Optimization of Travelling Cost through a Vertex-Weighted Undirected Graph

Jade Cheng

October 2008

University of Hawai'i at Mānoa
Department of Information and Computer Sciences
2500 Campus Road
Honolulu, HI 96822

Optimization of Travelling Cost through a Vertex-Weighted Undirected Graph

Jade Cheng

Department of Information and Computer Sciences
University of Hawai'i at Mānoa
Honolulu, HI 96822

yucheng@hawaii.edu

October 21, 2008

Abstract

The paper is concerned with the management of travelling to a single-source vertex with an optimized cost through a vertex-weighted undirected graph. We provide a comprehensive problem formulation, algorithm design, and complexity analysis for this task. We prove the correctness of applying a Breadth First Search algorithm, and we further optimize the algorithm by identifying and eliminating vertices for which the root selection is inappropriate. After optimization, the average run-time is reduced, but the worst case run-time, however, still exists as $O(V \times E)$. When the worst case occurs, the algorithm steps through all vertices to examine their single-source-shortest-path trees and minimizes the overall visiting cost.

1 Introduction

The paper is concerned with the management of travelling to a single-source vertex with an optimized cost through a vertex-weighted undirected graph. This is a common investigation in real life, not specific to any particular organization, and many engineers have to manage this issue. For example, the single source vertex could be a possible location for a residential area; the other vertices in the graph could be the possible locations for commercial facilities. Each facility has a different frequency of visitation. Developers would need to investigate the cost of traveling from the residential areas to the commercial facilities before they could make any decisions regarding future investments. They might need to optimize the travelling cost by selecting a residential location that is fairly close to most facilities. They could also optimize the road connections between places. There might be a set of possible tracks where the roads could be built. The investigators would need to decide which roads to construct to optimization travelling efficiency.

Hereon we present a mathematical description of the challenge presented above as a problem formulation, algorithm design, and complexity analysis. We make the assumption that a Decision Researcher with a strong mathematical background is the target reader.

2 Preliminaries

2.1 Problem Formulation

We investigate the input and output of this problem set. It is clear this problem falls into the category of optimization problem. In mathematics and computer science, an optimization problem is the problem of finding the best solution from all feasible solutions. In mathematics, in particular, the term optimization refers to the study of problems in which one seeks to minimize or maximize a real function by systematically choosing the values of real or integer variables from within an allowed set.[1]

An optimization problem can be represented in the following way:

Given: A function $f: A \to R$ from some set A to the real numbers.

Sought: An element x_0 in A such that $f(x_0) \leq f(x)$ for all x in A (minimization).

In our problem set, the expression could be written as:

Given: A function $f(T,r) = \sum q(v)\,|p(r,v)|$, where $G = (V,E)$ is a given undirected graph, $v \in V$ is associated with a weight $q(v) \in N$ that represents a frequency of visiting v, $T = (V',E')$ is a sub-graph of G, $V' \subseteq V$, and $E' \subseteq E$, $p(r,v)$ denotes the path from the root r to vertex $v \in V'$ in T, and $|p(r,v)|$ denotes the length of the path $p(r,v)$ from r to v in T.

[1] http://en.wikipedia.org/wiki/Optimization_problem

Sought: A set of edges and vertices (V_{min}, E_{min}) form a tree structure $T_{min} = (V_{min}, E_{min})$ $G(V, E)$, and the tree structure is rooted at the vertex r_{min} such that the function $f(T_{min}, r_{min}) = \sum q(v_{min})|p(r_{min}, v_{min})| \leq f(T, r) = \sum q(v)|p(r, v)|$, and all r in V (minimization).

Based on the description above, the final optimal solution is built upon the best solutions of all $f(T_{min}, r) = \sum q(v_{min})|p(r, v_{min})|$. Once the best solutions of all root cases in a particular graph G are found, a simple comparison of all these solutions gives the overall best solution of the original problem. Therefore, the *Sought* section above could be achieved by iteratively examining every vertex as the root. Hence, the *Sought* section of the problem formation could be further written as the following.

