Exploring Efficient Techniques for Connected Component Analysis in Dense Graphs: Unveiling Practical Applications

Azmi Mahmud Bazeid - 13522109
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13522109@mahasiswa.itb.ac.id

***Abstract*—This study explores an efficient technique for analyzing connected components in dense graphs. The algorithm, developed by external researchers, addresses computational challenges inherent in traditional methods and significantly improves both time and space efficiency. Interestingly, the algorithm has found applications beyond its initial scope. These findings underscore the algorithm's versatility and its potential to solve complex problems across various domains.**

***Keywords*—Connected Components, Sorting, Time Complexity, Space Complexity, Applications.**

I. Introduction

Graphs, as abstract representations of a set of objects where some pairs of the objects are connected by links, serve as one of the fundamental structures in the field of computer science. They are used in various domains, from network analysis to social media analytics, and from route planning to even biology. The versatility of graphs stems from their ability to model complex systems and relationships in a simplified and intuitive manner.

One of the critical tasks in graph analysis is the study of connected components. A connected component of an undirected graph is a subgraph in which any two vertices are connected to each other by a path, and which is connected to no additional vertices in the supergraph. Identifying these components helps in understanding the structure of the graph, the relationship between its entities, and the overall connectivity of the system it represents.

To facilitate this analysis, the disjoint-set data structure is often employed. This data structure allows two crucial operations: Union and Find. The Union operation can join two sets so that each member in the set belongs to the same connected component. On the other hand, the Find operation returns the representative of the set that a particular element belongs to. These operations can be performed in approximately constant time (in $O(\alpha(n))$ where $\alpha(n)$ is the inverse Ackermann function which is less than 5 for most n needed since $A(4,4) = 2^{2^{2^{65536}}} - 3$) making the disjoint-set data structure a powerful tool for connectivity analysis.

However, the disjoint-set data structure has a significant limitation. While it efficiently records the connectivity information, it loses the edge relationship between the vertices of a graph. In other words, it can tell us if two vertices are connected, but it cannot provide information about the specific edges that connect them. This loss of information can be a major drawback in scenarios where edge relationships are important.

An alternative approach to preserving edge relationships is to use a basic graph traversal method like Depth-First Search (DFS) or Breadth-First Search (BFS) using any common graph representation. These methods explore the graph by visiting its vertices in a systematic manner, thereby preserving the edge relationships. However, these methods work in linear time relative to the number of nodes and edges. Repeatedly traversing the graph conditionally can become computationally expensive, especially for dense graphs where the number of edges is on the order of the square of the number of vertices.

In this paper, we explore a modified DFS algorithm designed to handle dense graphs with a large number of edges. Our goal is to retain the benefits of preserving edge relationships while improving computational efficiency for many common operations. This is particularly important in the era of big data, where graphs can have millions or even billions of vertices and an even larger number of edges.

The modified DFS algorithm we study in this paper is not just a theoretical construct but has practical implications as well. It opens up new possibilities for graph analysis in various domains, thereby bridging the gap between theory and practice. We hope that our exploration will inspire further research in this direction, leading to more efficient and effective methods for graph analysis.

It's important to note that, in contrast to the disjoint-set data structure, our algorithm of interest does not have a specific name. For the purpose of this study, we will refer to it as the 'Modified DFS Algorithm'.

We will use C++ to demonstrate the algorithms instead of pseudocode, providing a practical and accessible representation. This choice enhances clarity in logic and implementation steps, making it more understandable for readers with a programming background and bridging the gap between theory and practice.

Our exploration will commence with an in-depth study of the DFS algorithm, particularly its role in identifying connected components. Following this, we will conduct a comprehensive review of the time complexity associated with the DFS algorithm. This examination will provide the motivation to introduce slight modifications to both the data structure representation and the DFS algorithm itself. The aim of these modifications is to equip the algorithm with the capability to efficiently handle dense graphs.

