SC2001 LAB 2 PRESENTATION

By Nisha,
Zixiao, Maru &
Shruti

ADJACENCY MATRIX+ARRAY: O(V^2)

```
def dijkstra_matrix_array(graph, source):
    V = len(graph)
    INF = float('inf')
    dist = [INF] * V
    visited = [False] * V
    dist[source] = 0

    for _ in range(V):      # O(V) iterations
        # Find min unvisited vertex - O(V)
        u = None
        min_dist = INF
        for i in range(V):  # O(V) per iteration
            if not visited[i] and dist[i] < min_dist:
                min_dist = dist[i]
                u = i

        if u is None or min_dist == INF:
            break

        visited[u] = True

        # Update neighbors - O(V)
        for v in range(V):   # O(V) per iteration
            if not visited[v] and graph[u][v] != INF:
                new_dist = dist[u] + graph[u][v]
                if new_dist < dist[v]:
                    dist[v] = new_dist  # O(1) update

    return dist
```

ADJACENCY LIST+HEAP: O((V+E) LOG V)

```python
def dijkstra_list_heap(graph, source):
    V = len(graph)
    dist = [math.inf] * V      # O(V)
    dist[source] = 0
    visited = [False] * V      # O(V)

    pq = [(0, source)]         # O(1)
    heapq.heapify(pq)          # O(1) - already heapified

    while pq:    # O(V) iterations
        d, u = heapq.heappop(pq)      # O(log E) per iteration
        if visited[u]:
            continue
        visited[u] = True

        for v, w in graph[u]:    # O(degree(u)) per iteration
            if not visited[v] and dist[v] > d + w:
                dist[v] = d + w
                heapq.heappush(pq, (dist[v], v))    # O(log E) per push

    return dist
```

THEORETICAL TIME COMPLEXITY

each vertex can connect to at most V-1 other vertices
∴ max degree = V-1

$$T.\ \text{Matrix, array} = \overbrace{O(V)}^{\text{init}} + \overbrace{(V-1)\,O(V)}^{\text{find min distance unvisited vertex}} + \overbrace{(V-1)\,O(V)}^{\text{update neighbour's distance}}$$

$$= O(V^2)\ //$$

$$T.\ \text{List, heap} = \overbrace{O(V)}^{\text{init}} + \overbrace{V \times O(\log_2 V)}^{\text{heap pop } |V| \text{ to extract min. dist.}} + \overbrace{E \times O(\log_2 E)}^{|E| \text{ update dist. to vertex in PQ by pushing duplicate}}$$

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

theoretical imp. uses the decrease key $O(\log V)$ but in practice:
- ↑ constant factors
- complex imp. w/ custom python heap

→ slow operations

Since $E \leq V^2 \rightarrow \log E \leq 2 \log V$

$\therefore\ O((V+E)\log E) = O((V+E)\log V)$ asymptotically

Sparse graphs:

$|E| \rightarrow |V|$, (edges grow linearly w/ vertices)

adj list: $O((V+E)\log V)$

↓

$O((V+V)\log V)$

↓

$O(V \log V) < O(V^2)$

∴ adj list more efficient

Dense graphs:

$|E| \rightarrow |V^2|$, (edges grow quardratically w/ vertices)

adj. list: $O((V+E)\log V)$

↓

$O((V+V^2)\log V)$

↓

$O(V^2 \log V) > O(V^2)$

∴ adj. matrix would be more efficient

TIME COMPLEXITY

Adjacency matrix+Array: O(V^2)

