The Application of Reconstructed Trees in Collegiate Programming Contests

Zijie Shen, Ruixiang Li, Junping Shi *
School of Computer Science and Engineering, Jishou University, Jishou, Hunan, China
**Corresponding Author.*

Abstract: In collegiate programming contests, reconstructed trees find extensive application in tackling path-edge-weight constraint problems within graphs. These trees, defined as structures where edges carry precisely unit weight, reflecting the original graph's node set, serve as potent tools. This article elucidates the methodology behind constructing reconstructed trees, primarily leveraging minimum spanning trees. Additionally, it delves into the optimization of queries employing binary lifting coupled with Lowest Common Ancestor (LCA) algorithms. By delving into specific problem instances, it discerns the multifaceted advantages inherent in employing reconstructed trees within the competitive ambiance of collegiate programming contests. Beyond merely resolving path-edge-weight constraints, these trees augment algorithmic efficiency and code comprehensibility. They furnish contestants with robust utilities and techniques, fostering enhanced performance and enabling them to navigate contests with greater adeptness, thereby contributing significantly to their competitive endeavors. Their utility extends beyond the confines of contests, finding application in diverse real-world scenarios, enriching problem-solving capabilities, and fostering a deeper understanding of graph theory concepts.

Keywords: Reconstructed Trees; Minimum Spanning Trees; Binary Lifting with Lowest Common Ancestor

1. Introduction

International College Programming Competition (ICPC) has evolved into the preeminent collegiate programming contest [1] globally, an annual extravaganza that serves as a platform for showcasing the innovative prowess, collaborative dynamics, and programming acumen of university students from diverse backgrounds and cultures. Over the course of its illustrious history spanning decades, the competition has not only grown in scale but also in significance, attracting an ever-expanding pool of participants eager to test their mettle against the best and brightest in the field. As the landscape of computer science continues to evolve, so too do the challenges presented at ICPC, with an increasing emphasis on graph theory and tree-based algorithms emerging as prominent themes. To meet these challenges head-on and to excel in the intense and competitive environment of ICPC, participants have turned to a myriad of sophisticated data structures and algorithmic techniques. Among these, the reconstructed tree has garnered particular attention for its efficacy in addressing problems that demand expedited solutions to queries related to path-edge weight constraints within graphs. Its versatility and efficiency have made it a cornerstone of modern programming competitions, enabling contestants to navigate complex problem sets with confidence and precision. As ICPC continues to push the boundaries of innovation and excellence, the strategic utilization of reconstructed trees underscores the symbiotic relationship between theoretical knowledge and practical application, empowering participants to push the envelope of what is possible in the ever-evolving landscape of computer science and programming competitions.

2. The Current Research Status of Reconstructed Trees

The essence of a reconstruction tree lies in transforming the edges of a tree into new nodes, wherein each new node encapsulates the information of the original edge, specifically its weight. The purpose of this

refactoring is to enhance the representation of connections between nodes, such as determining the maximum or minimum value along the path between two points. This transformation not only furnishes a more intuitive data structure but also furnishes a more efficient platform for executing various algorithms and queries on the tree. By consolidating edge information into nodes, we can effortlessly access pertinent attributes of paths, thereby optimizing the tree's structure and enhancing data processing efficiency.

Refactoring trees can be used in a wide range of scenarios, such as processing dynamic data flows and achieving efficient memory management. At present, the research of reconfigurable tree mainly focuses on how to optimize and improve its algorithm performance to adapt to the ever-changing application requirements. The research directions include algorithm optimization [2], dynamic maintenance [3], application extension [4], theoretical analysis [5], parallelization and distributed computing [6].

3. The Principles and Implementation of Reconstructed Trees

3.1 Definition

Refactoring trees, a cornerstone concept in computer science, embody a dynamic approach to processing and organizing intricate data structures. At its core, a refactoring tree represents a versatile framework that accommodates the evolving needs of software projects by facilitating the flexible reconstruction of data hierarchies. This concept transcends mere data organization; it encapsulates a philosophy of adaptability and scalability essential for navigating the complexities of modern software development. Unlike static data structures, refactoring trees offer a high degree of reusability, allowing developers to iteratively refine and optimize their codebase in response to changing requirements and evolving design paradigms. This inherent flexibility empowers software engineers to seamlessly adapt their data structures to accommodate new features, scale to accommodate growing datasets, and optimize performance to meet evolving user demands. By providing a modular and extensible foundation for data management, refactoring trees promote code reuse, modularity, and maintainability, fostering a more agile and resilient software development process. As such, mastering the principles of refactoring trees is not merely a technical skill but a strategic imperative for navigating the ever-changing landscape of modern software engineering.

3.2 Principle

The reconstructed tree is generally constructed in a bottom-up manner, from small nodes to large nodes, layer by layer, recursively, until it reaches the topmost root node. In a reconstructed tree, each node represents a specific data structure, such as groups of numbers, linked lists, trees, and so on. Each node has its own data field and pointer field for storing data and Pointers to other nodes. The key to refactoring trees is that existing nodes can be modified, added or removed according to new requirements, and then a new data structure can be reconstructed.