II. THEORY

A. Depth-First Search (DFS)

Depth-First Search (DFS) is a crucial algorithm employed for investigating nodes and edges within a graph. It navigates the graph in a depth-oriented manner and utilizes a stack to remember the vertices to which it needs to return after all neighboring vertices have been traversed. Given that DFS operates with a stack, it can be seamlessly implemented using recursion in numerous programming languages, eliminating the need for an explicit stack declaration.

In this paper, we will be using the adjacency list representation of the graph to facilitate DFS traversal in $O(V + E)$ where V is the number of vertices and E is the number of of edges. For simplicity, we will assume that the vertices are numbered from 0 to $V - 1$. The C++ code is as follows:

```
/*
We assume that the graph is already provided as input to the
adjacency list 'adj'. Otherwise, global variables, by default,
have values of zero or are set to false.
*/
const int V = 1000;
vector<int> adj[V];
bool visited[V];

void dfs(int &node) {
    visited[node] = true;
    for (const auto &neighbour : adj[node])
        if (!visited[neighbour])
            dfs(neighbour);
}

void run() {
    for (int node = 0; node < V; ++node)
        if (!visited[node])
            dfs(node);
}
```

This is a basic Depth-First Search (DFS) that traverses each node in the graph. For simplicity, details are omitted. The code can be easily adjusted to count the number of connected components by introducing a counter in the inner loop of the 'run' procedure. Additionally, one can copy each connected component to a new graph, forming a graph that consists exclusively of that particular connected component.

DFS visits each vertex exactly once, and the visited boolean array ensures that no vertex is visited more than once, contributing $O(V)$ to the time complexity. As DFS traverses every edge, each edge is considered twice (once for each incident vertex). Therefore, the contribution from edge traversal is $O(2E)$, which simplifies to $O(E)$ in terms of overall time complexity. Hence, the total time complexity is $O(V + E)$.

In dense graphs, the maximum number of edges is $V(V - 1)/2$, leading to a time complexity of $O(V^2)$ for DFS traversal. While this poses no problem for single traversals, certain applications may require conditional graph traversal multiple times, resulting in potential slowness in performance.

In addition to traversal concerns, some applications may necessitate operations other than traversal, and these operations may not be efficiently supported using the traditional DFS approach. This highlights the importance of considering the specific requirements of the application and potentially exploring alternative algorithms or optimizations tailored to the particular tasks at hand.

B. Self-balancing Binary Search Tree

A self-balancing binary search tree (BST) is a type of binary search tree where the structure of the tree is automatically adjusted or balanced after each insertion or deletion operation. The goal of this balancing act is to maintain the tree in a way that ensures relatively uniform depths of subtrees, preventing the tree from becoming highly skewed and degrading into a linked list.

In a regular binary search tree, the time complexity of operations like insertion, deletion, and search is typically $O(h)$, where h is the height of the tree. In the worst case, when the tree is highly unbalanced, the height could be close to n, where n is the number of elements in the tree. This worst-case scenario would lead to operations taking $O(n)$ time, which defeats the purpose of using a binary search tree.

To address this issue, self-balancing binary search trees use algorithms that automatically maintain balance during insertions and deletions. One common type of self-balancing binary search tree is the red-black tree. In a red-black tree, each node is assigned a color (either red or black), and the tree is adjusted based on a set of rules that ensure its balance. These rules include properties like ensuring that no two consecutive red nodes exist on any path from the root to a leaf and maintaining the same number of black nodes on all paths from the root to the leaves.

The balancing operations are performed in such a way that the height of the tree is logarithmic in the number of elements, keeping the time complexity of operations like insertion, deletion, and search at $O(\log n)$, where n is the number of elements in the tree. This ensures that the self-balancing binary search tree maintains efficient performance even in the face of dynamic operations.

Note that it is possible to ensure the height is logarithmic in the number of elements since $2^0 + 2^1 + 2^2 + \cdots + 2^{\log_2 n} \approx n$. Table I shows the time complexity of Red-Black tree.