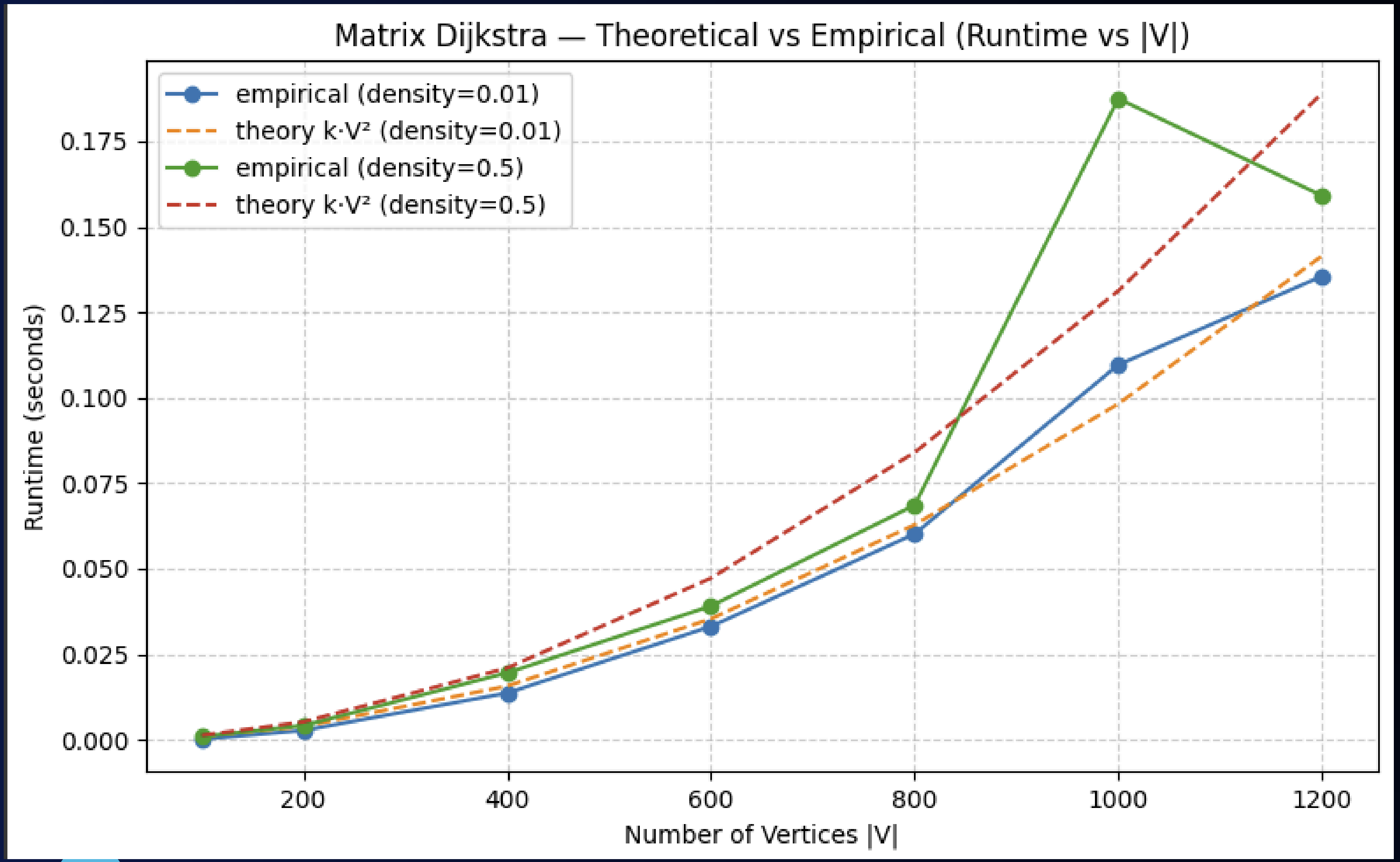
- Matrix checks all possible edges - wasteful for sparse
- Predictable, always $O(V^2)$ performance - good for dense
- Only depends on V - independent of E

