DESIGN ANALYSIS AND ALGORITHM

A Comprehensive Notes

DISTRIBUTED ALGORITHMS AND MULTITHREADING

Distributed Algorithm

A **Distributed Algorithm** is an algorithm designed to run on a network of interconnected computers (or processors) that **do not share memory** but **communicate by passing messages**. These algorithms are used to solve problems **cooperatively**, with each node performing part of the computation.

🧠 **Examples:**

- **Distributed Breadth-First Search (BFS)**
- **Distributed Minimum Spanning Tree (MST)**

The Distributed BFS Algorithm

The algorithm works by starting at a random vertex and then doing a Breadth First Search (BFS) until all vertices have been visited. The order in which the vertices are visited is important, as it allows the algorithm to find the shortest paths between any two vertices.

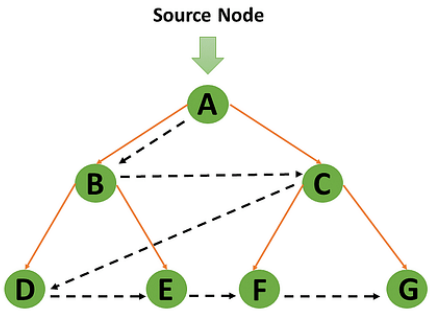
🔄 **Algorithm Steps:**

1. **Assign IDs**:
   Every node(processor) in the network is given a **unique ID**.
2. **Initiation by Root**:
   A **random node** is chosen as the **root** and starts the BFS by **requesting IDs** from its neighboring nodes.
3. **Tree Construction**:
   The root node uses the received IDs to **compute the shortest paths** to all nodes and builds a **BFS tree**.
4. **Broadcasting the Tree**:
   The root broadcasts the constructed tree to all nodes in the network.
5. **Subtree Identification**:
   Each node, upon receiving the tree, sets its **own ID as the parent** of all nodes in its **subtree**.

6. **Completion**:
   The process continues **until all nodes are visited** and the BFS tree is fully constructed.

🎯 Applications:

- Finding **shortest paths**
- **Community detection** in social networks (e.g., Facebook, Twitter)
- **Routing and task scheduling** in distributed systems



✅ Advantages:

- **Simple** to implement
- **Time and space efficient**
- Works well on **very large graphs**
- Better utilization of resources

❌ Disadvantages of Distributed BFS Algorithm:

1. **High Communication Overhead**:
   Frequent message passing between nodes can lead to network congestion.
2. **Limited to Undirected and Connected Graphs**:
   It does not perform well or may fail on disconnected or directed graphs.
3. **Not Suitable for Real-Time Processing**:
   It can be slow to converge, especially on very large graphs.
4. **Inefficient for Large Data Loads**:
   Processing large amounts of data via message passing is time-consuming.

Distributed Minimum Spanning Tree (DMST)

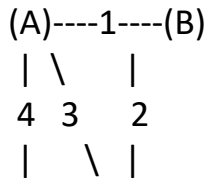
A **DMST** is the distributed computation of a **Minimum Spanning Tree** in a weighted, connected graph. Each processor (node) holds only a portion of the edges and cooperates via message passing to build the global MST.

Distributed Kruskal's Algorithm Steps

1. **Local Sort & MST**
    - Each processor sorts its local edges by weight.
    - Runs standard Kruskal on its subset to form a **local MST**.
2. **Global Min-Edge Election**
    - Each processor identifies its **lightest outgoing edge** (connects two different components).
    - Sends that edge to the **leader**.
3. **Leader Selection**
    - Leader picks the **global minimum** among received edges.
    - Broadcasts this edge to all processors.
4. **Component Merge**
    - Each processor uses Union-Find to **merge components** connected by the chosen edge.
    - Processors update their local MSTs, discarding any edges that now form cycles.
5. **Repeat**
    - Steps 2–4 continue until **all vertices** belong to a single component (global MST complete).

Example

Consider the following weighted graph:



(C)----5----(D)

Edges:
(A, B) = 1, (A, C) = 4, (A, D) = 3, (B, D) = 2, (C, D) = 5

Let's assume two processors handle the edges:

- **Processor 1:** (A, B), (A, C), (A, D)
- **Processor 2:** (B, D), (C, D)

Step-by-Step Execution

Step 1: Local Sorting

- **Processor 1:** (A, B) = 1, (A, D) = 3, (A, C) = 4
- **Processor 2:** (B, D) = 2, (C, D) = 5

Step 2: Local MST Construction

- **Processor 1 picks:** (A, B) = 1, (A, D) = 3
- **Processor 2 picks:** (B, D) = 2

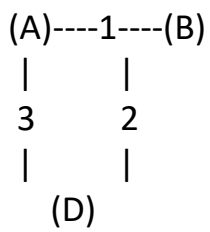
Local MSTs:

- **Processor 1:** (A, B), (A, D)
- **Processor 2:** (B, D)

Step 3: Communication and Merging

- The global leader compares edges and selects **(A, B) = 1** first.
- Then selects **(B, D) = 2** (since it doesn't form a cycle).
- Finally selects **(A, D) = 3**.
- **(C, D) = 5** and **(A, C) = 4** are discarded as they form cycles.

