Distributed Edge Computation for Finding an Optimal Meeting Location

Matthew Chou
San Jose, CA USA
matthewzchou@gmail.com

***Abstract*—We consider the problem of finding a "fair" meeting place when S people want to get together. Specifically, we will consider the cases where a "fair" meeting place is defined to be either 1) a node on a graph that minimizes the maximum time/distance to each person or 2) a node on a graph that minimizes the sum of times/distances to each of the sources. In graph theory, these nodes are denoted as the center and centroid of a graph respectively. In this paper, we propose a novel solution for finding the center and centroid of a graph by using a multiple source alternating Dijkstra's Algorithm. Additionally, we introduce a stopping condition that significantly saves on time complexity without compromising the accuracy of the solution. The results of this paper are a low complexity algorithm that is optimal in computing the center of S sources among N nodes. Furthermore, we show how the algorithm can be distributed between S edge devices in a wired/wireless network.**

I. Introduction

There exist many real world problems where finding the center of a graph is deemed useful where the center can be defined in many different ways. As an example, in the field of computer networks, one is often interested in finding the center of nodes in a network to determine where to place servers [11]. Another example is the facility location problem [10] where one wants to build a set of facilities that lies within the center of a set of nodes where each node may represent residential homes, businesses, etc. In this paper, we want to examine a problem where several people are interested in meeting at a common location. This location should be "fair" in the sense that it either minimizes the maximum time/distance or the total time/distance. Solving this problem amounts to finding the center or centroid of the nodes that represent the people in the graph. What distinguishes this problem from the aforementioned problems is that the previous problems are solvable offline whereas in our scenario, there may be a need to solve it continuously as the ideal location of the center may vary with time. As a result, we need an efficient solution. Moreover, the people may be represented as edge nodes of a wired/wireless network. It is therefore desirable to find an algorithm that can be either 1) distributed across the edge nodes or 2) run at a central server where the results are then distributed to the edge nodes.

The contributions of this paper are as follows

- We construct the problem of finding the center and centroid of S sources among N nodes where $S << N$ and call it the S-source-center problem
- We propose a solution using a multiple source Dijkstra's Algorithm [3] or A* Algorithm [7]
- We propose a stopping condition for our algorithm that is optimal for finding the center and near optimal for finding the centroid while providing significant complexity savings
- We propose a method for distributing our algorithm between S edge devices.

We start by reviewing related work followed by describing our algorithmic framework, and finish by describing distributed edge device computing and simulation results.

II. Related Work

The previous problems of the K-Center [2], K-Median [6], and Jordan Center [9] all have solutions [12] which are different than ours. Both the K-Center problem and the K-median problem do not have any practical application to the S-source-center problem we proposed. The Jordan center requires the consideration of all nodes whilst the S-source-center problem only requires the consideration of the S source nodes. The common approach to the Jordan center problem is the Floyd-Warshall [4] algorithm which solves the problem in O(V^3) time. On the other hand, our algorithm is comprised of S alternating Dijkstra's Algorithms [3] from each of the S sources. Each Dijkstra's Algorithm has time complexity of $O((V + E)log(V))$ given a binary min-heap implementation [5]. Therefore, by alternating S Dijkstra's Algorithms, the time complexity is $O(S(V + E)log(V))$. Furthermore, we propose a stopping condition so that not all vertices, V, need to be explored in each of the S Dijkstra's Algorithm, leading to significant time complexity savings without compromising accuracy.

III. Algorithmic Framework

The two different problems of finding the centroid and center of the S-source-center problem have different objectives, requiring different algorithms for each one. In the following subsections, we will present both problems as well as a solution and an example for each one. In our algorithms, the reference to distance in a graph may represent either distance, time, or some other measure. Our algorithm works for both undirected and directed graphs with positive weights.

979-8-3315-2096-0/25/$31.00 ©2025 IEEE

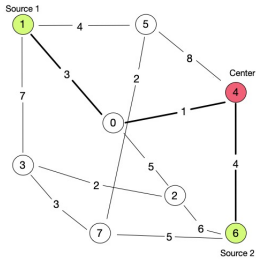

Fig. 1. An example for finding the center where the source nodes are located at Node 1 and Node 6. Both of the sources take turns running iterations of Dijkstra's Algorithm and find Node 4 to be the node that minimizes the maximum distance (i.e., 4)

A. Center

To find the center of the S-source-center problem we have to solve the following optimization problem:

$$\hat{v} = \arg\min_{v_i \in V} \max_{s_j \in S} d(v_i, s_j) \tag{1}$$

where V is the set of all vertices within the graph and S is the set of all source nodes.

