EPiC Series in Computing

Volume 98, 2024, Pages 129–139

Proceedings of 39th International Conference on Computers and Their Applications

Optimizing Time Complexity: A Comparative Analysis of Techniques in Recursive Algorithms - A Case Study with Path Sum Algorithm in Graphs and Binary Trees

Jonathan Shields and Thitima Srivatanakul

York College, City University of New York, New York, U.S.A.

jonathan.shields@yorkmail.cuny.edu, tsrivatanakul@york.cuny.edu

Abstract

Programming infrastructures commonly employ graphs and binary trees to model systems and networks. Efficient operations on trees and graphs are pivotal in enhancing software performance and reducing computational costs, particularly for data-dependent tasks during runtime. This paper analyzes the optimization techniques for recursive algorithms, focusing on the widely used Path Sum algorithm designed for identifying cumulative value sequences that equal a specified target. Employing three distinct techniques—recursion, tabulation, and memoization—this study evaluates their computation time on two prominent data structures: trees and graphs. Results indicate that the memoization approach is completed in less computational time than the regular approach. In contrast, the tabular approach completes in significantly increased computational time, suggesting its inadequacy for traversal optimization. The findings affirm that optimization techniques, particularly memoization, effectively reduce traversal computation time, offering valuable insights for educators and developers working with recursive algorithms in graph and tree-based systems.

1 Introduction

The integration of graphs and binary trees in various programming infrastructures is ubiquitous. Graphs are in various real-world applications, including navigation systems, route optimization, social networking, and recommendation systems. Similarly, binary trees find applications in scenarios like hierarchical data representation. Efficient operations on both graphs and binary trees are crucial for improving software performance and minimizing computational expenses. Graph and binary tree algorithms can be implemented through different approaches, including recursion and dynamic programming, to address specific challenges within these structures. Recursion, a fundamental

programming technique, involves a function defining its own process by calling itself, with each call until a base case is met. While the use of recursion reduces the need for complex tabular approaches promotes the development of concise code, and enhances overall readability, it does come with computational setbacks [1]. The recursion process of repeated function calls can become computationally intensive, increasing time complexity with each call. The successive self-function calls are managed within the call stack, an internal data structure responsible for organizing and managing function calls. However, this call stack has a predetermined allocation size, and exceeding this limit leads to a stack overflow error, resulting in an abrupt interruption of program execution [2].

Scholarly attention has been directed towards optimization techniques using dynamic programming methods [3], demonstrating their potential to significantly improve resourcefulness over recursive algorithms [2]. The two optimization techniques of focus are memoization [3] and tabulation [4]. These are dynamic programming techniques that reduce time complexity via efficient methods. The memoization approach is a top-down approach that caches computed results. When a repeated function is called the cache can recall the stored output instead of reevaluating the function, saving computational resources [5]. The tabular approach is a bottom-up approach that breaks down the problem into smaller subproblems and builds up solutions incrementally. It utilizes a container, such as array, for all possible values, avoiding the complexity that comes with recursion [6].

The paper seeks to provide educators and developers with insights into the application of optimization techniques to a recursive algorithm. The recursive algorithm used for the study is the Path Sum algorithm. The Path Sum algorithm is designed to identify a sequence of values that cumulatively sum up to a specified target value. This study implements two distinct techniques – tabulation and memoization – for implementing the Path Sum Algorithm on two different data structures: trees and graphs. In analyzing how the time complexity changes with these optimizations in graph and binary tree structures, the research aims to explore the connection between theoretical concepts and their practical implementations. Through this exploration, valuable insights can be for both theoretical understanding and real-world implementation.

The paper is organized as follows: Section 2 on Literature Review discusses prior studies on this topic. Section 3 discusses the methodology to explain the procedure and components used for the test. Section 4 outlines the results and discussion. Section 5 concludes the paper.

2 Literature Review

The field of graph optimization is an area of study active with ongoing research. Many of these studies were conducted by researchers interested in multifaceted distributed systems. These systems find application in a wide array of scenarios requiring quick dynamic storage and retrieval of data. In such cases, enhancements in optimization effectiveness can result in compounding increases in computational efficiency in these distributed systems, allowing the reallocation of computational resources to uphold overall system performance [7].