	Amortized	Worst Case
Search	$O(\log n)$	$O(\log n)$
Insert	$O(1)$	$O(\log n)$
Delete	$O(1)$	$O(\log n)$

Table I. Time complexity of Red-Black Tree

We will use red-black tree in the modified DFS algorithm as well as the data structure used to represent the graph. In the modified DFS algorithm, we have incorporated the red-black tree strategically within the data structure used for graph representation.

The red-black tree plays a pivotal role in maintaining the structural integrity of the graph representation. With its self-balancing characteristics, such as logarithmic height and balanced node distribution, the red-black tree significantly optimizes the time complexities associated with key graph operations, including vertex insertion, deletion, and adjacency queries.

Importantly, the red-black tree is selectively applied within the data structure and does not encompass the complete representation of the graph. Rather than being a comprehensive solution, it acts as a nuanced tool, supporting specific operations to leverage its advantages. This careful integration ensures that the red-black tree harmonizes with other components of the data structure, collectively enhancing the overall efficiency of the modified DFS algorithm.

In C++, the `set` container in the Standard Template Library (STL) is typically implemented using a red-black tree. For our algorithm, we will make use of the pre-existing `set` implementation provided by C++, leveraging the efficiency and balanced properties inherent in the red-black tree for seamless coding of our algorithm.

C. Hash Table

A hash table is a data structure that implements an associative array abstract data type, a structure that can map keys to values. It achieves this mapping through a hash function, which takes an input (or key) and produces a fixed-size string of characters, which is usually a hash code. This hash code is then used as an index or address into the array where the desired value can be found.

The primary advantage of a hash table is its ability to provide efficient insertion, deletion, and retrieval of data. When properly implemented, these operations can have an average time complexity of $O(1)$. However, the efficiency relies on the distribution and handling of hash collisions, which occur when two different keys hash to the same index.

To handle collisions, various techniques can be employed, such as chaining (where each array index points to a linked list of elements that hashed to the same index) or open addressing (where the algorithm looks for the next available slot in the array).

Hash tables are widely used in computer science due to their efficiency in implementing dynamic sets, caches, and databases, among other applications. They provide a balance between time and space complexity, making them a versatile and essential data structure in many algorithms and software systems.

In C++, the `unordered_set` container in the Standard Template Library (STL) is commonly implemented using hashing. For our algorithm, we will utilize the existing `unordered_set` provided by C++, taking advantage of its hashing-based implementation for efficient coding.

III. The Algorithm

A. Motivation

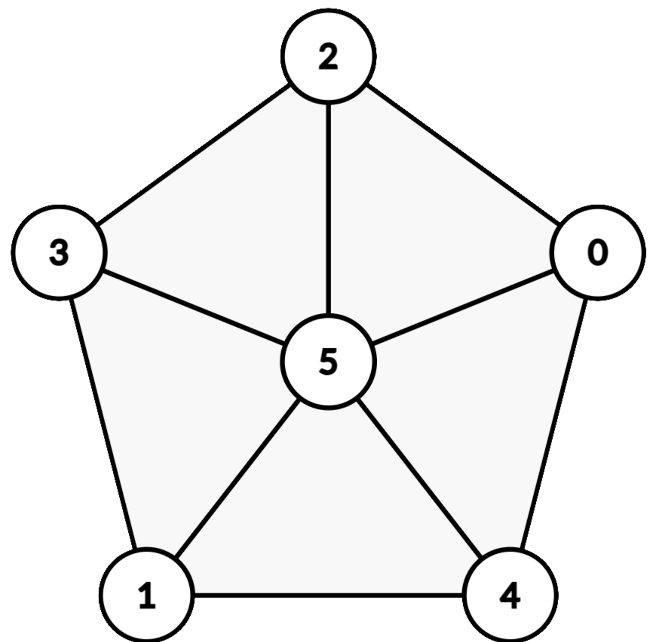


Fig II. A graph consisting of one connected component

Traversing the graph depicted in Fig. II, vertices 0 to 4 are assumed visited, and the current focus is on vertex 5. Upon entering the 'for' loop to check adjacent vertices, it swiftly detects that all adjacent vertices are already visited.