We want to determine the node $\hat{v}$ with the minimum value of $\mathrm{f}(v_j) = \max_{s_j \in S} d(v_i, s_j)$. The proposed solution [1] is to run a Dijkstra's Algorithm [3] in alternating steps from each source node and store the shortest path for every node. By storing the shortest path from every node to every source node, one is able to determine the $\mathrm{f}(v_j)$ value for all nodes by keeping track of the maximum shortest path to all source nodes. The algorithm will also maintain a separate priority queue for each source node and alternate between the queues after a node has been extracted from a queue. This is done so that the algorithm switches between source nodes while exploring. Similar to our algorithm is the Bidirectional Dijkstra's Algorithm [8], but unlike our algorithm, Bidirectional Dijkstra's Algorithm is used to find the shortest path between two nodes. Our proposed algorithm is shown in Algorithm 1. To illustrate how our algorithm works, we provide an example (see Figure 1) where our algorithm will find Node 4 to be the center with a maximum distance of 4.

We can improve upon our algorithm by adding a stopping condition that performs an additional check upon finding the first intersection node . After finding the first intersection node v_u (i.e. an extracted node that has been visited by all sources), a variable called *minimax* is assigned the value of $\mathrm{f}(v_u)$. The algorithm then continues the alternating Dijkstra's Algorithm but with an additional check whenever another intersection node is encountered. If the next intersection node discovered has a maximum distance value that is less than *minimax*, then *minimax* is updated to equal the node's maximum distance. Furthermore, we claim that a Dijkstra's Algorithm for a given source may be terminated if an extracted node has a distance larger than *minimax*. The following Theorem makes this claim formal.

Algorithm 1 Multiple Source Dijkstra's Algorithm for Finding the Center

1) Initialization: Create a priority queue for each source node and initialize the distance of source nodes to 0 and all the other nodes to ∞
2) Selection: Pick the unvisited node with the smallest d(s_i,v) (initially the source node) and extract node
3) Relaxation: For the extracted node, check all neighboring nodes and compare/update its distances. If extracted node has been visited by all sources, keep track of its maximum distance to all source nodes
4) Alternation: The extracted node is marked as visited and will not be visited again. Repeat steps 2-3 alternating between each of the sources, for all nodes
5) Stopping: Find the node with the minimum eccentricity to all nodes. Alternatively, the minimum eccentricity could be kept track of whenever a node is extracted and has been visited by all sources.

Theorem. *Assume that there are S Dijkstra's Algorithms initiated from S different source nodes. Once a node has been visited by all S Dijkstra's Algorithms, then let these nodes be called $intersection$ nodes and let d_{max} represent the maximum distance to all source nodes from one of the intersection nodes. Then, for any of the S Dijkstra's Algorithms, if a node is extracted from its priority queue with a distance that exceeds d_{max}, then all remaining nodes that are to be explored for that Dijkstra's Algorithm cannot result in a smaller maximum distance than d_{max}.*

Proof. Dijkstra's Algorithm extracts nodes in a non decreasing manner, implying that every extracted node's distance cannot be smaller than a previously extracted node's distance. If an extracted node has a distance, d, larger than d_{max}, then any extracted node after that will have a distance larger than d and therefore larger than d_{max}. Therefore, any maximum distance that is calculated based on remaining nodes cannot result in a smaller maximum distance than d_{max}. □

Given the above Theorem, we know that if an extracted node has a distance greater than *minimax*, then the extracted node and all successive nodes will result in a maximum distance larger than *minimax*. Therefore none of these nodes can be a center. Our improved stopping condition for each Dijkstra's Algorithm is to check if an extracted node has distance larger than *minimax* and if it does, then terminate that Dijkstra's Algorithm. Our improved algorithm with a modified stopping condition is described in Algorithm 2:

For the same example that we considered above (see Figure 1), our improved algorithm finds the optimal node (Node 4) by exploring only 50 % of the nodes (i.e. 8 total nodes).