Final MST



MST Edges: **(A, B), (B, D), (A, D)**
Total Cost = **1 + 2 + 3 = 6**

Advantages of Distributed Kruskal's Algorithm

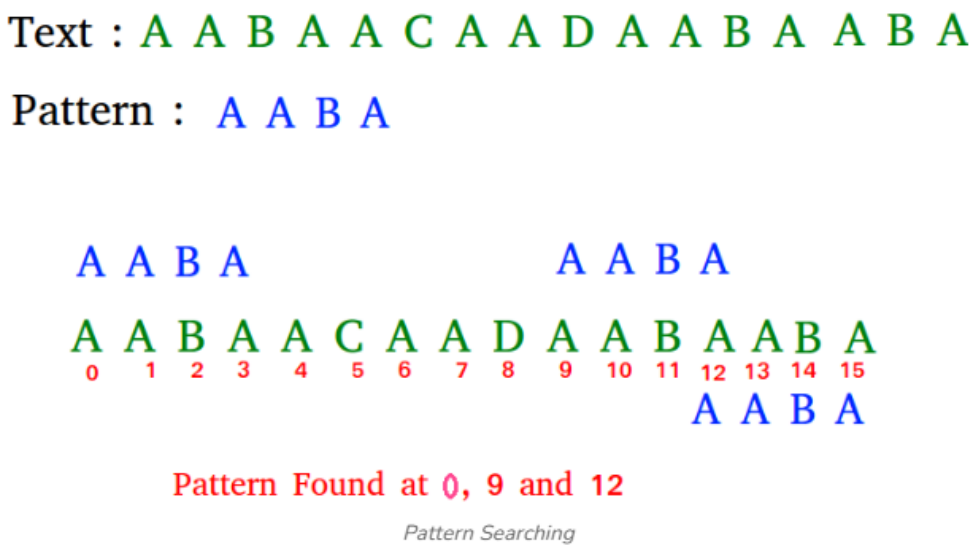
- **Parallelism**: Local work reduces overall runtime.
- **Scalability**: Handles very large graphs by distributing storage/computation.
- **Communication-Efficient**: Only lightest candidate edges are exchanged.

Disadvantages of Distributed Kruskal's Algorithm

- **Synchronicity & Coordination**: Requires reliable leader election and synchronized rounds.
- **Message Overhead**: Repeated broadcasts and component updates can be costly on high-latency networks.
- **Complexity**: Union-Find merging across distributed processors demands careful conflict resolution.

Pattern Searching

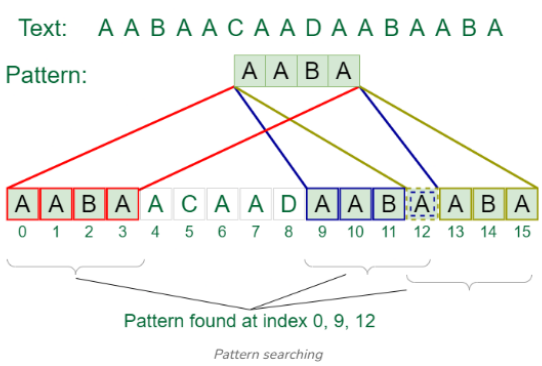
Pattern searching (string matching) is the task of finding all occurrences of a **pattern** P of length *m* inside a **text** *T* of length *n*. It underpins applications in text editors, search engines, data mining, and bioinformatics. plays a crucial role in tasks such as text processing, data retrieval, and computational biology



Pattern Searching

1. Naive Pattern Searching:

Naive pattern searching is a simple algorithm used to find all occurrences of a pattern (substring) within a text (string). The algorithm compares the pattern with the text character by character and moves one character at a time if a mismatch occurs.



