# Findings

## 1. Data Structures: Time & Space Complexity

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data Structure | Operations | Time Complexity | Space Complexity | Notes / Trade-offs |
| DynamicVertexArray | • findVertexIndexLinear(id) • pushBack(Vertex\*) • removeAt(idx) • sortVerticesQuick() • sortVerticesMerge() • findVertexIndexBinary(id) | • findVertexIndexLinear: O(n) • pushBack: Amortized O(1) • removeAt: O(n) • sortVerticesQuick: Average O(n log n), Worst O(n²) • sortVerticesMerge: O(n log n) • findVertexIndexBinary: O(log n) | • O(n) (array of n pointers) • QuickSort: O(log n) extra for recursion • MergeSort: O(n) extra | • Resizing amortized: O(1) • QuickSort: In-place but worst-case O(n²) • MergeSort: Guaranteed O(n log n) but uses extra space • Binary search requires sorted array |
| DynamicEdgeArray | • pushBack(Edge) • findEdgeIndex(dest) • removeAt(idx) | • pushBack: Amortized O(1) • findEdgeIndex: O(m) • removeAt: O(m) | • O(m) (array of m Edge objects) | • Resizable adjacency list • Good for sparse graphs where m is small • Shifts tail on removal |
| MinHeap | • insertKey(v, dist) • decreaseKey(v, newDist) • extractMin() • isInMinHeap(v) | • insertKey: O(log V) • decreaseKey: O(log V) • extractMin: O(log V) • isInMinHeap: O(1) | • O(V) (array of V MinHeapNode + pos[] of size V) | • Standard binary heap for priority queue • Supports efficient decrease-key • pos[] allows O(1) position lookup |

## 2. Comparative Analysis: Sorting & Searching Algorithms

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algorithm | Category | Time Complexity | Space Complexity | Stable? | When to Use / Trade-offs |
| QuickSort | Sorting | Average: O(n log n) Worst: O(n²) | O(log n) (recursion) | No | • In-place, low extra memory • Fast on average • Worst-case can degrade unless pivot optimized |
| MergeSort | Sorting | O(n log n) (avg and worst) | O(n) (temporary arrays) | Yes | • Guaranteed O(n log n) • Uses extra space • Stable sort |
| Linear Search | Searching | O(n) | O(1) | N/A | • No preprocessing required • Good for small or unsorted arrays • Inefficient for large n |
| Binary Search | Searching | O(log n) | O(1) | N/A | • Requires sorted array • O(n log n) to sort once, then O(log n) per lookup • Efficient for multiple searches |

### Justification of Sorting and Searching Choices

#### Why Two Sorting Algorithms?

**QuickSort:**Pros:  
In‐place, minimal auxiliary memory (only recursion stack).  
Excellent average‐case performance on real‐world data.  
Simple to implement.

Cons:  
Worst‐case O(n²) if input is already sorted or all elements are identical.I  
n our CLI, QuickSort is offered as a “fast” sort—users see sorted IDs quickly most of the time.

**MergeSort:**Pros:  
Guaranteed O(n log n) regardless of input.  
Stable (though not needed for unique IDs, but valuable if extending to more complex data).

Cons:  
Requires O(n) extra space for the merging process.  
We provide MergeSort so the application has a “predictable” sorting routine. If you care about worst‐case bounds (e.g., processing extremely large vertex lists where QuickSort’s worst‐case is unacceptable), you can switch to MergeSort.

Why Two Searching Algorithms?  
**Linear Search:**  
Use when the array is small or unsorted, and you only need to perform a single lookup.  
No sorting/preprocessing overhead → O(1) extra space but O(n) time.

**Binary Search:**  
Use when you’ll perform multiple lookups over the same sorted set of IDs, or the size n is large enough that O(n) is prohibitive.  
Requires sorting first (O(n log n)), then each individual lookup is O(log n).I  
n our menu we automatically sort (via QuickSort) before doing a binary search, so the total time for one search becomes O(n log n + log n) ≈ O(n log n). Practically, if users intend to perform many searches in a session, it’s better to maintain a sorted array (e.g., sort once, then do multiple binarySearch calls).

## Overall Justifications and Trade-offs

### Why Use Adjacency Lists (DynamicEdgeArray) + DynamicVertexArray?

**Sparse vs. Dense**  
If |E| ≪ |V|² (which is typical for road networks, social networks, etc.), adjacency lists use O(V + E) space, whereas an adjacency matrix would use O(V²) regardless.

**CRUD Efficiency**  
Inserting a new vertex is amortized O(1). Deleting a vertex is O(V + E\_per\_vertex) in the worst case, due to index‐shifting. Since vertex deletions are relatively rare compared to running Dijkstra, this cost is acceptable.I  
nserting/updating/removing an edge from a vertex’s adjacency list is O(m) where m is that vertex’s degree. In sparse graphs, m is small.

**Flexibility**  
Each adjacency list can hold arbitrary metadata (we store only weight here, but could be extended to capacity, travel time, or color).  
Easy to implement and reason about in an OOP style.

### Why Use a Binary MinHeap for Dijkstra?

**Good Practical Performance**A binary heap’s O(log V) decreaseKey and extractMin are fast, small‐constant‐factor, and easy to code.

**Alternatives & Why They Weren’t Chosen**Fibonacci Heap:  
Pros: Amortized O(1) decreaseKey, O(log V) extractMin → can reduce Dijkstra to O(E + V log V).  
Cons: Very complex to implement correctly, with large hidden constants; not worth it unless |E| is extremely large (≫ 10⁶).

Van Emde Boas or Pairing Heaps:  
Also more complex, require specialized libraries or advanced code.

For most mid-sized graphs (V up to ~10⁴, E up to ~10⁵), a binary MinHeap gives excellent empirical performance.

### Why Include Both QuickSort and MergeSort?

Demonstrates mastery of two fundamental divide-and-conquer sorting approaches.

Allows the user to choose “fast in practice” (QuickSort) versus “guaranteed worst-case” (MergeSort).

Even though sorting vertex IDs is not critical to shortest paths, it fulfills the project requirement of implementing two distinct sorting algorithms.

### Why Include Both Linear and Binary Search?

Linear Search is trivial to implement and works on unsorted data. Including it shows the most basic search method.

Binary Search highlights how sorting can accelerate repeated lookups.

Together, they demonstrate trade-offs: no preprocessing vs. O(n log n) preprocessing + O(log n) lookup.

## Summary

### Space Complexity

Adjacency lists + dynamic arrays use O(V + E) total memory—ideal for sparse graphs.  
MinHeap uses O(V) extra overhead (in addition to graph).  
MergeSort uses O(n) extra scratch space; QuickSort uses only O(log n).

### Time Complexity

**Graph CRUD:**Add Vertex: amortized O(1).  
Remove Vertex: O(V + total adjacency traversals) → O(V + E\_adj).  
Add/Update/Remove Edge: O(m) where m is degree of source.

**Dijkstra (with MinHeap):**O((V + E) log V) in total. Each of the V vertices is extracted once (O(log V)), and each edge relaxation (up to E times) may involve a decreaseKey (also O(log V)).

**Sorting / Searching:**QuickSort: O(n log n) average, O(n²) worst. MergeSort: O(n log n) always.  
Linear Search: O(n). Binary Search: O(log n) (after O(n log n) sort).

By choosing dynamic arrays for adjacency lists, a binary MinHeap for Dijkstra, and implementing two sorting/searching routines, we strike a balance between simplicity, performance, and educational value—showcasing multiple algorithmic paradigms while ensuring the core shortest‐path functionality stays efficient for typical mid-sized inputs.