The work by Lee et al. [8] discusses utilizing dynamic tracing to identify points in data processing in a distributed system that could be optimized with memoization. With the use of dynamic tracing, the scaling performance of distributed systems can improve computational efficiency by an average of 4.9 times its usual ability after performing traversal through ~ 256 nodes within a model representation of the graph [8]. The study covered both the application of memoization and the utilization of graph structures. It effectively illustrates how memoization functions as an optimization

technique within real-world services that integrate graph structures into the application schema. The study by Gong et al. [9], conducted in collaboration with Alibaba Group, a multinational conglomerate operating diverse Cloud Computing and E-commerce Platforms, proposes a potential solution for dynamic data handling. The authors recommend using an automated system for incremental graph processing known as Ingress [9]. Described as "an automated incrementalization framework," Ingress strategically applies varying degrees of memoization to support diverse computational resources, optimizing memory usage. The study asserts that their developed Ingress algorithm outperforms modern incremental graph systems by an average of 15.93 times. This research further emphasizes memoization as a vital technique for maintaining efficient systems under dynamic conditions. Although the paper introduces the use of four different levels of memoization, the rationale behind this approach is not extensively elaborated.

Papers addressing the tabular approach concerning graphs were limited, yet a few discussed the advantages of employing a tabular approach in handling computationally intensive processes. Lue & Pope [10] explored into the utilization of an adaptive tabulation algorithm called ISAT, where tabulation represents a specific form of a tabular approach. Their work aimed to enhance the ISAT tabulation by incorporating "table-searching strategies" and introducing "error checking and correcting algorithms" [10]. This paper offers valuable insights into the detailed explanation of tabulation, playing a pivotal role in data retrieval as memoization reduces computational time. Studies comparing optimization techniques for both graphs and trees are limited. This paper aims to provide additional insights into one of the recursive algorithms.

3 Methodology

The Path Sum algorithm is an algorithm that implements the use of recursion to find a sequence of values that collectively sum up to a specified target value. In a binary tree, the algorithm moves through an unsorted binary tree from the root of the tree down to adjacent paths, traversing the entire tree to find instances of the target value in the tree. With graphs, a traversal algorithm is used to visit distinct nodes. The traversal algorithm of choice was depth first search (DFS) due to its memory efficient nature in comparison to other traversal algorithms such as breath first search (BFS). In this study Path Sum algorithms that were unoptomized, utilizing memoization, and utilizing tabulation were tested on binary trees and graphs of various node sizes.

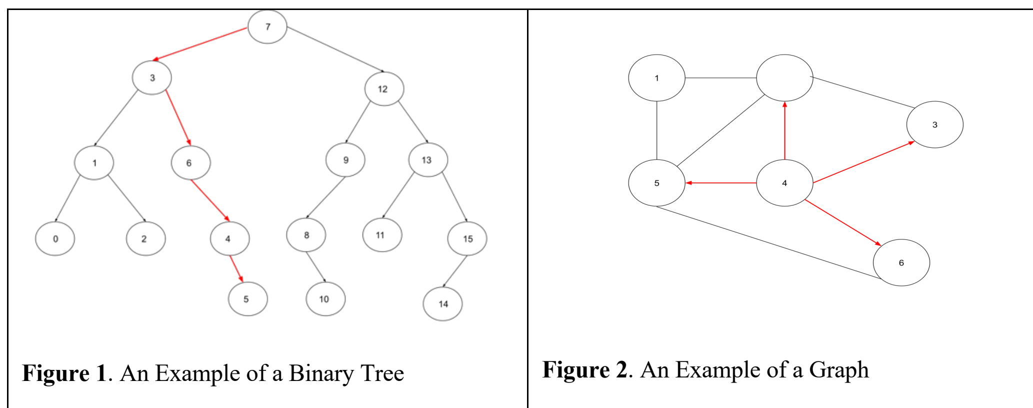

Figure 1. An Example of a Binary Tree

Figure 2. An Example of a Graph

In Figure 1, the red arrows line out a potential path for a target sum. The total number of paths that equal the target sum will be returned. For example, this figure shows an accepted path if the target sum was 25. Figure 2, the red arrows show the direction of search. The figure shows a multidirectional search to return the total number of paths that equal the target sum. The Path Sum algorithms used in the study were written in C++ and compiled using gcc version 11.4.0 on a stand-alone personal computer utilizing an Intel i5-825OU with 8 cores clocked at 3.4 GHz.