Pattern searching

Time Complexity: O(n*m)

Algorithm

```
void search(string& pat, string& txt) {
    int M = pat.size();
    int N = txt.size();

    for (int i = 0; i <= N - M; i++) {
        int j;
        for (j = 0; j < M; j++) {
            if (txt[i + j] != pat[j]) {
                break;
            }
        }
        if (j == M) {
            cout << "Pattern found at index " << i << endl;
        }
    }
}

int main() {
    // Example 1
    string txt1 = "AABAACAADAABAABA";
```

```
    string pat1 = "AABA";
    cout << "Example 1: " << endl;
    search(pat1, txt1);
    return 0;
}
```

Output
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13

🧠 Complexity Analysis of Naive Pattern Searching Algorithm

Let:

- **n** = length of the text
- **m** = length of the pattern

✅ Best Case: O(n)

- Occurs when:
  - The **first character mismatch** happens immediately at each alignment, or
  - The **pattern is found at the beginning** of the text.
- Only a single comparison per alignment.
- Total comparisons ≈ O(n)

💡 Example best case:
Text: "ABCDE"
Pattern: "A"

❌ Worst Case: O(n × m)

- Occurs when:
  - The text contains repeated characters similar to the start of the pattern.

    - The pattern is not present, or appears only at the end.
- The algorithm checks **every position** in the text (i.e., n−m+1n - m + 1n−m+1 alignments), and for each position, may compare **all m characters**.
- $Total\ comparisons\ \approx\ O\big((n-m+1)\times m\big)\approx\ O(n\times m)$

💡 Example worst case:
txt = "AAAAAAAAAA"
pat = "AAAA"
Here, a lot of characters match before a mismatch, causing the algorithm to check repeatedly.

🧮 **Auxiliary Space: O(1)**

- The algorithm uses a constant amount of extra memory.

2. Rabin Karp Algorithm:

The Rabin-Karp algorithm is a string-searching algorithm that uses hashing to find patterns within a text efficiently. It was developed by Michael O. Rabin and Richard M. Karp. The key idea behind Rabin-Karp is to use a rolling hash function to quickly check for potential matches between the pattern and substrings of the text.

Time Complexity: O(n + m)

Like the Naive Algorithm, the Rabin-Karp algorithm also check every substring. But unlike the Naive algorithm, the Rabin Karp algorithm matches the hash value of the pattern with the hash value of the current substring of text, and if the hash values match then only it starts matching individual characters. So Rabin Karp algorithm needs to calculate hash values for the following strings.

How is Hash Value calculated in Rabin-Karp?
Hash value is used to efficiently check for potential matches between a pattern and substrings of a larger text. The hash value is calculated using a rolling hash function, which allows you to update the hash value for a new substring by efficiently removing the contribution of the old character and adding the contribution of the new character. This makes it possible to slide the pattern

over the text and calculate the hash value for each substring without recalculating the entire hash from scratch.

Step-by-step approach:

1. Initially calculate the hash value of the pattern.
2. Start iterating from the starting of the string:
3. Calculate the hash value of the current substring having length m.
4. If the hash value of the current substring and the pattern are same check if the substring is same as the pattern.
5. If they are same, store the starting index as a valid answer. Otherwise, continue for the next substrings.
6. Return the starting indices as the required answer.

Code:

```
void search(string pat, string txt, int q)
{
   int M = pat.size();
   int N = txt.size();
   int i, j;
   int p = 0; // hash value for pattern
   int t = 0; // hash value for txt
   int h = 1;
   int d = 256; // d is the number of characters in the input alphabet
   // The value of h would be "pow(d, M-1)%q"
   for (i = 0; i < M - 1; i++)
      h = (h * d) % q;

   // Calculate the hash value of pattern and first
   // window of text
   for (i = 0; i < M; i++) {
      p = (d * p + pat[i]) % q;
      t = (d * t + txt[i]) % q;
   }

   // Slide the pattern over text one by one
   for (i = 0; i <= N - M; i++) {
```

```
        // Check the hash values of current window of text
        // and pattern. If the hash values match then only
        // check for characters one by one
        if (p == t) {
            /* Check for characters one by one */
            for (j = 0; j < M; j++) {
                if (txt[i + j] != pat[j]) {
                    break;
                }
            }

            // if p == t and pat[0...M-1] = txt[i, i+1,
            // ...i+M-1]

            if (j == M)
                cout << "Pattern found at index " << i
                     << endl;
        }

        // Calculate hash value for next window of text:
        // Remove leading digit, add trailing digit
        if (i < N - M) {
            t = (d * (t - txt[i] * h) + txt[i + M]) % q;

            // We might get negative value of t, converting
            // it to positive
            if (t < 0)
                t = (t + q);
        }
    }
}

/* Driver code */
int main()
{
    string txt = "GEEKS FOR GEEKS";
    string pat = "GEEK";

    // we mod to avoid overflowing of value but we should
```

```
    // take as big q as possible to avoid the collison
    int q = INT_MAX;
    // Function Call
    search(pat, txt, q);
    return 0;
}

// This is code is contributed by rathbhupendra
```

Output
Pattern found at index 0
Pattern found at index 10

The average and best-case running time of the Rabin-Karp algorithm is O(n+m), but its worst-case time is O(nm).
The worst case of the Rabin-Karp algorithm occurs when all characters of pattern and text are the same as the hash values of all the substrings of T[] match with the hash value of P[].