The existing inefficiency in our traversal approach stems from a fundamental challenge: the current representation in the adjacency list doesn't readily provide a clear distinction between neighboring vertices that have already been visited and those that are yet to be explored. This lack of differentiation introduces a bottleneck in the optimization of the traversal algorithm.

To delve deeper, when examining a vertex's adjacency list, the absence of explicit information regarding the visitation status of adjacent vertices necessitates additional checks during the traversal process. This results in redundant assessments, impacting the overall efficiency of the algorithm. The algorithm, as it stands, lacks a mechanism to swiftly identify previously visited vertices, and this hinders the streamlining of the traversal process.

In response to the inherent challenge at hand, we find inspiration to implement a set of strategic maneuvers:

1. Acknowledging the dense nature of the graph, a pivotal strategy involves contemplating the complement of the graph, presumed to exhibit sparser characteristics. This tactical shift aims to exploit the potential advantages offered by a sparser representation, contributing to enhanced computational efficiency.

2. The adoption of a red-black tree emerges as a judicious choice for the storage of unvisited vertices. This decision is grounded in the self-balancing properties of red-black trees, facilitating optimal insertion, deletion, and retrieval operations. Such efficiency contributes significantly to the streamlined performance of the algorithm.

3. Complementing our approach, the incorporation of a hash table is proposed for the storage of edges absent from the original graph. This strategic use of a hash table leverages its hashing mechanism to expedite search and retrieval processes, providing an efficient solution for managing edges that do not exist in the primary graph structure.

B. Algorithm

Below, we implement the modified DFS algorithm incorporating the three strategies mentioned earlier.

```
/*
We presume that the graph is pre-supplied as input to the
adjacency list 'adj'. Additionally, we assume that the set
'unvisited' is initially populated with integers from 0 to V-1,
where V represents the number of vertices in the graph. In the
absence of explicit input, global variables, by default, hold
values of zero or are automatically set to false.
*/
const int V = 1000;
unordered_set<int> adj[V];
set<int> unvisited;

void dfs(int &node) {
    unvisited.erase(node);
    auto it = unvisited.begin();
    while (it != unvisited.end()) {
        int neighbour = *it;
        if (adj[node].count(neighbour)) {
            ++it;
            continue;
        }
        dfs(neighbour);
        it = unvisited.lower_bound(neighbour);
    }
}

void run() {
    while (!unvisited.empty())
        dfs(*(unvisited.begin()));
}
```

C. Time Complexity Analysis

Each node can be either skipped or visited. After a node is visited, it cannot be revisited since it is no longer in `unvisited`. This ensures that each node is visited exactly once. Additionally, acknowledging that there are M edges absent from the graph, and each edge adds exactly 2 to the skip count, the total number of skips is bounded by $2M$. Consequently, the overall computational complexity is $O(V\log(V)+M)$. The factor $O(\log V)$ is due to calling the `lower_bound` function proportional to the number of times the `dfs` function is called.

IV. APPLICATION

Currently, the algorithmic approach is clear-cut, and its potential utility might not be immediately apparent. To illustrate its practicality, we will introduce a problem scenario where the modified DFS algorithm becomes valuable.

To pique interest, we'll present a problem that doesn't revolve around a dense graph. We'll illustrate how the modified DFS algorithm efficiently solves this problem, despite our prior emphasis on its application in the context of dense graphs.

Imagine a sizable grid with dimensions N by N. The flood fill algorithm proves valuable in navigating this grid by treating each cell as a vertex, with adjacent cells forming edges between corresponding vertices. This algorithm finds practical application in various scenarios within paint software, notably when utilizing the bucket tool.

Suppose each cell in the grid is assigned a numerical value. In this context, a "region" is characterized by a set of cells sharing the same number. For a group of cells to constitute a region, each cell in the region must be directly adjacent to another cell in the same region, considering only above, below, left, or right directions (diagonals are not considered). The task of identifying the largest region can be efficiently tackled using the flood fill algorithm, employing the traditional DFS algorithm, with an optimal time complexity of $O(N^2)$.