B. Centroid

In some scenarios, one may wish to find a solution that minimizes total time/distance to a common location instead of

Algorithm 2 Multiple Source Dijkstra's Algorithm for Finding the Center with an Improved Stopping Condition

1) Initialization: Same as Algorithm 1. Set $minimax$ to ∞
2) Selection: Same as Algorithm 1
3) Relaxation: For the extracted node, check all neighboring nodes and compare/update its distances. If extracted node has been visited by all sources, compare its maximum distance to $minimax$. If it is less than $minimax$, then set $minimax$ to this maximum distance.
4) Alternation: The extracted node is marked as visited and will not be visited again. If the extracted node has a distance that is larger than $minimax$ then go to step 5. Otherwise, repeat steps 2-3 alternating between each of the sources that have not been terminated, for all nodes.
5) Stopping: If all sources have been terminated then the center is the node with value $minimax$ and $minimax$ is the smallest maximum distance. Otherwise, repeat steps 2-3 alternating between each of the sources that have not been terminated, for all nodes.

minimizing the maximum time/distance to a common location. This is equivalent to finding the centroid of the S-source-center problem. To find the centroid of the S-source-center problem we have to solve the following optimization problem:

$$\hat{v} = \arg\min_{v_i \in V} \sum_j d(v_i, s_j) \tag{2}$$

where V is the set of all vertices within the graph and $s_j \in S$ represents the jth source node with S representing the set of all sources.

Similar to finding the center of the S-source-center problem, we can apply an algorithm based on a multiple source Dijkstra's Algorithm. Instead of optimizing for the function $f(v_i)$ we instead optimize for $g(v_i)$ where $g(v_i) = \sum_j d(v_i, s_j)$. The node with the minimum value of $g(v_i)$ is the centroid. The algorithm follows the same steps as section III-A, except we keep track of the sum of distances to each source.

We can apply a similar algorithm as Section III-A for finding centroid except the theorem that we used to create the stopping condition for finding the center does not directly apply for finding the centroid. We can still however use the theorem as a heuristic to find a relative minimum of the sum of distances to the sources. Specifically, we can check for any intersection node whether the sum of distances is larger than the previously stored minimum sum of distances. If it is, then we can update this minimum sum of distances. Otherwise, we can terminate the Dijkstra's Algorithm that resulted in a larger sum of distances. Our new algorithm with a modified stopping condition is described in Algorithm 3.

As mentioned above, Algorithm 3 may end up finding a relative minimum instead of a global minimum. As an example, consider Figure 2 where the algorithm terminates at Node 3 with a sum distance of 13. However, there exists a smaller sum of 11 at Node 5.

Algorithm 3 Multiple Source Dijkstra's Algorithm for Finding Centroid

1) Initialization: Same as Algorithm 1. Set $minsum$ to ∞.
2) Selection: Same as Algorithm 1
3) Relaxation: For the extracted node, check all neighboring nodes and compare/update its distances. If extracted node has been visited by all sources, compare its sum to $minsum$. If it is less than $minsum$, then set $minsum$ to this sum.
4) Alternation: The extracted node is marked as visited and will not be visited again. If the extracted node has been visited by all sources and results in a sum that is larger than $minsum$ then go to step 5. Otherwise, repeat steps 2-3 alternating between each of the sources that have not been terminated, for all nodes.
5) Stopping: If all sources have been terminated then the centroid is the node with value $minsum$ and $minsum$ is the smallest sum. Otherwise, repeat steps 2-3 alternating between each of the sources that have not been terminated, for all nodes.

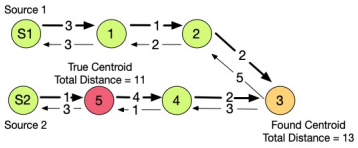

Fig. 2. A case in which the heuristic for finding the centroid fails to find the optimal solution

C. Solution Utilizing A* Algorithm

Similar to Dijkstra's Algorithm, we can adapt A* Algorithm for our problem. We can utilize the same algorithm as in Algorithm 1 except we will run A* algorithm from each source node. Since A* algorithm without a heuristic function is essentially the same as Dijkstra's Algorithm, applying a heuristic function to A* algorithm will reduce the number of explored nodes without reducing accuracy.

A* algorithm maintains the same priority queue as Dijkstra's Algorithm, so we are able to alternate between sources to have an alternating A* algorithm. Note that the stopping condition derived from our theorem may not be optimal when applied to A* algorithm.

IV. Distributed Computation

In this section, we want to map our algorithm to a computer network where the S sources represent S edge devices and there exists a server that contains real time information about the graph. One method would be to send the locations of each of the S sources to the server and then have the server compute the center or centroid to send back to each of the S sources. Another method would be to distribute the computation among the edge devices by having each device run their own Dijkstra's Algorithm and then send the computed nodes and their distances back to the server. The server would

then decide whether that source's Dijkstra's Algorithm should be terminated. The protocol (see Figure 3) for distributing the computation between S devices is as follows:

- Each of the edge devices runs n iterations where n can be any number from 1 to N. Each edge device then sends the n nodes and n distances to the server
- After the server receives the information from each of the S sources that have not been terminated, it will update $minimax$ (or $minsum$) and decide whether a source should be terminated based on Algorithm 2 (or Algorithm 3). A bit, isTerminate, is sent back to each source to indicate whether that source should be terminated