3.1 Dynamic programming using Memoization and Tabulation

Dynamic programming remains a highly active area of study in computer science and is only projected to increase due to its applicability in diverse domains. Its relevance is based on the need for various computational processes to solve complex problems efficiently. Dynamic programming involves taking a whole complex problem and dividing it into parts of simpler subproblems. The solution to these simpler subproblems is then stored in a container. When an identical problem needs to be solved, instead of spending resources recalculating the values the result can be referenced from the container. Dynamic programming can be implemented using two different approaches: the top-down approach with memoization and the bottom-up approach with tabulation.

Memoization is a recursive approach to solving complex problems. It addresses problems by initially tackling the entire problem and then recursively solving smaller subproblems, storing the solutions in an array or hash map along the way. When encountering a problem that has already been solved and stored, the stored solution is retrieved. Otherwise, the solution is computed and stored. This ensures that identical problems are only computed once, with subsequent occurrences retrieving the precomputed answer. Memoization provides significant improvements for problems involving repetitive calculations and nested subproblems. As a result, it reduces program execution time and enhances system performance.

Tabulation is an iterative approach to solving complex problems. Instead of solving the most complex form of a problem and solving subsequent subproblems with memoization, tabulation takes a bottom-up approach. Tabulation begins with the smallest form of a complex problem and gradually builds up on top of the solution to the original problem. Starting from the base case of the problem, computed values from a table data structure such as a 2-dimensional array. Once this array is filled, an iterative approach is used to go through the object and retrieve the correct solution. This allows solutions to be extracted from the table instead of through repetitive computations, which leads to improved algorithm performance.

3.2 Path Sum Algorithms on Binary Trees

For binary trees, we tested plain Path Sum algorithm using recursion, along with optimized Path Sum algorithms employing memoization and tabulation. The algorithms were evaluated on binary trees of different sizes, specifically 10-node, 100-node, 1000-node, 10,000, and 100,000 trees. The values for each node were randomly populated. The execution times (μs) were then compared across 10 runs and averaged. The Path Sum algorithm recursively explores paths, subtracting node values from a target sum and checking if leaf nodes satisfy the sum condition. The approach utilizes a depth-first search, efficiently examining left and right subtrees.

The pseudocode of each algorithm is shown below:

```
Algorithm Recursion(node,k):
       Input: A node “node” in the binary tree, integer k storing target
       value
       Output: An integer that is the number of paths that equal k.

       if n == 1 then
              return 0;
       return  (k  ==  node.data  ?  1:0)  +  Recursion(node.left,k  –
       node.data) +
       Recursion(node.right, k – node.data);
```

Figure 3. The pseudocode for Recursion Algorithm without optimization.

```
Algorithm memoization(node,k,sum,map):
     Input: A node “node” in the binary tree, integer k stores target
value, sum is an integer values to keep track of sum, map is unordered
list contains integer values for key and value.
     Output: An integer that is the number of paths that equal k.

       if(node == nullptr) then
              return 0;

       sum = sum + node.data
              map[sum] = map[sum] + 1
              int count = map[sum – k]
              int result = count + memoization (node.left,k, sum, map)
              map[sum] = map[sum] -1
              return result
```

Figure 4. The pseudocode for Recursion Algorithm using memoization.

```
Algorithm tabulation(node,k):
       Input: A node “node” in the binary tree, integer k stores target
       value.
       Output: A integer that is the number of paths that equal k.

       2D_array matrix
       // Populating the matrix
       result = 0
       for each row in matrix do
       for each sum in row do
       if sum in row then
              if sum == k then
                     result = result + 1
       return result
```

Figure 5. The pseudocode for Recursion Algorithm using tabulation.

In Figure 3, it depicts the pseudocode for the algorithm searching for a target value in a binary tree without any optimization modifications, in which case each path in a tree will be visited regardless of the path has been visited previously. In Figure 4, the recursive algorithm for searching a target value in a binary tree is shown, incorporating memoization to cache visited paths. Finally, Figure 5 illustrates the use of tabulation to populate a matrix from the nodes in a binary tree, utilizing iteration directly instead of recursion to determine the total number of paths equaling a target value.

3.3 Path Sum Algorithms on Graphs

As opposed to binary trees, graphs are a generic structure that has vertices connected by edges. While a binary tree can have a maximum of two edges to a node, a graph vertex can have any number of edges and therefore makes them a preferred structure to model more complex relationships. Since graphs can have multiple edges to a vertex, a more complex traversal algorithm is needed. The traversal algorithm of choice was the depth first search algorithm. This algorithm traverses through a graph starting at any node and moves to as far as it can before backtracking to other nodes.