3.3 Concrete Implementations

3.3.1 Filtering key data

In a graph, there exist insignificant edge data. To filter out critical edge weight information for building a reconstructed tree, the Kruskal algorithm [7], can be used to eliminate irrelevant edges. Kruskal is an excellent minimum spanning tree [8], algorithm based on union lookup set.

The general process of the Kruskal algorithm is as follows:

First, sort all the edge weights in ascending order.

Then, enumerate each edge in sequence. If the two vertices of an edge are not in the same connected component, add this edge to the edge set of the spanning tree and merge the two vertices into one connected component.

As shown in the Figure 1:

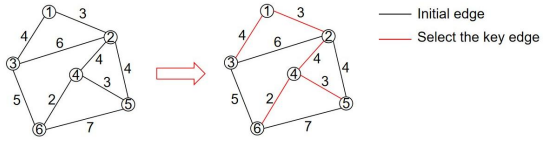


Figure 1. Minimum Spanning Tree Result

3.3.2 Build reconfiguration tree

First, according to the nodes in the figure, a single node tree with itself as the parent node is generated successively.

Then the edge data screened by the minimum spanning tree algorithm is sorted according to

the edge weight from small to large. The edges are taken successively, and a new parent node is established based on the edge weight of the edge. The two child nodes of the parent node are the top parent nodes of the left and right nodes of the edge taken respectively.

As shown in the Figure 2:

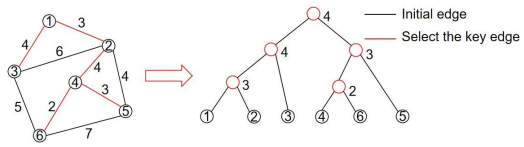


Figure 2. Reconstruction Tree Creation

Repeat the above steps until all edges are reconstructed. The reconstructed tree is successfully established.

3.3.3 Information query

To query the minimum value of the maximum edge weight of the path between any two points in the graph, that is, to find the nearest common parent of the two points in the reconstruction tree, and the data of the parent node is the minimum value of the maximum edge weight of the path between the two points. If the amount of data is small, you can search queries directly. But in fact, if you consider the reality, the general amount of data will be large, which will lead to the branch chain of the tree is relatively long. In this case, the time complexity of brute force query is high. Therefore, binary lifting with LCA to improve query efficiency.

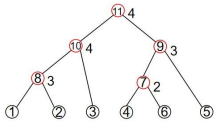
Suppose we want to solve the parent node of two points u and v.

The LCA violence method is that we always jump up the deeper one step until the depth of u and v is equal for the first time, and then jump up the two points at the same time until their parent nodes are the same.

The binary lifting with LCA method is actually to jump as much as possible in each step on the basis of the violence method, and its idea is actually consistent with the idea of dichotomies. The following is the approximate process of binary lifting with LCA method:

First, using recursion, record the first, second, fourth, eighth, ..., 2^n parent nodes of each node in the reconstructed tree from top to bottom. As shown in the Figure 3.

Then, calculate the depth difference between the two requested nodes u and v, denoted as dep = deep(u) - deep(v). Convert dep into a sum of 2^n numbers, i.e., dep = 2^0 + 2^1 + ... + 2^n. After that, make the node with a deeper depth jump up sequentially by 2^0, 2^1, ..., 2^n. This process can be quickly achieved with the help of the parent node table obtained from Figure 3. For example, if the two requested nodes are 2 and 9, and dep = deep(2) - deep(9) = 2 = 2^1, referring to Figure 3, we can directly get that node 2 jumps to node 10. This way, the depths of the two nodes are made equal.



current node \ parent node	2^0	2^1	.....
1	8	10	
2	8	10	
3	10	11	
4	7	9	
5	9	11	
6	7	9	
7	9	11	
8	10	11	
9	11		
10	11		
11			

Figure 3. Reconfiguration Tree

Finally, when the depths of the two nodes are equal, we obtain the current depth dep = deep(u) = deep(v), and decompose dep into a series of sums of 2^n numbers, for example, dep = 2^0 + 2^1 + ... + 2^n. Iterate i from n to 0. If the depth of the current node is greater than 2^i, and the two nodes are not the same node after jumping up 2^i steps, make the two nodes jump up 2^i. Otherwise, decrement i. In this process, we can use the parent node table obtained from Figure 3 to quickly find the 2^i-th parent node of each node. After completing the loop, the parent node of the current two nodes is their LCA [9]. Taking nodes 2 and 6 in Figure 3 as an example, dep = 4 = 2^2. The values of i are 2, 1, and 0 in sequence. When i is 2 or 0, the conditions are not met. When i is 1, nodes 2 and 6 both jump up 2^1 steps. According to the parent node table, node 2 jumps to node 10, and node 6 jumps to node 9. At this point, their parent node is node 11, so the maximum value of the path edge weights between nodes 2 and 6 is the data contained in node 11, which is 4.