The above protocol is ended when all sources are terminated. The server will then send the center (or centroid) to all edge devices. Note that each source will run n iterations at a time where n may be larger than 1. The reason is to reduce the overhead of sending nodes and distances to the server by bundling n of them together. We assume that the edge weights for the map do not change from the time that the sources start calculating its path until the time that the optimal meeting location is calculated. There may be delays if one device has much larger computation or transmission time compared to the other devices. If the computation and transmission times are assumed to be roughly symmetric across devices then delays should be minimized. As the number of source nodes increases, the likelihood of one of the source nodes having significantly larger computation or transmission times increases, increasing the potential for larger delays. For a very large number of sources, it may be beneficial for the server to do all of the computations to avoid large delays, but this would imply more computation for the server.

There exists a possibility that during the above protocol, 1) one or more source nodes request to join the session or 2) one or more source nodes disconnect from the session. In the first case, the protocol should be reset to account for the new source nodes and all source nodes will start calculations from the beginning. If the protocol is not reset, then our algorithm may not work properly as marked intersection nodes may not be actual intersection nodes. In the second case, the protocol does not necessarily need to be reset as the server contains enough information to be able to determine intersection nodes with fewer sources along with determining which of the remaining sources may be terminated.

V. Experimental Setup

To test the algorithms described in the previous sections, we wrote C code to test the percent of nodes explored by using Algorithm 2 for finding the center algorithm as well as the accuracy of the percent of nodes explored by using Algorithm 3 for finding the centroid (see section III-B). We also measure the accuracy of our algorithm for finding the centroid node. We wrote C code to randomly generate graphs. For a fixed number of vertices and source nodes, our program would randomly generate directed edges with random weights between 1 and 100. The program would also randomly choose n vertices within the graph as source nodes. We discounted

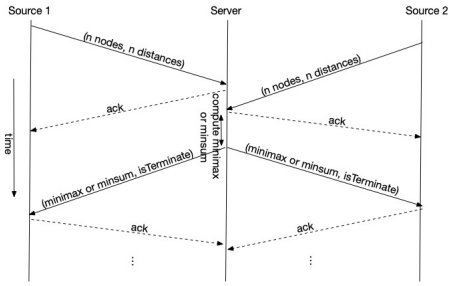

Fig. 3. An example of two client side sources sending their data to a server that computes either the $minimax$ or $minsum$

any graphs where the graph was disconnected and were unable to reach any intersecting nodes. We chose to test 2, 3, 5, and 10 sources. Additionally, we utilized a wide range of vertices (20, 50, 100, 500) to try and capture both small and large graphs. Running 1000 iterations for each combination of source nodes (2, 3, 5, 10) and vertices (20, 50, 100, 500) yielded results spanning both sparse and dense graphs. We did not run simulations with the A* algorithm.

VI. Experimental Results

Number of Source Nodes	Number of Vertices			
	20	50	100	500
2	28.484	18.643	13.993	7.948
	14.61	10.74	9.50	4.92
3	42.059	30.736	25.614	18.473
	14.81	11.27	11.09	8.60
5	56.209	45.071	40.852	35.130
	13.51	12.43	11.68	10.98
10	70.214	61.116	57.771	55.757
	11.74	11.35	11.11	12.41

TABLE I
Average (top) and standard deviation (bottom) of percentage of nodes explored for minimum of maximum distances

In our first experiment, we tried finding the center of various graphs using Algorithm 2 and compared it to Algorithm 1. We compared the percentage of nodes explored using Algorithm 2 versus the number of nodes explored in Algorithm 1 (see Table I). In Table I, we see that the average percentage of nodes explored is directly proportional to the number of source nodes. The reason for this is because as the number of source nodes increases, the likelihood of one of the source nodes being further away from the center increases, hence resulting in more nodes being explored before an intersection node is even found. On the other hand, the average percentage of nodes explored is inversely proportional to the number of vertices due to our stopping condition. Specifically, our stopping condition allows for efficient termination once an intersection node is found and a larger number of vertices implies a larger number of edges which increases the likelihood of shorter paths to intersection nodes. As a result, fewer nodes need to be explored for a graph that has more vertices (e.g. 500)

with random edges than a graph with less vertices (e.g. 20). We notice anywhere from a 1.5x to a 12x savings in nodes explored depending on variations in the amount of vertices and number of nodes in the simulation.

