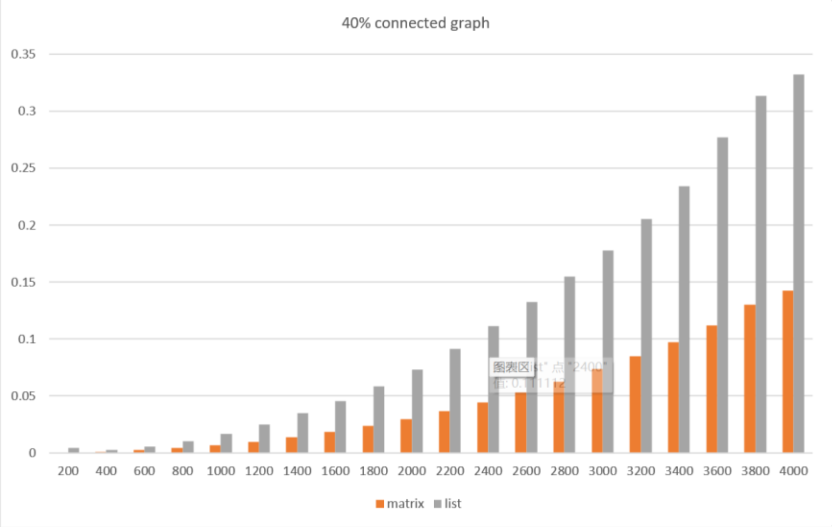
(c) Time complexity

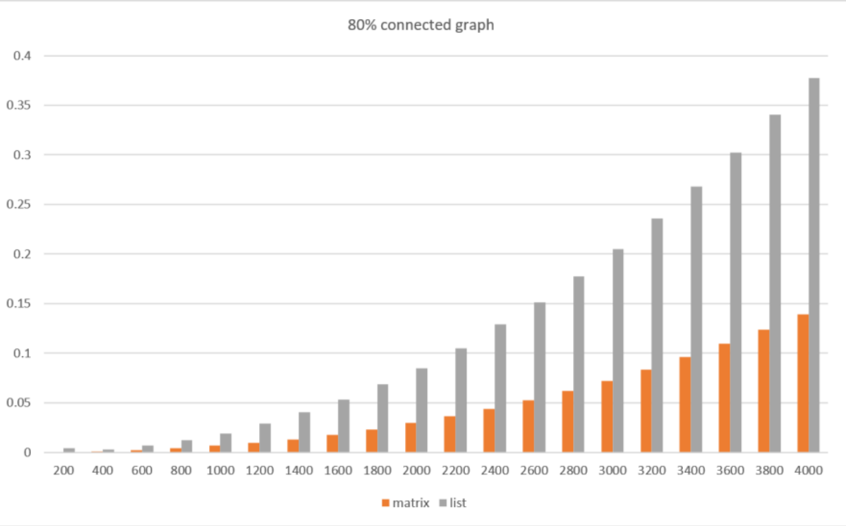


(c) Time complexity

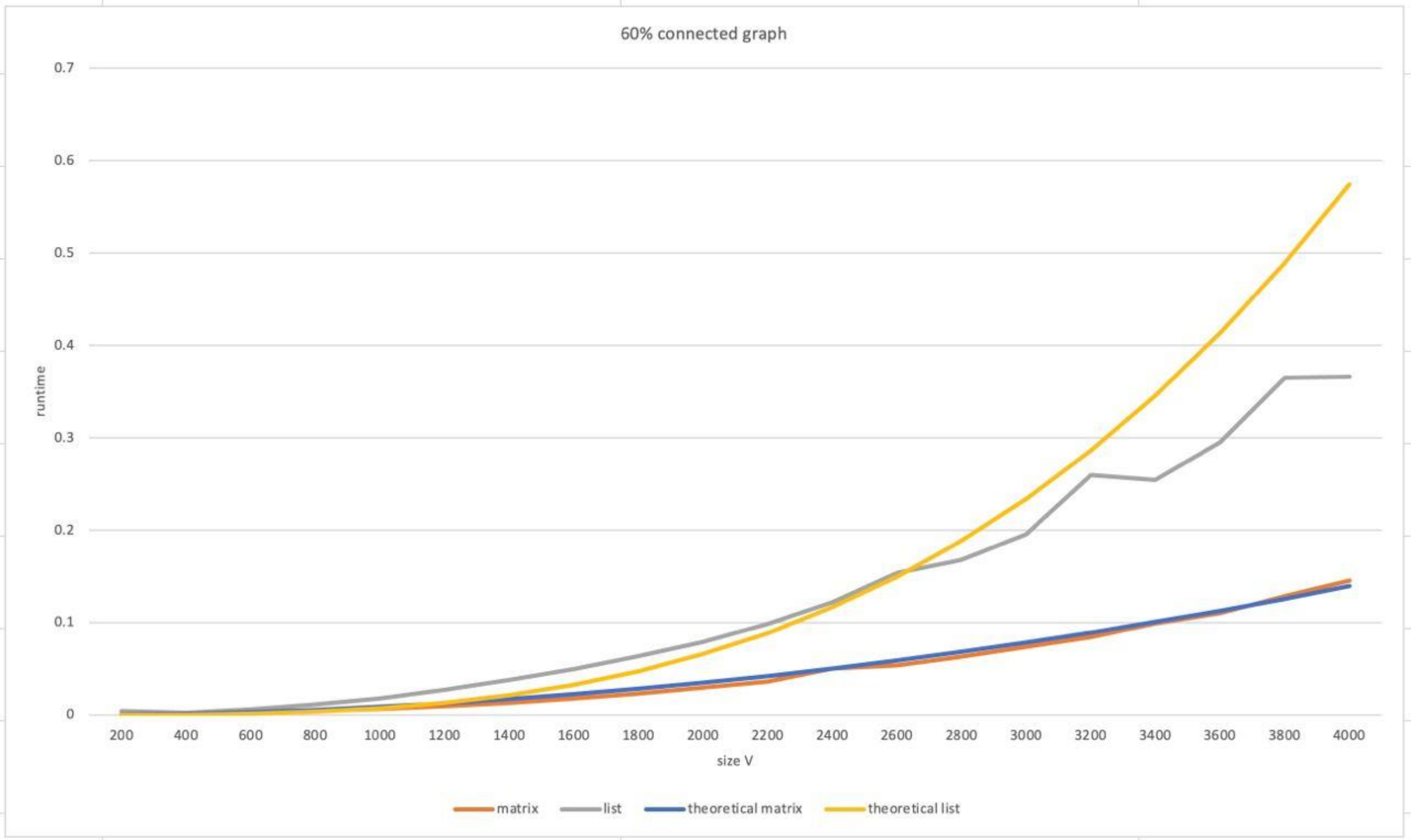


(c) Empirical time complexity





(c) Time complexity

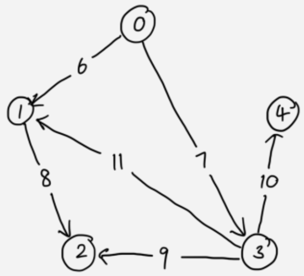


(c) Memory Usage

(a) Data structures

Graph
adjacency matrix

`g.adj_matrix`



	0	1	2	3	4
0	0	6	-	7	-
1	-	0	8	-	-
2	-	-	0	-	-
3	-	11	9	0	10
4	-	-	-	-	0

Priority Queue
array

```
int visited[g->V]; // S
int d[g->V];
int prev_vertex[g->V];
// pi
```

```
        0   1   2   3   4
S  =  [                     ]
d  =  [                     ]
pi =  [                     ]
```