4. Practical Application and Analysis

4.1 Problem Description

A has n cities, numbered from 1 to n, with m two-way roads between them. Every road has a weight limit for vehicles, referred to as the weight limit. There are now q trucks transporting goods, and drivers want to know the maximum weight each vehicle can carry without exceeding the vehicle weight limit.

4.2 Input Format

The first line has two integers n, m separated by A space, indicating that there are n cities and m roads in A.

Next line m contains three integers x, y, z, separated by a space between each of the two integers, indicating that there is a road from city x to city y with a weight limit of z. Attention :x! =y, there may be multiple roads between the two cities.

The next line has an integer q, indicating that q trucks need to deliver goods. Next line q, two integers x in each line, separated by a space between y, indicating that a truck needs to transport goods from City x to City y, guaranteeing x! =y.

4.3 Output Format

There are q rows, each with an integer indicating the maximum load for each wagon. If the truck does not reach its destination, output -1.

4.4 Scope of Data

For 30% of the data, 1<=n<1000, 1<=m<10000, 1<=q<1000.

For 60% of the data, 1<=n<=1000, 1<=m< 5000000, 1<=q<1000.

For 100% data, 1<=n<=10000, 1<=m< 50000000, 1<=q<30000.

4.5 Problem Analysis

This problem is a typical reconstructed tree problem, q trucks are q queries.

If you do not use the reconstructed tree for optimization, but run a search query directly, then each query needs to traverse all nodes at least once, with a time complexity of O(n). The time complexity [10] is q*O(n).

If a refactoring tree is built and optimized using a doubling LCA, the time complexity per query is roughly log2(n). The time complexity is q*log2(n). Compared with direct search, the speed of query is greatly improved by building reconstructed tree for query.

5. Conclusion

As society continues to advance, the landscape of collegiate programming competitions reflects a corresponding rise in complexity and sophistication. In this evolving environment, contestants encounter increasingly intricate challenges, with graph theory and tree-based problems occupying prominent positions among them. Mastery of the refactoring tree emerges as a pivotal skillset, offering contestants a competitive edge and equipping them with the tools necessary to navigate the complexities of modern problem sets. By adeptly leveraging the capabilities of refactoring trees, contestants can not only distinguish themselves within the competitive arena but also gain a decisive advantage in resolving path-edge weight restrictions within graphs. The versatility and efficiency of refactoring trees enable contestants to streamline their problem-solving workflows, facilitating rapid and accurate computations while enhancing code readability and maintainability. Furthermore, proficiency in refactoring trees transcends the confines of competition, empowering individuals to excel in diverse real-world scenarios and contribute meaningfully to the advancement of computer science and software engineering. As such, the acquisition and refinement of skills related to refactoring trees represent not only a strategic imperative for success in collegiate programming contests but also a pathway to continued growth and innovation in the broader field of technology.

References

[1] Blum J Competitive programming participation rates: an examination of trends in US ICPC regional contests. Discover Education, 2023, 2(1): 11.

[2] Nadimi-Shahraki M H, Zamani H, Asghari Varzaneh Z, et al. A systematic review of the whale optimization algorithm: theoretical foundation, improvements, and hybridizations. Archives of Computational Methods in Engineering, 2023, 30(7): 4113-4159.

[3] Paiva B, Manrique I, Dimopoulos M A, et al. MRD dynamics during maintenance for improved prognostication of 1280 patients with myeloma in the TOURMALINE-MM3 and-MM4 trials. Blood, 2023, 141(6): 579-591.

[4] Pham T T, Nguyen L L P, Dam M S, et al. Application of edible coating in extension of fruit shelf life. Agri Engineering, 2023, 5(1): 520-536.

[5] Qiao H, Wu Y X, Zhong S L, et al. Brain-inspired intelligent robotics: Theoretical analysis and systematic application.

Machine Intelligence Research, 2023, 20(1): 1-18.

[6] Khan S, Khan M A, Khan M, et al. Optimized feature learning for anti-inflammatory peptide prediction using parallel distributed computing. Applied Sciences, 2023, 13(12): 7059.

[7] Situmorang Y M, Mansyur A. Pengoptimalan Jaringan Pipa Primer PDAM Tirtanadi Cabang Tuasan Dengan Menggunakan Algoritma Kruskal. Jurnal Riset Rumpun Matematika dan Ilmu Pengetahuan Alam (JURRIMIPA), 2023, 2(2): 225-237.

[8] Labbé M, Landete M, Leal M. Dendrograms, minimum spanning trees and feature selection. European Journal of Operational Research, 2023, 308(2): 555-567.

[9] Wan G, Du B, Pan S, et al. Reinforcement learning based meta-path discovery in large-scale heterogeneous information networks. Proceedings of the aaai conference on artificial intelligence. 2020, 34(04): 6094-6101.

[10] Shah B, Bhavsar H. Time complexity in deep learning models. Procedia Computer Science, 2022, 215: 202-210.

