**Adj list vs Ajd matrix**

Example graph

Graph with 3 vertices: A, B, C

Edges:

* A → B, weight 5
* A → C, weight 2
* B → C, weight 1

1. Adjacency list

E = {

'A': [('B', 5), ('C', 2)],

'B': [('C', 1)],

'C': []

}

* Here, for each vertex, we only store the neighbors that exist and their weights.
* for v, w in E[u]: will only loop over existing edges.

2. Adjacency matrix

V = ['A', 'B', 'C']

A = [

[0, 5, 2], # A → A=0, A → B=5, A → C=2

[float('inf'), 0, 1], # B → A=∞, B → B=0, B → C=1

[float('inf'), float('inf'), 0] # C → no outgoing edges

]

* Here, each row corresponds to a vertex, and each column represents a possible edge to every vertex.
* You would loop over all columns:
* for v in range(len(V)):
* if A[u][v] != float('inf'):
* # relax edge
* Even if most entries are ∞ (no edge), you still check every vertex.

Key difference:

| Aspect | Adjacency List | Adjacency Matrix |
| --- | --- | --- |
| Storage | Only store existing edges | Store all possible edges |
| Looping | Only neighbors of u | All vertices v = 0..V-1 |
| Sparse graph | Very efficient | Inefficient (many ∞ entries) |

**Array-based priority queue vs minimizing heap**

**1. Array-based priority queue**

* **How it works:**
  + Maintain an array (or list) of tentative distances dist[].
  + To pick the next vertex, **scan the array for the minimum distance among unvisited vertices**.
  + No special data structure is used; it’s just linear search.
* **Time complexity:**
  + Picking minimum vertex: O(V) per iteration → total O(V²)
  + Relaxing neighbors:
    - For adjacency matrix: O(V²) because you scan all vertices for neighbors.
    - For adjacency list: O(E), but picking min is still O(V²) in worst case
  + **Overall:** O(V²) (matrix)
* **Space complexity:** O(V) for distances + O(V²) for adjacency matrix
* **Pros:** Simple to implement; works fine for **small or dense graphs**.
* **Cons:** Slow for **large or sparse graphs** because scanning the whole array every time is inefficient.

**2. Min-heap (binary heap) priority queue**

* **How it works:**
  + Maintain a **min-heap** of (distance, vertex) pairs.
  + Extract the vertex with the minimum distance in O(log V).
  + When distances are updated during relaxation, update the heap in O(log V).
* **Time complexity:**
  + Extract-min: O(log V) × V = O(V log V)
  + Relax edges and push updates: O(log V) × E = O(E log V)
  + **Overall:** O((V + E) log V)
* **Space complexity:** O(V) for heap + O(V + E) for adjacency list
* **Pros:** Much faster for **large, sparse graphs**; scales better with more vertices and edges.
* **Cons:** Slightly more complex to implement; heap overhead may dominate for tiny graphs.

**3. Key differences**

| **Aspect** | **Array PQ** | **Min-Heap PQ** |
| --- | --- | --- |
| Picking min | Linear scan O(V) | Heap O(log V) |
| Updating distances | Simple assignment | Heap push/update O(log V) |
| Time complexity | O(V²) | O((V + E) log V) |
| Best for | Small or dense graphs | Large or sparse graphs |
| Implementation | Simple | Slightly more complex |

**Summary:**

* Use **array** when the graph is small or dense (V not too large).
* Use **min-heap** when the graph is large or sparse (E << V²).
* Min-heap **asymptotically dominates** array-based PQ in almost all practical large graphs.