The Path Sum algorithm works by traversing paths within the graph structure, deducting node values from a sum, and finding whether any path satisfies the given sum condition. Using a recursive approach like depth-first search (DFS), the algorithm explores different paths in the graph. It looks to find if the cumulative sum matches the target sum criteria along the explored path. Unlike trees that have left and right subtrees, in graphs, the algorithm navigates through nodes that are interconnected, exploring potential paths until it finds a valid path fulfilling the sum condition. The use of the efficient DFS-based search enables it to traverse the graph, analyzing various potential paths to determine the cumulative values identify any path that satisfies the required sum condition

```
Algorithm recursion(graph, nodeValue, sum, totalSum):
        Input: graph represents the graph object, nodeValue is the value
        of a node and the totalSum is the total value of the node.

        Output: Returns the total sum of paths

        sum = sum + nodeValue;
        if (graph is empty)
               totalSum = totalSum + sum;
               return;

        for (v in graph)
               recursion(graph,v,sum,totalSum);
```

Figure 6. Pseudocode for Recursion Algorithm without optimization

```
Algorithm memoization(graph, nodeValue, sum, totalSum, memo):
       Input: Graph represents the graph object, nodeValue is the value
of a node, sum is the target value the totalSum is the total value of
the node, memo is the map.
       Output: Returns the total sum of paths

              sum = sum + nodeValue;
              if (graph is empty)
              totalSum = totalSum + sum;
              return;

              for (v in graph)
                     recursion(graph,nodeValue,sum,totalSum);
              memo[nodeValue] = sum;
```

Figure 7. Pseudocode for Recursion Algorithm with memoization

```
Algorithm tabulation(graph, nodeValue)
       Input: Graph represents the graph object and the nodeValue is a
variable that holds the total value at a node.

       Output: Returns the total sum of paths

              totalSum = 0;
              Stack stk;
              Array visited;

              for (i = 0; i < nodeValue; ++i)
              stk.push({i,0})

              while(not stk.empty())
              curr = stk.top();
              stk.pop();

              nodeValue = curr.first;
              sum  = curr.second + nodeValue;

              if (not visited[nodeValue].empty())
              totalSum = totalSum + sum;

                     if(graph[nodeValue].empty())
                            totalSum =  totalsum + sum;
                     else
                         for (v in graph[nodeValue])
                             stk.push({v,sum]);
```

Figure 8. Pseudocode for Resursion Algorithm with Tabulation

In Figure 6, it illustrates the recursive algorithm that implements a search through the graph using recursive DFS where the traversal is performed without any recursive optimizations. In Figure 7,

memoization is implemented to store repeated value of paths by the map object labeled *memo* to avoid visited paths. Finally, Figure 8 demonstrates the implementation of tabulation, in which a table is created and used to perform a search in a matrix, utilizing a stack for traversal of each row.

The study was conducted on 5 graphs, each containing various amounts of vertices. Each graph consists of the following sizes: 10, 100, 1,000, 10,000, and 100,000 vertices respectively. The objective is to use three different algorithms to return the total sum of values in an Undirected Graph. As stated earlier, the differentiation between the algorithms is the methods used to traverse through the graph. There is traversal through recursion without any formulated optimization techniques, then traversal using memoization technique and lastly traversal through tabulation technique. Each of the three Path Sum algorithms will traverse through 5 differently sized graphs. The ten trials will be conducted for each path sum algorithm on each graph and the average time of the ten will be recorded for analysis.

4 Results and Discussion

In the tables below, we present the results of a comparative analysis of the Path Sum algorithm on binary trees and graphs for various node sizes. The three techniques considered for evaluation are recursion, tabulation, and memoization, utilizing the algorithms discussed in the previous section. The execution times, measured in microseconds, offer insights into the efficiency of each approach in addressing the Path Sum problem.