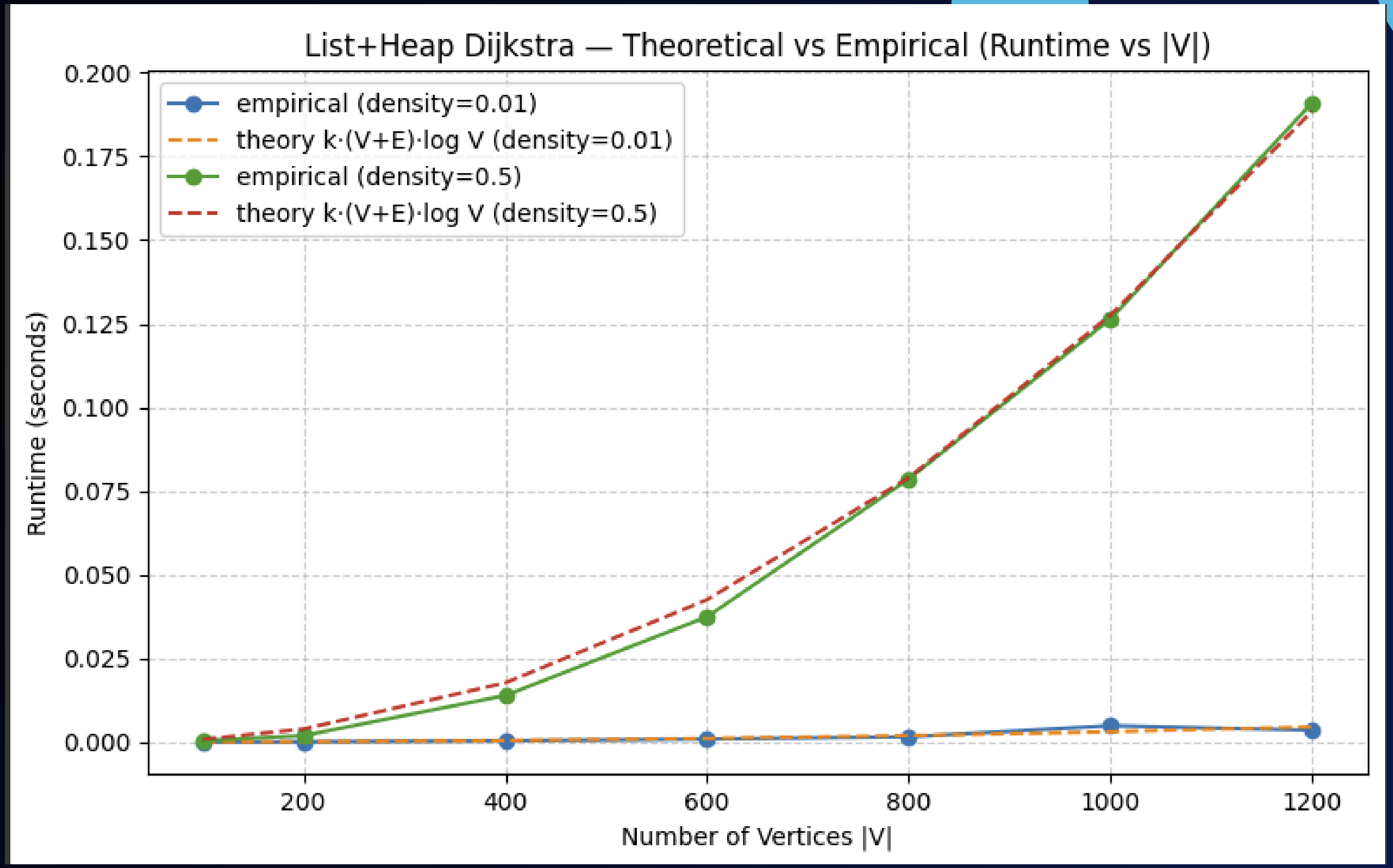
Adjacency list+heap: O((V+E) log V)

- Lists only process actual edges - perfect for sparse
- Depends on both V and E
- Memory efficient