Now, if we aim to identify the largest region formed by at most two numbers, a naive approach would involve a time complexity of $O(N^6)$. This is because, in an N x N grid, there are at most N^2 numbers, resulting in $C(N^2,2)=\frac{N^2(N^2-1)}{2}=O(N^4)$ pairs of numbers. For each pair, we can find the largest possible region in $O(N^2)$, leading to an overall time complexity of $O(N^6)$.

An improved solution can be outlined as follows. We can conceptualize the problem as a graph, where each vertex denotes the locally maximum region, and edges signify adjacency between two regions. The size of the region can be stored as the weight of the corresponding vertex. This transforms the problem into the quest for a path where the sum of weights is maximized, and efficient edge traversal algorithms become crucial for this optimization.

The core concept involves employing a modified DFS. Instead of iteratively traversing all edges connected to a vertex to identify traversed edges, we utilize a set data structure. This allows us to find the untraversed edge in $O(\log N)$ time, as opposed to linear time.

Below is the complete C++ program. The initial input will be N, followed by N x N numbers representing the values in each cell of the grid. For example, given the input below, the output would be 10, using region 10 and region 02.

```
4
03 04 10 04
05 10 10 02
10 10 02 08
03 02 02 10
```

Here is the complete C++ implementation of the described approach:

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;

int n;
int board[250][250];
int board_vertex_id[250][250];
int cow_id[62'500];
int vertex_size[62'500];
set<pair<int, int>> adj[62'500];
bool v[250][250];

void floodfill(int row, int col, int id, int vertex_id, int &region_size) {
    if (!(0 <= row && row < n && 0 <= col && col < n)) return;
    if (board[row][col] != id) return;
    if (v[row][col]) return;

    v[row][col] = true;
    ++region_size;
    board_vertex_id[row][col] = vertex_id;

    floodfill(row+1, col, id, vertex_id, region_size);
    floodfill(row, col+1, id, vertex_id, region_size);
    floodfill(row-1, col, id, vertex_id, region_size);
    floodfill(row, col-1, id, vertex_id, region_size);
}

void dfs(int vertex_id, int other_cow_id, int &total_size, set<int> &visited) {
    if (visited.find(vertex_id) != visited.end()) return;
    total_size += vertex_size[vertex_id];
    visited.insert(vertex_id);

    auto it = adj[vertex_id].lower_bound({other_cow_id, 0});
    while (it != adj[vertex_id].end()) {
        auto neighbour_vertex_id = it->second;
        if (it->first != other_cow_id) return;
        adj[vertex_id].erase(it);
        adj[neighbour_vertex_id].erase({cow_id[vertex_id], vertex_id});
        dfs(neighbour_vertex_id, cow_id[vertex_id], total_size, visited);
        it = adj[vertex_id].lower_bound({other_cow_id, 0});
    }
}
int main() {
    cin >> n;
    for (int row = 0; row < n; ++row) {
        for (int col = 0; col < n; ++col) {
            cin >> board[row][col];
        }
    }
    int vertex_id = 0;
    for (int row = 0; row < n; ++row) {
        for (int col = 0; col < n; ++col) {
            if (v[row][col]) continue;
            int region_size = 0;
            floodfill(row, col, board[row][col], vertex_id, region_size);
            vertex_size[vertex_id] = region_size;
            cow_id[vertex_id] = board[row][col];
            ++vertex_id;
        }
    }
    for (int row = 0; row < n; ++row) {
        for (int col = 0; col < n-1; ++col) {
            if (board_vertex_id[row][col] == board_vertex_id[row][col+1]) continue;
            adj[board_vertex_id[row][col]].insert({board[row][col+1], board_vertex_id[row][col+1]});
            adj[board_vertex_id[row][col+1]].insert({board[row][col], board_vertex_id[row][col]});
        }
    }
    for (int row = 0; row < n-1; ++row) {
        for (int col = 0; col < n; ++col) {
            if (board_vertex_id[row][col] == board_vertex_id[row+1][col]) continue;
            adj[board_vertex_id[row][col]].insert({board[row+1][col], board_vertex_id[row+1][col]});
            adj[board_vertex_id[row+1][col]].insert({board[row][col], board_vertex_id[row][col]});
        }
    }
    int second_answer = 0;
    for (int i = 0; i < vertex_id; ++i) {
        while (!adj[i].empty()) {
            auto it = adj[i].begin();
            int total_size = 0;
            set<int> visited;
            dfs(i, it->first, total_size, visited);
            second_answer = max(second_answer, total_size);
        }
    }

