JAYPEE INSTITUTE OF INFORMATION TECHNOLOGY, NOIDA
B.Tech VI Semester
Non- Linear Data Structures Project Report

Comparative Performance Analysis of Naive vs Optimized Algorithms in Data Structures

Submitted By:

S. No.	Name	Enrollment no.	Batch
1	Aditya Patil	22102174	A7
2	Smriti Aggarwal	22102178	A7
3	Abhishek Mittal	22102160	A7

Abstract

This project aims to compare the efficiency of naive and optimized algorithms in solving fundamental computer science problems: shortest path finding, pattern matching, and disjoint set union operations. These problems are foundational in domains such as networking, compilers, search engines, and operating systems.

We implemented and compared:

- Dijkstra's Algorithm: using both priority_queue and set (to analyze the impact of data structures).
- Pattern Matching: using a naive trie-based approach and an optimized approach using KMP combined with Trie.
- Disjoint Set Union: comparing naive union-find with path compression and union by rank optimization.

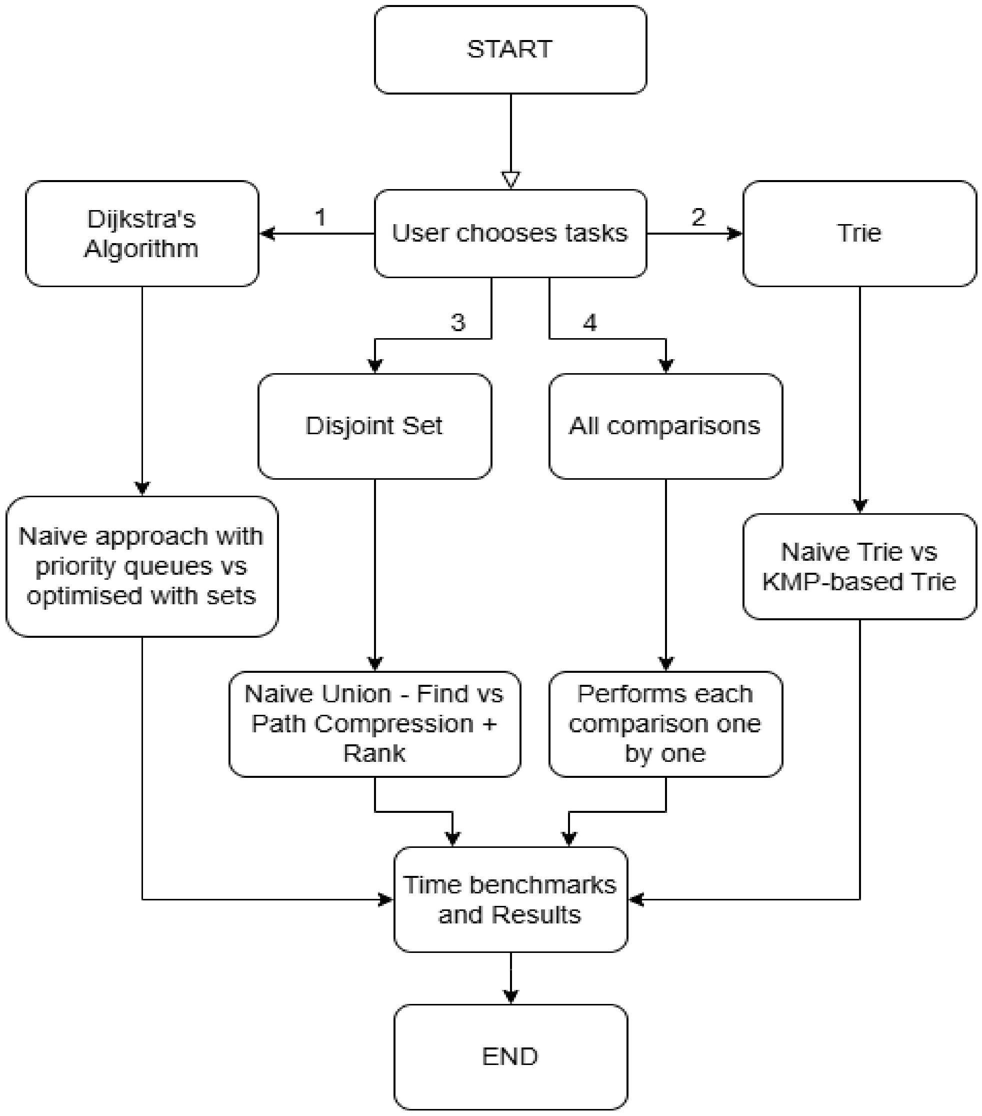
Through this project, we not only reinforced our understanding of data structures like heaps, tries, graphs, and sets, but also gained insights into algorithm complexity analysis, space-time tradeoffs, and real-world problem modeling.
Extensive benchmarking was done using randomized inputs to measure average and total execution time of each approach. The results demonstrated a consistent performance gain with optimized methods, validating their practical relevance.
This project combines theoretical concepts from our Non-Linear Data Structures (NLDS) course and applies them in a comparative, experimental setting.