THEORETICAL VS EMPIRICAL

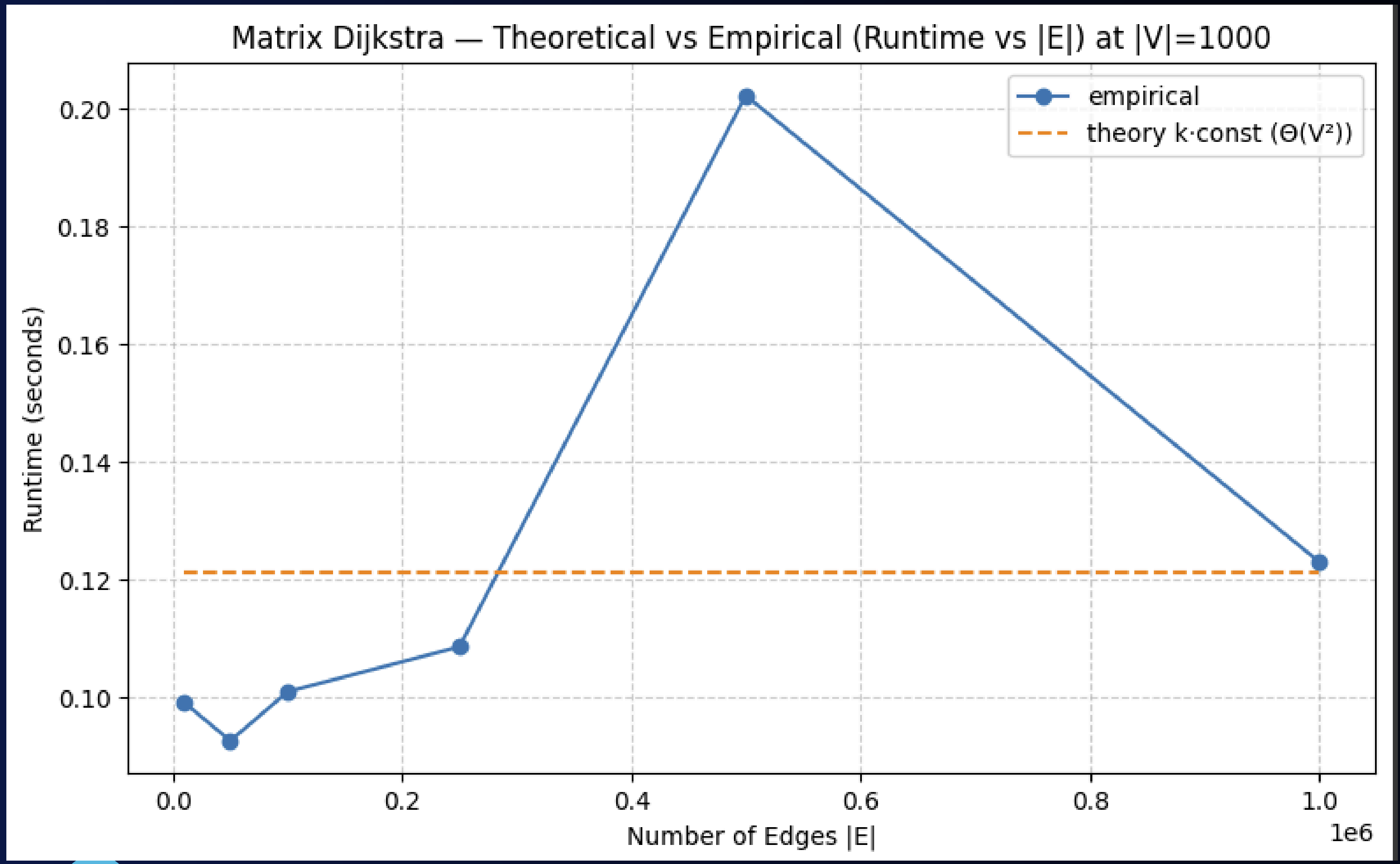
1. Runtime vs |V|

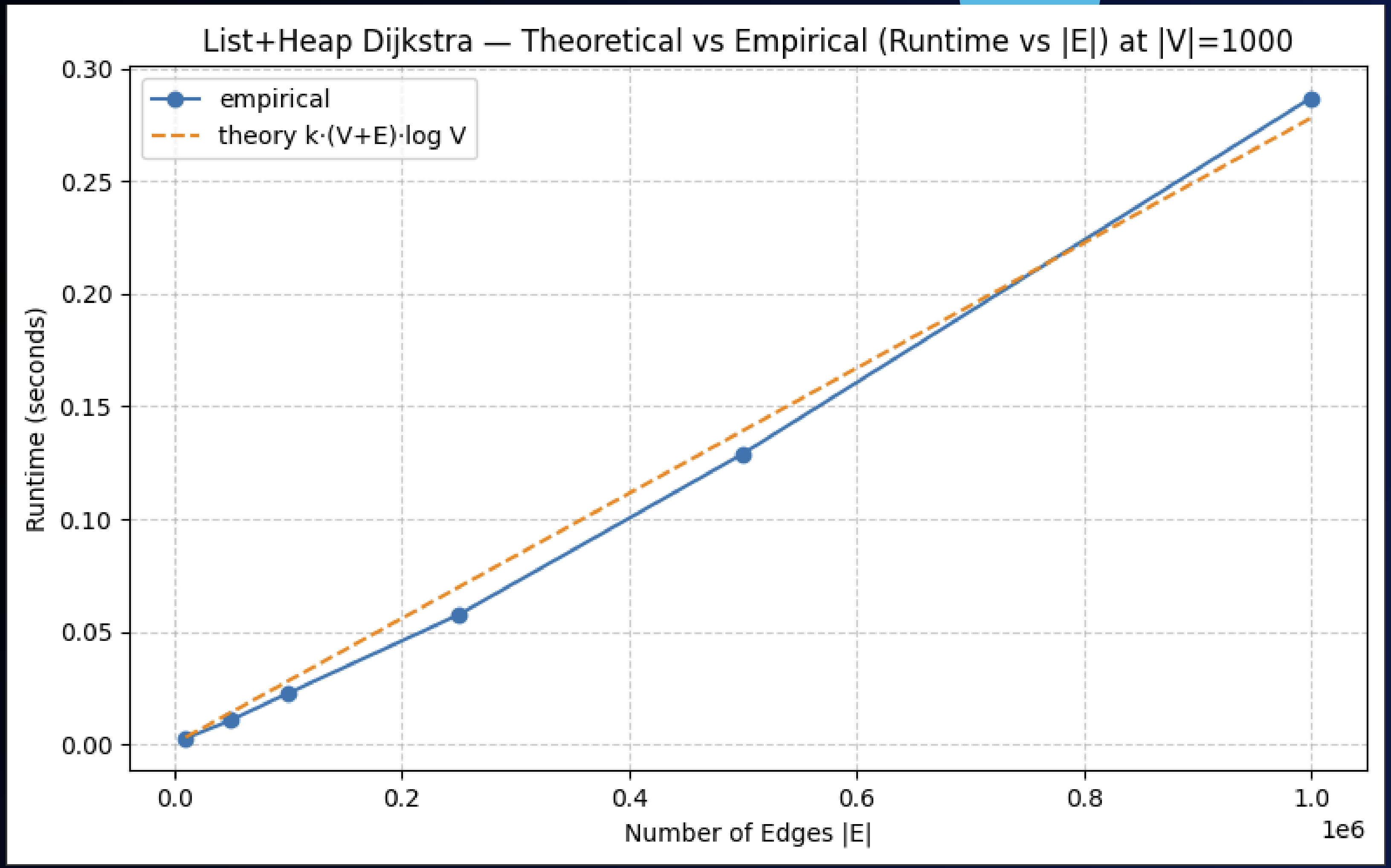