Node size	Average Time for Recursion (µs)	Average Time for Memoization (µs)	Average Time for Tabulation (µs)
10	3	16	13
100	12	152	20706
1,000	1550	1036	N/A
10,000	4856	3370	N/A
100,000	11496	7412	N/A

Table 1. Comparison of Average Execution Times (in µs) for Binary Tree (varying node size) with Recursion, Memoization, and Tabulation Techniques

Node size	Average Time for Recursion	Average Time for Memoization	Average Time for Tabulation
10	0	1.5	11
100	13	17.2	205
1,000	205	188	1143
10,000	596	483	N/A
100,000	5483	3891	N/A

Table 2. Comparison of Average Execution Times (in µs) for Graph (varying node size) with Recursion, Memoization, and Tabulation Techniques

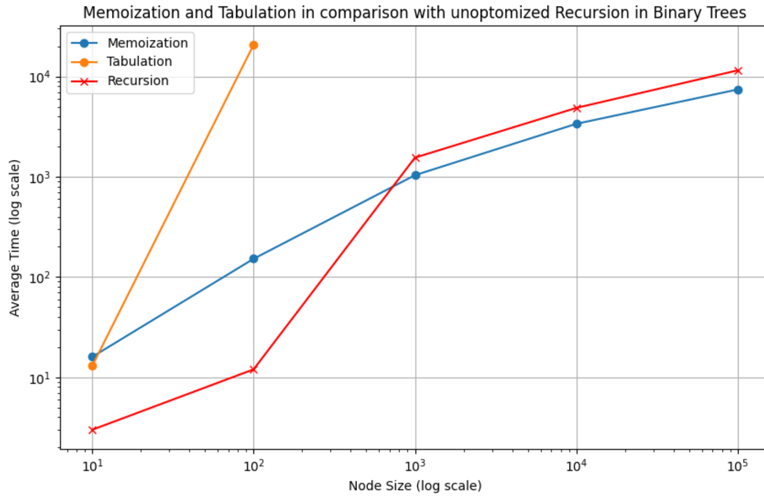

Figure 9. Plot of the average execution times of recursion without optimization, memoization and tabulation utilizing a binary tree on a logarithmic scale performing the Path Sum algorithm.

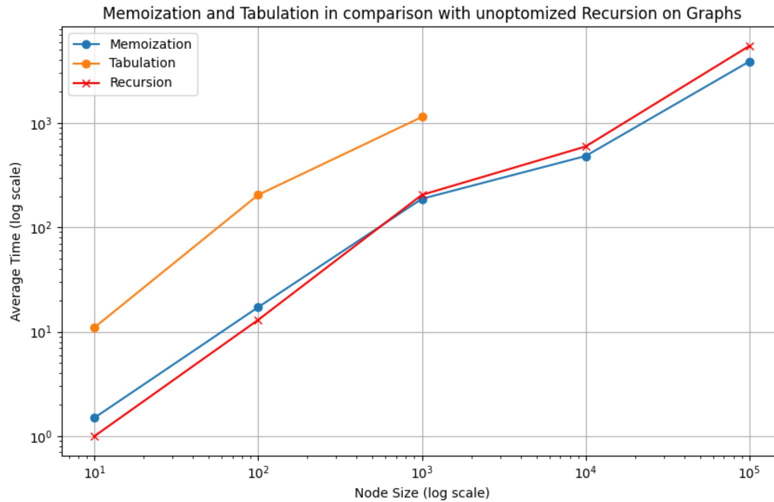

Figure 10: Plot of the average execution times of recursion without optimization, memoization and tabulation utilizing a Graph on a logarithmic scale performing the Path Sum algorithm.

The results obtained from the comparative analysis of the Path Sum algorithm on binary trees using recursion, memoization, and tabulation as presented in Table 1. The table highlights a fundamental difference between memoization and recursion without optimization techniques, influenced by the inherent characteristics of the binary tree structure. Recursion, while efficient for smaller node sizes, experiences notable increases in execution time as the tree size grows. On the other hand, with memoization, completion time was initially greater for smaller trees compared to unoptimized recursion. However, as the number of nodes in the tree grew, the rate of computation

became smaller in comparison. This is evident when comparing the completion time between a tree of size 100 and a tree of size 1000. Tabulation demonstrates the worst performance with the greatest completion time compared to the other algorithms. It also failed to perform during the third test, which yielded inconclusive results (in this case, N/A).

The results obtained from the comparative analysis of the Path Sum algorithm on graphs using recursion, memoization, and tabulation as presented in Table 2. Regarding the graph data structure, similar trends were observed, with memoization eventually outperforming other methods in the long term. The tabulation, once again, performed poorly, providing inconclusive results for higher trees. This may be attributed to the algorithm's design, as tabulation typically excels in static problems that can be subdivided into smaller, manageable problems.