Topics of NLDS Used

1. **Graphs and Shortest Path Algorithms**
    a. Dijkstra's Algorithm: Implemented to find the shortest paths from a single source node to all other nodes in a weighted graph with non-negative edge weights. It's widely used in routing and navigation systems.
    b. Graph Representation using Adjacency List: A memory-efficient way to represent sparse graphs. Each node maintains a list of its adjacent nodes, which helps in performing efficient graph traversals like BFS and DFS.
2. **Trie Data Structure**
    a. Naive Trie Construction: Built a basic trie for storing a set of strings, allowing fast lookup, insertion, and prefix-based searches.
    b. Trie Combined with KMP for Pattern Matching: Enhanced the trie structure by integrating the Knuth-Morris-Pratt (KMP) algorithm to improve string searching efficiency. This combination helps in applications like autocomplete and DNA sequence analysis.
3. **Disjoint Set Union-Find**
    a. Naive Approach: Implemented basic union and find operations using parent pointers without optimizations.
    b. Optimized Approach: Improved the performance using:
        i. Path Compression: Flattens the structure of the tree whenever find() is called, making future operations faster.
        ii. Union by Rank: Ensures the smaller tree is always attached under the root of the larger tree, maintaining a balanced structure.

4. **Priority Queue (Min-Heap) & Set STL Container**
    a. Used Min-Heap based priority queues to efficiently fetch the minimum element, especially useful in Dijkstra's Algorithm and task scheduling scenarios.
    b. Utilized the Set STL Container in C++ for storing unique elements in a sorted order and achieving fast insertions, deletions, and lookups.
5. **Algorithm Complexity Analysis**
    a. Performed time and space complexity analysis for each algorithm and data structure to evaluate performance and scalability.
    b. Compared naive vs optimized implementations to understand trade-offs in real-time systems.
6. **Efficient String Matching (KMP Algorithm)**
    a. Used the KMP Algorithm to perform linear-time string matching, which avoids unnecessary comparisons by preprocessing the pattern into a partial match table.
    b. Applied in scenarios like searching substrings within large texts, spam filtering, and data validation.

Flowchart / Design of the Project


START
User chooses tasks
1
2
3
4
Dijkstra's Algorithm
Trie
Disjoint Set
All comparisons
Naive approach with priority queues vs optimised with sets
Naive Trie vs KMP-based Trie
Naive Union - Find vs Path Compression + Rank
Performs each comparison one by one
Time benchmarks and Results
END

Implementation Details

The codebase has been structured into three key modules and a driver module:

1. Dijkstra's Algorithm (Graph.h / Graph.cpp) by Aditya Patil (22102174)

- NaiveGraph uses priority_queue without optimization.
- Graph uses set for optimal edge relaxation.
- Implemented with:
    - Adjacency list representation.
    - Min-heap behavior using STL.
    - Multiple test cases and execution loops.