THEORETICAL VS EMPIRICAL

1. Runtime vs |E| (V=1000)

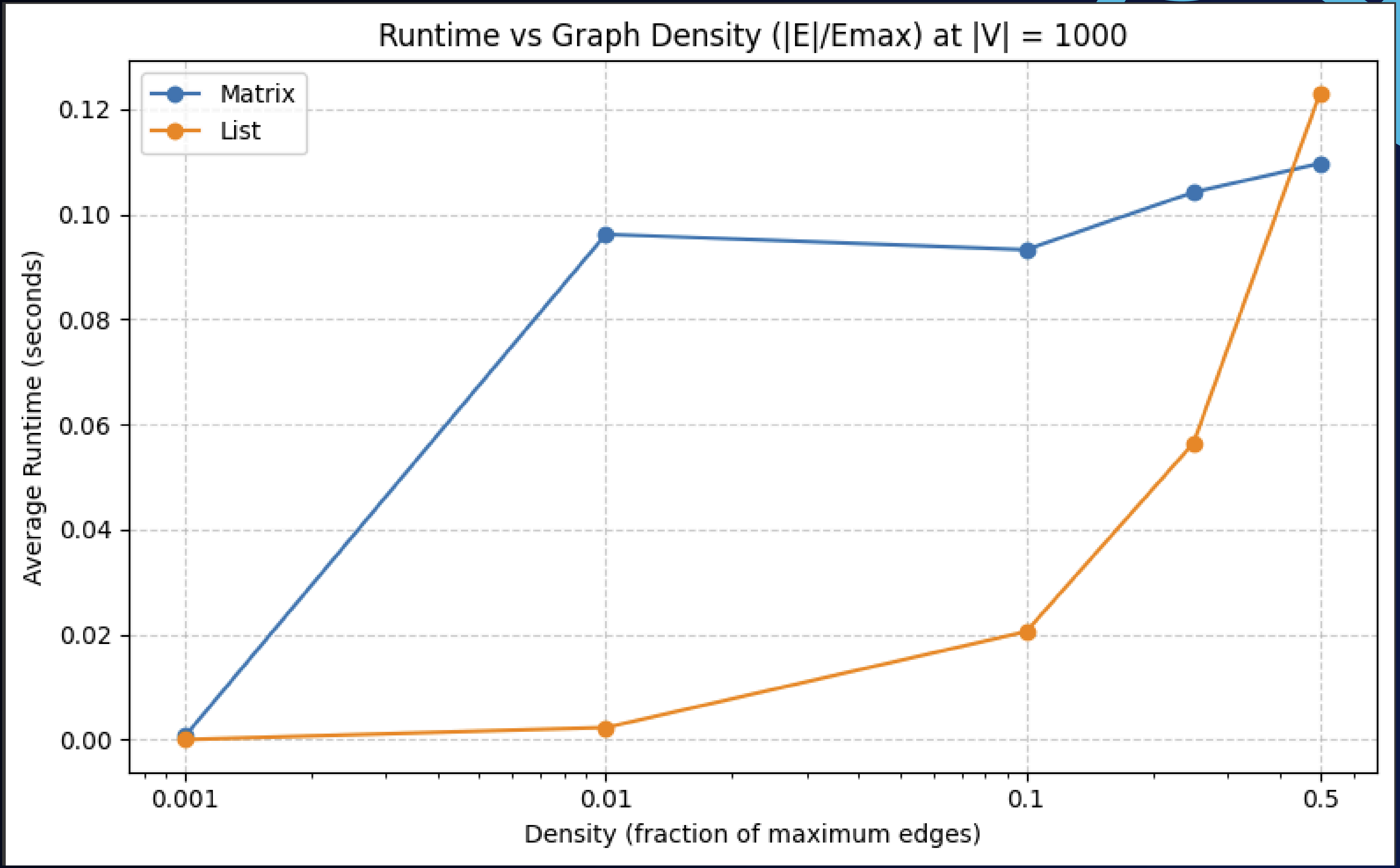