Auxiliary Space: O(1)

Limitations of Rabin-Karp Algorithm
Spurious Hit: When the hash value of the pattern matches with the hash value of a window of the text but the window is not the actual pattern then it is called a spurious hit. Spurious hit increases the time complexity of the algorithm. In order to minimize spurious hit, we use good hash function. It greatly reduces the spurious hit

📝 Multithreading

Multithreading is a technique where a program creates multiple **threads** (lightweight subprocesses) to execute tasks concurrently. It allows better CPU utilization and faster execution for problems that can be divided into independent sub-tasks.

🔢 Multithreaded Matrix Multiplication

Problem: Multiply two matrices A and B using multiple threads to speed up computation.

✨ **CODE:**

```
#include <iostream>

#include <vector>

#include <thread>

using namespace std;


const int N = 3;

int A[N][N] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };

int B[N][N] = { {9, 8, 7}, {6, 5, 4}, {3, 2, 1} };

int C[N][N];  // Resultant matrix


void multiplyRow(int row) {

   for (int i = 0; i < N; i++) {

      C[row][i] = 0;

      for (int j = 0; j < N; j++) {

         C[row][i] += A[row][j] * B[j][i];

      }

   }

}
```

```
int main() {
    thread threads[N];

    // Launch N threads
    for (int i = 0; i < N; i++)
        threads[i] = thread(multiplyRow, i);

    for (int i = 0; i < N; i++)
        threads[i].join();

    // Output result
    cout << "Resultant Matrix C:\n";
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            cout << C[i][j] << " ";
        cout << endl;
    }
    return 0;
}
```

🕒 **Time Complexity:**

- **Sequential:** O(N^3)
- **Multithreaded:** O(N^3 / T) (ideal case, where T is the number of threads)

Multithreaded Quicksort

Idea: Split array into parts, sort each part using a separate thread.

```
#include <iostream>
#include <thread>
using namespace std;

void quicksort(int arr[], int left, int right) {
    if (left >= right) return;

    int pivot = arr[right], i = left - 1;
    for (int j = left; j < right; j++) {
        if (arr[j] < pivot)
            swap(arr[++i], arr[j]);
    }
    swap(arr[i + 1], arr[right]);

    int pi = i + 1;
    thread t1, t2;

    t1 = thread(quicksort, arr, left, pi - 1);
    t2 = thread(quicksort, arr, pi + 1, right);
```

```
    t1.join();
    t2.join();
}

int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr)/sizeof(arr[0]);

    quicksort(arr, 0, n - 1);

    cout << "Sorted array: ";
    for (int i = 0; i < n; i++) cout << arr[i] << " ";
    return 0;
}
```

🕒 **Time Complexity:**

- **Best/Average Case:** O(n log n / T)
- **Worst Case:** O(n^2) (if poor pivot is chosen and multithreading doesn't help much)

Multithreaded Mergesort

Idea: Divide array into halves and sort each half in a separate thread, then merge.

```
#include <iostream>
#include <thread>
```

```
using namespace std;

void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1, n2 = r - m;
    int L[n1], R[n2];
    for (int i = 0; i < n1; i++) L[i] = arr[l + i];
    for (int i = 0; i < n2; i++) R[i] = arr[m + 1 + i];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2)
        arr[k++] = (L[i] < R[j]) ? L[i++] : R[j++];

    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}

void mergesort(int arr[], int l, int r) {
    if (l >= r) return;

    int m = l + (r - l) / 2;

    thread t1(mergesort, arr, l, m);
```

```
    thread t2(mergesort, arr, m + 1, r);

    t1.join();
    t2.join();
    merge(arr, l, m, r);
}

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr)/sizeof(arr[0]);

    mergesort(arr, 0, n - 1);

    cout << "Sorted array: ";
    for (int i = 0; i < n; i++) cout << arr[i] << " ";
    return 0;
}
```

🕒 **Time Complexity:**

- **Best/Average/Worst Case:** O(n log n / T)
  (T = number of parallel threads used)

Race Condition

A **race condition** happens when:

- **Two or more threads** access shared data at the **same time**.
- At least **one thread modifies** the data.
- The **final result depends on the timing** of how threads are scheduled.