Code:

```cpp
// ===================== Dijkstra.h =====================
#ifndef DIJKSTRA_H
#define DIJKSTRA_H
#include <vector>
#include <queue>
#include <set>
#include <climits>
using namespace std;
class NaiveGraph {
public:
    int V;
    vector<vector<pair<int, int>>> adj;
    NaiveGraph(int V);
    void addEdge(int u, int v, int w);
    vector<int> dijkstra(int src);
};

class Graph {
public:
    int V;
    vector<vector<pair<int, int>>> adj;
    Graph(int V);
    void addEdge(int u, int v, int w);
    vector<int> dijkstra(int src);
};
NaiveGraph::NaiveGraph(int V) : V(V) { adj.resize(V); }
void NaiveGraph::addEdge(int u, int v, int w) {
    adj[u].emplace_back(v, w);
    adj[v].emplace_back(u, w);
}
vector<int> NaiveGraph::dijkstra(int src) {
    vector<int> dist(V, INT_MAX);
    dist[src] = 0;
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
    pq.push({0, src});
    while (!pq.empty()) {
        int d = pq.top().first;
        int u = pq.top().second;
        pq.pop();
        if (d > dist[u]) continue;
        for (auto& edge : adj[u]) {
            int v = edge.first, w = edge.second;
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                pq.push({dist[v], v});
            }
        }
    }
    return dist;
}

Graph::Graph(int V) : V(V) { adj.resize(V); }
void Graph::addEdge(int u, int v, int w) {
    adj[u].emplace_back(v, w);
    adj[v].emplace_back(u, w);
}
vector<int> Graph::dijkstra(int src) {
    vector<int> dist(V, INT_MAX);
    dist[src] = 0;
    set<pair<int, int>> pq;
    pq.insert({0, src});
    while (!pq.empty()) {
        int u = pq.begin()->second;
        pq.erase(pq.begin());
        for (auto& edge : adj[u]) {
            int v = edge.first, w = edge.second;
            if (dist[u] + w < dist[v]) {
                pq.erase({dist[v], v});
                dist[v] = dist[u] + w;
                pq.insert({dist[v], v});
            }
        }
    }
    return dist;
}
#endif
```

2. Pattern Matching (KMPTrie.h / KMPTrie.cpp) by Smriti Aggarwal (22102178)

- NaiveTrie: Standard trie-based word matching.
- KMPTrie: Enhances the trie with Knuth-Morris-Pratt preprocessing.
- Multiple test strings and patterns inserted and searched repeatedly.

Code:

```
// ==================== KMPTrie.h ====================

#ifndef KMPTRIE_H
#define KMPTRIE_H
#include <unordered_map>
#include <vector>
#include <string>
using namespace std;
struct TrieNode {
    unordered_map<char, TrieNode*> children;
    bool isEnd = false;
};

class NaiveTrie {
    TrieNode* root;
public:
    NaiveTrie();
    void insert(const string& word);
    vector<string> searchInText(const string& text);
};
```

```cpp
class KMPTrie {
    TrieNode* root;
    unordered_map<string, vector<int>> lpsMap;
public:
    KMPTrie();
    void insert(const string& word);
    vector<string> searchInText(const string& text);
private:
    vector<int> computeLPS(const string& pattern);
};
NaiveTrie::NaiveTrie() { root = new TrieNode(); }
void NaiveTrie::insert(const string& word) {
    TrieNode* node = root;
    for (char c : word) {
        if (!node->children[c]) node->children[c] = new TrieNode();
        node = node->children[c];
    }
    node->isEnd = true;
}
vector<string> NaiveTrie::searchInText(const string& text) {
    vector<string> found;
    for (int i = 0; i < text.size(); ++i) {
        TrieNode* node = root;
        string word;
        for (int j = i; j < text.size(); ++j) {
            char c = text[j];
            if (!node->children.count(c)) break;
            word += c;
            node = node->children[c];
            if (node->isEnd) found.push_back(word);
        }
    }
    return found;
}

KMPTrie::KMPTrie() { root = new TrieNode(); }
void KMPTrie::insert(const string& word) {
    TrieNode* node = root;
    for (char c : word) {
        if (!node->children[c]) node->children[c] = new TrieNode();
        node = node->children[c];
    }
    node->isEnd = true;
    lpsMap[word] = computeLPS(word);
}
vector<string> KMPTrie::searchInText(const string& text) {
    vector<string> found;
    for (auto it = lpsMap.begin(); it != lpsMap.end(); ++it) {
        string pattern = it->first;
        vector<int> lps = it->second;
        int m = pattern.size(), n = text.size();
        int i = 0, j = 0;
        while (i < n) {
            if (text[i] == pattern[j]) {
                i++; j++;
                if (j == m) {
                    found.push_back(pattern);
                    j = lps[j - 1];
                }
            } else if (j > 0) {
                j = lps[j - 1];
            } else {
                i++;
            }
        }
    }
    return found;
}
vector<int> KMPTrie::computeLPS(const string& pattern) {
    vector<int> lps(pattern.size(), 0);
    int len = 0;
    for (int i = 1; i < pattern.size(); ) {
        if (pattern[i] == pattern[len]) {
            lps[i++] = ++len;
        } else if (len) {
            len = lps[len - 1];
        } else {
            lps[i++] = 0;
        }
    }
    return lps;
}
#endif
```