(c) Memory Usage

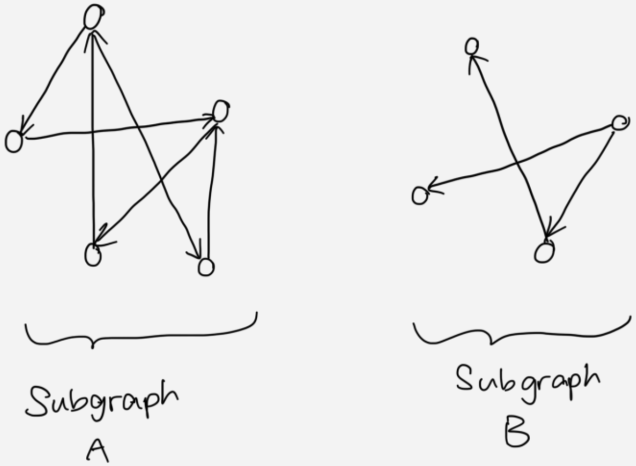
For **dense** graphs (where |E| is close to |V|^2), the **adjacency matrix** approach consumes **more memory**. In contrast, for **sparse** graphs, the **adjacency list** approach is **more memory-efficient**.

Comparison

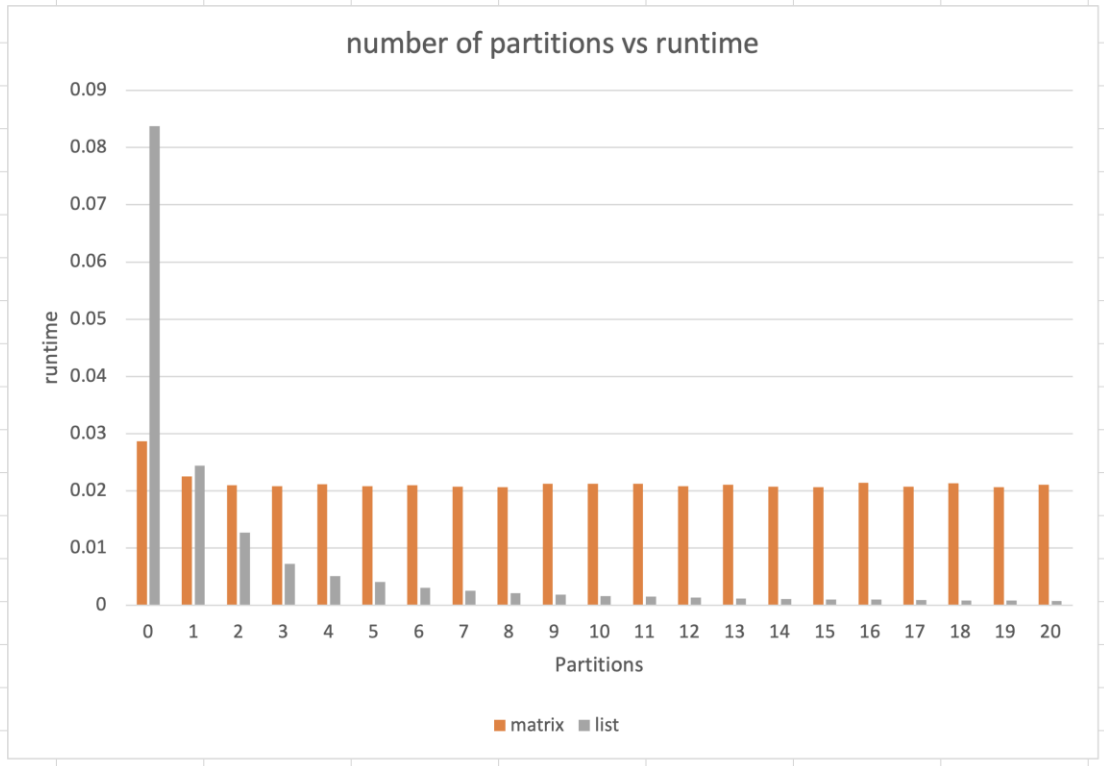
	Time Complexity	Memory Usage	Ease of Implementation
Adjacency matrix	O(V^2)	• dense graphs consumes more memory	• conceptually simpler to implement
Adjacency list	O([V+E]*lg([V]))	• dense graphs consumes less memory	• more complex to implement
Conclusion	the adjacency matrix is better in most scenarios	• adjacency list approach is more memory-efficient in most cases	

Just one more thing…

What about graphs which are disjoint?



What about graphs which are disjoint?



Thanks!