    cout << second_answer;
}
```

```
4
03 04 10 04
05 10 10 02
10 10 02 08
03 02 02 10
```

Input

```
10
```

Output

The code begins by taking the input N, representing the size of the grid. Subsequently, it reads N by N numbers to form the grid. Following this, a standard flood-fill algorithm is initiated. Each connected component within the grid is treated as a vertex, and the weight assigned to the vertex corresponds to the number of cells within the connected component. The next step involves establishing edges by traversing the grid, considering every adjacent horizontal and vertical pair of cells. Subsequently, the program navigates through the graph using a technique akin to the modified depth-first search (DFS) algorithm.

It's important to note that this isn't an entirely distinct algorithm from the modified DFS algorithm. We continue to employ the `lower_bound` function to determine the next vertex to traverse, akin to the modified DFS algorithm. Consequently, the concept behind the algorithm in the modified DFS isn't exclusively tailored for dense graphs; rather, it can be applied in various scenarios.

V. Conclusion

While we have presented the modified DFS algorithm, we have also showcased several alternative strategies. These include employing a balanced binary tree in place of a conventional array to eliminate redundant checks, utilizing an array of hash tables for efficient validation, formulating the problem within the framework of graph theory, and exploring various other approaches.

We have also presented an application of the idea, demonstrating that even when the graph is not dense, the essence of the modified DFS algorithm can still be effectively utilized to solve the problem.

This paper explores several key concepts. Firstly, it delves into time complexity analysis. Additionally, we employ the concept of a tree, specifically a balanced binary search tree using a red-black tree structure, enabling efficient $O(\log n)$ operations. Combinatorial arguments are utilized to determine the time complexity. Moreover, the study incorporates data structures that leverage hashing.

VI. Acknowledgment

The author would like to express gratitude to Allah SWT because, by His grace and mercy, the author was able to complete this paper titled "Exploring Efficient Techniques for Connected Component Analysis in Dense Graphs: Unveiling Practical Applications" successfully. Not forgetting, the author sincerely thanks both parents who have provided unwavering support, prayers, and love throughout the journey of writing this paper. Their presence has been a source of inspiration and strength for the author. Additionally, the author would like to convey heartfelt thanks to the lecturer of the Discrete Mathematics course, Dr. Fariska Zakhralativa Ruskanda, S.T., Dr. Ir. Rinaldi Munir, M. T., and Dr. Nur Ulfa Maulidevi, S. T, M. Sc., for their guidance during the course. Last but not least, the author would like to express gratitude to the author's friends who have provided moral support and encouragement during the preparation of this paper.

Reference

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms, fourth edition (4th ed.). MIT Press.

[2] Skiena, S. S. (2020). The Algorithm Design Manual (3rd ed.). Springer.

[3] Rosen, K. H. (2018). Discrete Mathematics and its Applications. McGraw hill.

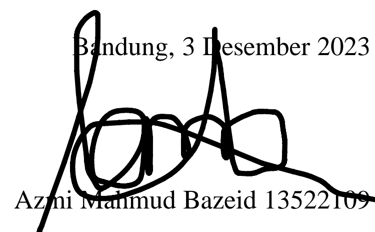
[4] Problem - 920E. (n.d.). Codeforces. Retrieved December 11, 2023, from https://codeforces.com/problemset/problem/920/E

[5] USACO. (n.d.). Usaco.org. Retrieved December 11, 2023, from https://www.usaco.org/index.php?page=viewproblem2&cpid=836

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 3 Desember 2023



Azmi Mahmud Bazeid 13522109