3. Disjoint Set (DisjointSet.h / DisjointSet.cpp) by Abhishek Mittal (22102160)

- NaiveDisjointSet: Union without optimizations.
- DisjointSet: Path compression and union by rank.
- Includes batch unite and find operations.

Code:

```
// ==================== DisjointSet.h ====================

#ifndef DISJOINTSET_H
#define DISJOINTSET_H
#include <vector>
using namespace std;
class NaiveDisjointSet {
    vector<int> parent;
public:
    NaiveDisjointSet(int n);
    int find(int x);
    void unite(int x, int y);
};

class DisjointSet {
    vector<int> parent, rank;
public:
    DisjointSet(int n);
    int find(int x);
    void unite(int x, int y);
};
NaiveDisjointSet::NaiveDisjointSet(int n) : parent(n) {
    for (int i = 0; i < n; ++i) parent[i] = i;
}
int NaiveDisjointSet::find(int x) {
    while (x != parent[x]) x = parent[x];
    return x;
}
void NaiveDisjointSet::unite(int x, int y) {
    int rootX = find(x), rootY = find(y);
    if (rootX != rootY) parent[rootY] = rootX;
}

DisjointSet::DisjointSet(int n) : parent(n), rank(n, 0) {
    for (int i = 0; i < n; ++i) parent[i] = i;
}
int DisjointSet::find(int x) {
    if (x != parent[x]) parent[x] = find(parent[x]);
    return parent[x];
}
void DisjointSet::unite(int x, int y) {
    int rootX = find(x), rootY = find(y);
    if (rootX == rootY) return;
    if (rank[rootX] < rank[rootY]) {
        parent[rootX] = rootY;
    } else if (rank[rootX] > rank[rootY]) {
        parent[rootY] = rootX;
    } else {
        parent[rootY] = rootX;
        rank[rootX]++;
    }
}
#endif
```

4. Main Driver File

- Menu-driven interface for:
  - Independent module testing.
  - Full comparison run (Run All Comparisons).
  - Execution time benchmarking using chrono library.