(C) COMPARING THE ALGORITHMNS

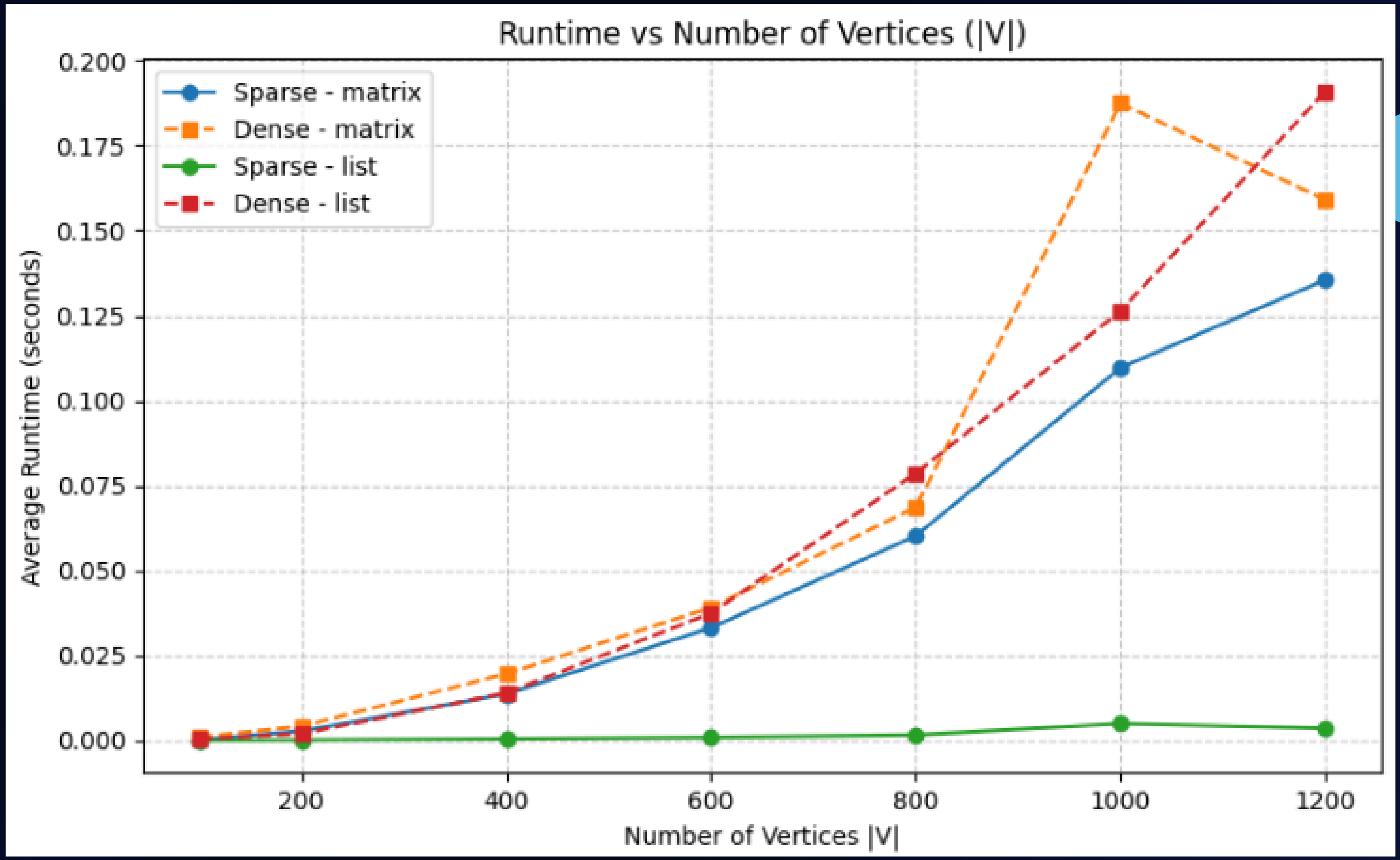
1. Varying density (V=1000)