A similar outcome that was apparent in both data structures was how object initialization makes up the bulk of the time complexity for the tabular approach. This is evident from the observed values from the tabulation data for both binary trees and graphs. For both, tabulation was unable to finish the tests in the experiment. For the binary trees, that occurred when the tree size reached 1,000 nodes, and when the size reached 10,000 node size for the graph. However, the same results were also demonstrated using memoization were evidence suggested a decrease in the overall computational time in relation to recursion without optimization. This is in harmony with the work by Gong et al. [9] in which the use of memoization was a key feature in the implementation of high performing incremental graph processing. The implications of these results demonstrate that memoization seems to be a more efficient technique when working with large datasets. Based on the results, it seems that for graphs with fewer vertices, the path sum algorithm traversed the entire graph more quickly compared to the process of memoization. This is especially clear in the traversal of the graph with 10 vertices where the regular non optimized approach traversed in < 0 μs. However, when the number of vertices reaches 1000 the memoization tends to complete quicker than the regular counterpart. This is a phenomenon that only increases as the number of vertices grows.

Overall, the results do seem to resonate with other papers that explore this topic in more detail, however there are some parts of the testing process that may have been overlooked. For one, the path sum algorithm could have been optimized further with the use of specific optimization flags accepted by the gcc compiler. Though not a major issue, it could have returned more exaggerated times that more explicitly identified the computational savings of each algorithm. Another potential shortcoming is the lack of consistent garbage collection within the program. Due to the nature of C++, the lack of an autonomous garbage collection means that space demanding data structures such as the graphs can interfere with the efficiency of programs in runtime. Though in testing they may not serve as much of an influence in production it can be more of an issue that programmers need to be mindful of. Lastly in memoization, hash collisions can significantly impact lookup time, potentially undermining the efficiency gains of caching computed values.

5 Conclusion

In conclusion, the results gathered show that the tabular approach is not a proper technique to optimize traversal given that it takes over five times more computational time on average than the regular approach. The memoization approach yielded results that signify quicker computational time compared to the regular approach. The rate of computational time between each graph in ascending order was on average 1100 μs while memoization was on average 778 μs per graph, a growth rate that

is about two thirds of regular recursion. Therefore, the study still stands that with the use of optimization techniques traversal computation time can be reduced, specifically with the use of memoization.

These results can have a broad implication on various programming systems that rely on graph structures for data storage/retrieval and as a performance layer leading to the utilization of programs that query quicker and consume less computational resources. For future research directions, we would investigate the potential for parallel computing for optimizing traversal algorithms or adaptive algorithms that can dynamically switch between different optimization techniques based on the size of the graph and other factors.

References

[1] Wiedenbeck, S. (1988). Learning recursion as a concept and as a programming technique. ACM SIGCSE Bulletin, 20(1), 275-278.

[2] Liu, Y. A., & Stoller, S. D. (1999, November). From recursion to iteration: what are the optimizations?. In *Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation* (pp. 73-82)

[3] Bellman, R. (1966). Dynamic programming. *Science*, *153*(3731), 34-37.

[4] Bird, R. S. (1980). Tabulation techniques for recursive programs. *ACM Computing Surveys (CSUR)*, *12*(4), 403-417.

[5] Blelloch, G. E., & Harper, R. (2015). Cache efficient functional algorithms. *Communications of the ACM*, *58*(7), 101-108.

[6] Zhou, N. F., Sato, T., & Shen, Y. D. (2008). Linear tabling strategies and optimizations. *Theory and Practice of Logic programming*, *8*(1), 81-109.

[7] Ventryshia, A., & Wirawan, I. (2020). Analysis of Fibonacci Numbers Calculations Using Static Programming and Dynamic Programming Algorithms to Get Optimal Time Efficiency. *International Journal of Open Information Technologies*, *8*(12), 19-22.

[8] Lee, W., Slaughter, E., Bauer, M., Treichler, S., Warszawski, T., Garland, M., & Aiken, A. (2018, November). Dynamic tracing: Memoization of task graphs for dynamic task-based runtimes. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis* (pp. 441-453). IEEE

[9] Gong, S., Tian, C., Yin, Q., Yu, W., Zhang, Y., Geng, L., ... & Zhou, J. (2021). Automating incremental graph processing with flexible memoization. *Proceedings of the VLDB Endowment*, *14*(9), 1613-1625.

[10] Lu, L., & Pope, S. B. (2009). An improved algorithm for in situ adaptive tabulation. *Journal of Computational Physics*, *228*(2), 361-386.