Code:

```
// ==================== main.cpp ====================
#include <iostream>
#include <chrono>
#include <cstdlib>
#include "Dijkstra.h"
#include "KMPTrie.h"
#include "DisjointSet.h"
using namespace std;
int main() {
    while (true) {
        cout << "\nChoose an algorithm to run:\n";
        cout << "1. Dijkstra's Algorithm (Naive vs Optimized)\n";
        cout << "2. Pattern Search (Naive Trie vs KMP + Trie)\n";
        cout << "3. Union-Find (Naive vs Optimized)\n";
        cout << "4. Run All Comparisons\n";
        cout << "0. Exit\n";
        cout << "Enter choice: ";

        int choice;
        cin >> choice;
        if (choice == 0) break;

        if (choice == 1 || choice == 4) {
            cout << "\n[Running Dijkstra Comparison]\n";
            const int N = 1000;
```

```cpp
            NaiveGraph ng(N);
            Graph og(N);
            for (int i = 0; i < N - 1; ++i) {
                int w = 1 + rand() % 10;
                ng.addEdge(i, i + 1, w);
                og.addEdge(i, i + 1, w);
            }
            for (int i = 0; i < 10000; ++i) {
                int u = rand() % N, v = rand() % N;
                if (u != v) {
                    int w = 1 + rand() % 10;
                    ng.addEdge(u, v, w);
                    og.addEdge(u, v, w);
                }
            }
            ng.dijkstra(0);
            og.dijkstra(0);
            auto start = chrono::high_resolution_clock::now();
            for (int i = 0; i < 10; ++i) ng.dijkstra(0);
            auto end = chrono::high_resolution_clock::now();
            auto t1 = chrono::duration_cast<chrono::milliseconds>(end -
start).count();
            start = chrono::high_resolution_clock::now();
            for (int i = 0; i < 10; ++i) og.dijkstra(0);
            end = chrono::high_resolution_clock::now();
            auto t2 = chrono::duration_cast<chrono::milliseconds>(end -
start).count();
            cout << "Naive:      (avg over 10 runs): " << t1 / 10.0 << " ms\n";
            cout << "Optimized:  (avg over 10 runs): " << t2 / 10.0 << " ms\n";
        }

        if (choice == 2 || choice == 4) {
            cout << "\n[Running Pattern Search Comparison]\n";
            string text(10000, 'a');
            for (int i = 1000; i < 1100; ++i) text[i] = "catbatrat"[rand() % 9];
            vector<string> words = {"cat", "bat", "rat", "aaa", "aab"};
            NaiveTrie nt;
            KMPTrie kt;
            for (const auto& w : words) {
                nt.insert(w);
                kt.insert(w);
            }
            nt.searchInText(text);
            kt.searchInText(text);
            auto start = chrono::high_resolution_clock::now();
            for (int i = 0; i < 100; ++i) nt.searchInText(text);
            auto end = chrono::high_resolution_clock::now();
            auto t1 = chrono::duration_cast<chrono::milliseconds>(end -
start).count();
            start = chrono::high_resolution_clock::now();
            for (int i = 0; i < 100; ++i) kt.searchInText(text);
            end = chrono::high_resolution_clock::now();
            auto t2 = chrono::duration_cast<chrono::milliseconds>(end -
start).count();
            cout << "Naive:      (avg over 100 runs): " << t1 / 100.0 << " ms\n";
            cout << "Optimized:  (avg over 100 runs): " << t2 / 100.0 << " ms\n";
        }

        if (choice == 3 || choice == 4) {
            cout << "\n[Running Union-Find Comparison]\n";
            const int N = 1000;
            NaiveDisjointSet nds(N);
            DisjointSet ds(N);
            vector<pair<int, int>> ops;
            for (int i = 0; i < 10000; ++i) {
                int u = rand() % N, v = rand() % N;
                if (u != v) ops.emplace_back(u, v);
            }
            auto start = chrono::high_resolution_clock::now();
            for (const auto& p : ops) nds.unite(p.first, p.second);
            for (int i = 0; i < N; ++i) nds.find(i);
            auto end = chrono::high_resolution_clock::now();
            auto t1 = chrono::duration_cast<chrono::microseconds>(end -
start).count();
            start = chrono::high_resolution_clock::now();
            for (const auto& p : ops) ds.unite(p.first, p.second);
            for (int i = 0; i < N; ++i) ds.find(i);
            end = chrono::high_resolution_clock::now();
            auto t2 = chrono::duration_cast<chrono::microseconds>(end -
start).count();
            cout << "Naive:      (total): " << t1 / 1000.0 << " ms\n";
            cout << "Optimized:  (total): " << t2 / 1000.0 << " ms\n";
        }
    }
    return 0;
}
```

References

- CLRS (Introduction to Algorithms) – T. Cormen et al. – Dijkstra, Union-Find, KMP
- GeeksforGeeks.org – KMP Algorithm and Graph Shortest Path
- cplusplus.com / cppreference.com – STL priority_queue, set, unordered_map
- NLDS Course Lectures and Class Notes