	V	E	density	representation	time_sec
0	1000	999	0.001	matrix	0.000749
1	1000	999	0.001	list	0.000009
2	1000	9990	0.010	matrix	0.096160
3	1000	9990	0.010	list	0.002275
4	1000	99900	0.100	matrix	0.093225
5	1000	99900	0.100	list	0.020524
6	1000	249750	0.250	matrix	0.104150
7	1000	249750	0.250	list	0.056406
8	1000	499500	0.500	matrix	0.109626
9	1000	499500	0.500	list	0.123029

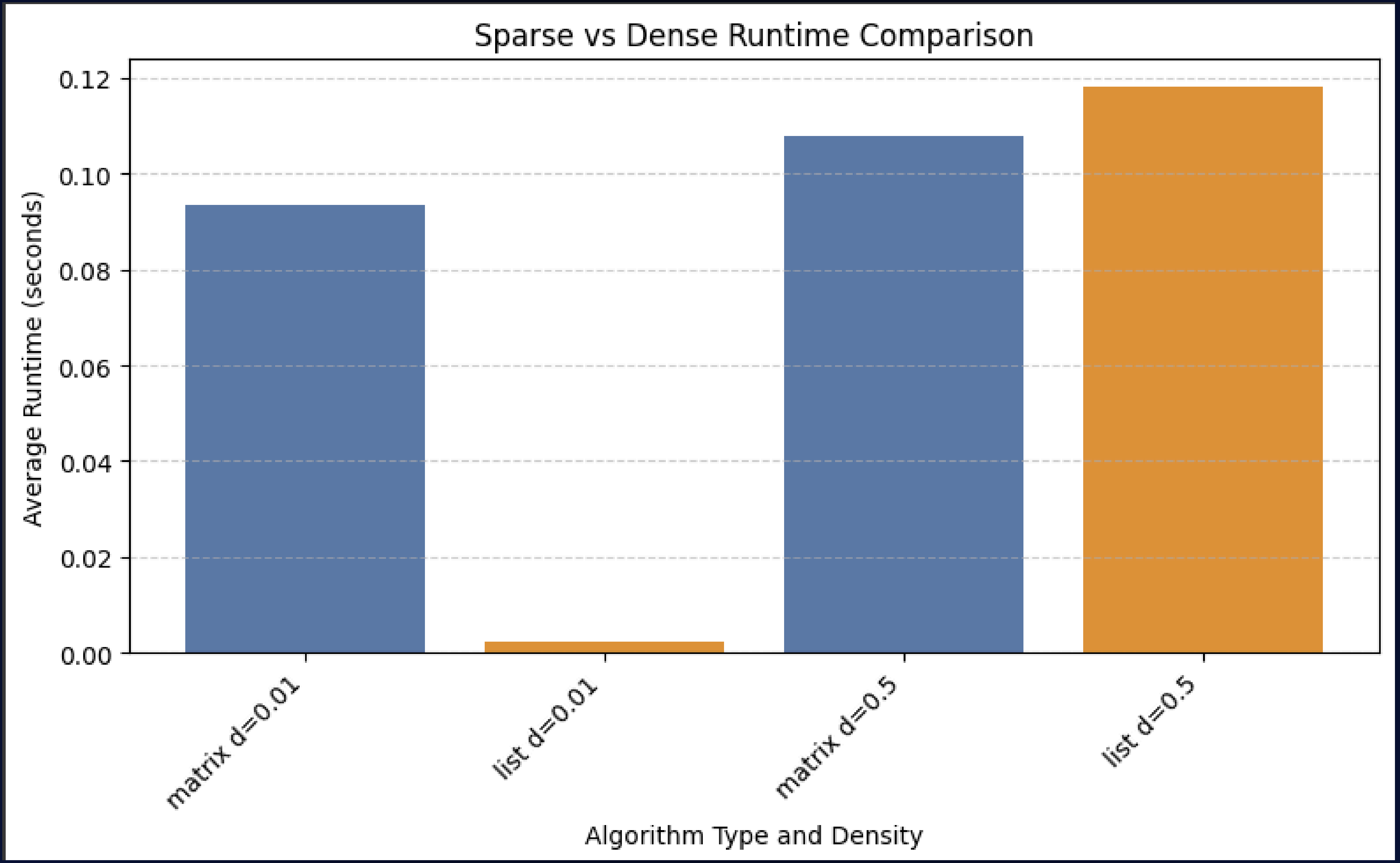


2. Varying Vertex with different densities

	V	E	density	representation	time_sec
0	100	99	0.01	matrix	0.000161
1	100	99	0.01	list	0.000009
2	100	4950	0.50	matrix	0.000942
3	100	4950	0.50	list	0.000449
4	200	398	0.01	matrix	0.002736
5	200	398	0.01	list	0.000111
6	200	19900	0.50	matrix	0.004181
7	200	19900	0.50	list	0.001946
8	400	1596	0.01	matrix	0.013678
9	400	1596	0.01	list	0.000414
10	400	79800	0.50	matrix	0.019572
11	400	79800	0.50	list	0.013952
12	600	3594	0.01	matrix	0.033130
13	600	3594	0.01	list	0.000907
14	600	179700	0.50	matrix	0.039102
15	600	179700	0.50	list	0.037437
16	800	6392	0.01	matrix	0.060175
17	800	6392	0.01	list	0.001593
18	800	319600	0.50	matrix	0.068584
19	800	319600	0.50	list	0.078587
20	1000	9990	0.01	matrix	0.109641
21	1000	9990	0.01	list	0.004921
22	1000	499500	0.50	matrix	0.187451
23	1000	499500	0.50	list	0.126385
24	1200	14388	0.01	matrix	0.135536
25	1200	14388	0.01	list	0.003549
26	1200	719400	0.50	matrix	0.159086
27	1200	719400	0.50	list	0.190943



SPARSE VS DENSE RUNTIME COMPARISON



ANALYSIS & DISCUSSION

Theoretical Comparison

- Matrix + Array: $O(V^2)$
- List + Heap: $O((V + E) \log V)$

Empirical Observations

- Contrary to theoretical expectations, List + Heap consistently outperforms Matrix + Array across a wide range of graph densities.