Number of Source Nodes	Number of Vertices			
	20	50	100	500
2	35.343	21.790	15.785	7.320
	15.54	10.16	9.84	2.40
3	49.674	33.780	26.541	16.002
	13.15	8.07	8.05	3.55
5	65.203	48.325	40.518	29.500
	9.80	8.31	7.77	4.25
10	81.100	66.013	57.088	47.448
	6.68	6.48	6.25	5.57

TABLE II
AVERAGE (TOP) AND STANDARD DEVIATION (BOTTOM) OF PERCENTAGE OF NODES EXPLORED FOR MINIMUM OF SUM DISTANCES

The next experiment that we ran was to find the centroid of various graphs using Algorithm 3 and compare it to Algorithm 1. We compared the percentage of nodes explored using Algorithm 3 to the number of nodes explored using Algorithm 1 (see Table II). In Table II, like Table I, we can draw many of the same conclusions as the previous experiment because Algorithm 3 maintains many of the same aspects as Algorithm 2. However, we do see that the averages in Table II are almost always larger than the ones in Table I. This is due to the fact that in Algorithm 3, nodes must be marked as visited by all sources before comparing a sum for termination whereas in Algorithm 2, a node does not need to be visited by every source to check the termination condition. This stricter termination condition results in more nodes being explored in the case of the centroid algorithm. Similar to the previous experiment, we notice a savings in nodes explored of around 2x to 12x.

Number of Source Nodes	Number of Vertices			
	20	50	100	500
2	77.836	68.962	63.333	63.948
3	78.340	67.598	61.663	55.791
5	84.008	74.948	65.854	48.036
10	91.67	83.874	77.324	52.367

TABLE III
PERCENT ACCURACY OF CENTROID LOCATING ALGORITHM

Finally, we also measure the percentage of times that Algorithm 3 finds the true centroid (see table III). We notice that the accuracy decreases as the number of vertices increases and increases as the number of source nodes increases. This is consistent with our previous observations. As the number of source nodes increases, the number of nodes explored also increases, resulting in a higher likelihood of finding the true centroid. Similarly, as the number of vertices increases, the number of nodes explored decreases, resulting in a lower likelihood of finding the true centroid.

Comparing to traditional algorithms like Floyd Warshall, our algorithm with and without a stopping condition outperforms the Floyd-Warshall algorithm. The Floyd-Warshall algorithm does unnecessary exploration of all $N-1$ nodes, when only exploration of S nodes is needed [4]. As a result, our algorithm achieves an additional $N-1/S$ savings over the Floyd Warshall algorithm. Our stopping condition can achieve up to $(N-1/S)*12$ savings of explored nodes.

In the above, the computational savings may not always be equal across the source nodes. For example, when the center/centroid is near the first intersection node, all sources will experience roughly the same amount of savings. However, when the center/centroid is far from the first intersection node, the computational savings may be asymmetric across the source nodes.

VII. CONCLUSION

We proposed solutions to the problem of finding the center of S sources by utilizing S different Dijkstra's Algorithms for both the centroid and center. Furthermore, we proposed optimizations to reduce the amount of exploration of said algorithm by a factor of 2x to 12x. We also provided an optimal solution for finding the center and centroid of a graph utilizing S Dijkstra's Algorithms and a stopping condition that significantly reduces the time complexity. Finally, we showed how our algorithm can be used in mapping applications where computation may either be distributed across S edge devices or computed at a central location.

Directions for future work are to improve the accuracy of our solution to finding the centroid and possibly find a more efficient way to distribute computation.

REFERENCES

[1] Matthew Chou. Finding the center and centroid of a graph with multiple sources. *arXiv:2408.13688v1 [cs.DM]*, 2024.

[2] Mark S. Daskin. *Network and Discrete Location: Models, Algorithms, and Applications*. Wiley, 1995.

[3] Edsger W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[4] Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

[5] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[6] N. K. Gupta and P. K. K. Ghosh. The k-median problem. *Journal of Algorithms*, 29(2):331–351, 1998.

[7] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4(3):100–107, 1968.

[8] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A faster algorithm for finding the shortest paths. *Journal of the ACM*, 16(2):235–239, 1969.

[9] J. Jordan. On the theory of the center problem in networks. *Mathematical Programming*, 1(1):55–68, 1971.

[10] Richard M. Karp. An algorithm for the facility location problem. *Networks*, 2(3):211–216, 1972.

[11] H. J. Korte and J. Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer, 2008.

[12] Mikkel Thorup. Quick k-median, k-center, and facility location for sparse graphs. In *ICALP '01: Proceedings of the 28th International Colloquium on Automata, Languages and Programming*, volume 2076, pages 249–260, London, UK, 2001. Springer-Verlag.