- Matrix + Array in practice, we see a dependency on |E| due to the overhead of the loops themselves, it only becomes competitive at extremely dense graphs

Explanation

- Theoretical analysis does not account for constant factors and implementation overhead.
- In Python:
  - heapq operations are highly optimized.
  - Adjacency lists provide better cache locality and avoid unnecessary iteration over empty edges.
- These practical considerations shift the crossover point to much higher densities than theory predicts.

Insight

- For real-world Python applications, List + Heap is generally the more efficient and scalable choice, even in cases where Matrix-based implementations appear favorable in theory.

CONCLUSION

- The performance of Dijkstra’s algorithm is strongly influenced by graph representation and priority queue structure.
- Theoretically, Matrix + Array runs in $O(V^2)$ and List + Heap in $O((V + E) \log V)$.
- Empirically, the List + Heap implementation maintains its advantage across almost all densities, outperforming the Matrix + Array approach even in moderately dense graphs.
- This is because, in practice, Python’s optimized heap operations and the cache efficiency of adjacency lists outweigh the theoretical benefits of matrix-based edge access.
- The crossover point where Matrix + Array might outperform List + Heap occurs only in extremely dense graphs, which are rare in real-world scenarios.
- Therefore, for practical Python applications and large-scale graphs such as road or network data, List + Heap is the consistently better and more scalable choice.