Sought: A set of edges and vertices (V_{min}, E_{min}) form a tree structure $T_{min_i} = (V_{min}, E_{min})$ in graph $G = (V, E)$, such that the function $f(T_{min_i}, r_i) = \sum q(v_{min})|p(r_i, v_{min})| \leq f(T, r_i) = \sum q(v)|p(r_i, v)|$ for all T in G, and a particular r_i in V, where $i = 1, 2, 3, \dots, V$ (minimization).

2.1.1 Problem formation for the branch problem

We break down the overall optimization problem into branches problems. The branch problem is described at the end of the previous section. We write it down in a formal form:

$$\begin{aligned} f(T_{min_i}, r_i) &= \sum q(v_{min})|p(r_i, v_{min})| \\ &= q(v_1)|p(r_i, v_1)| + q(v_2)|p(r_i, v_2)| + \cdots + q(v_{V-1})|p(r_i, v_{V-1})| \end{aligned} \quad (1)$$

In equation (1), the values of $q(v_j)$ are givens, where $j = 1, 2, 3, \dots, V - 1$. We assign the root vertex with an index of $j = 0$. This is based on the problem formation in the previous section. Each vertex $v \in V$ is associated with a weight $q(v) \in N$ that represents a frequency of visiting v, where N denotes the set of natural numbers.

If there exists a minimal value of $|p(r_i, v_1)|$, then we can calculate the minimal value of the first component in equation (1), $q(v_1)|p(r_i, v_1)|$. If there exists a minimal value of $|p(r_i, v_2)|$, then we can calculate the minimal value of the second term in the equation, $q(v_1)|p(r_i, v_2)|$, and so on.

Finding the individual minimal value of $|p(r_i, v_1)|$ is a problem know as the shortest-path problem of an unweighted, undirected graph, with a given root r_i. An unweighted, undirected, shortest path from u to v is a path of minimum weight from u to v. The shortest-path weight from u to v is defined as $\delta(u, v) = \min\{w(p) : p \text{ is a path from } u \text{ to } v\}$, where the edge weight is 1 for all edges.[2] Therefore, the minimal value of $|p(r_i, v_1)|$ could be expressed as $\delta(r_i, v_1)$. The same logic

[2] http://en.wikipedia.org/wiki/Shortest_path_problem

applies to all of the individual minimal values of $|p(r_i, v_1)|, |p(r_i, v_2)|, \dots,$ and $|p(r_i, v_{V-1})|$. They could be expressed as $\delta(r_i, v_1), \delta(r_i, v_2), \dots,$ and $\delta(r_i, v_{V-1})$ individually.

In order to consider the overall optimal solution of equation (1), it would be nice if we could directly use the optimal solutions for all individual components, $q(v_1)|p(r_i, v_1)|$, $q(v_2)|p(r_i, v_2)|, \dots,$ and $q(v_{V-1})|p(r_i, v_{V-1})|$. Proving two assumptions suffices for this purpose.

1. The individual minimal solutions for all components for equation (1) occur at the same time under the same conditions. When equation (1) is optimized, all of its components are optimized at the same time as well. No vertex needs to sacrifice its own shortest path for the overall optimization.

2. If the first assumption is proved, the output graph T_{min_i} is still a connected graph and forms a spanning tree structure.

If both of these assumptions are proved, the individual optimal solutions must always lead to the overall optimal solution for equation (1). As discussed in the previous paragraphs, for each component, minimizing $q(v_j)|p(r_i, v_j)|$ is the same problem as looking for $\delta(r_i, v_j)$. The entire set of problems is converted to finding $\delta(r_i, v_1)$, $\delta(r_i, v_2), \dots,$ and $\delta(r_i, v_{V-1})$.

Looking for $\delta(r_i, v_1)$, $\delta(r_i, v_2), \dots,$ and $\delta(r_i, v_{V-1})$ is a problem know as the unweighted-undirected-single-source-shortest-path tree problem, where $G = (V, E)$ is a given connect graph, $\delta(u, v) = \min\{w(p) : p \text{ is a path from } u \text{ to } v\}$, and r_i is a given root. In this kind of problem, we find shortest paths from a source vertex v to all other vertices in the graph. The source vertex for this particular problem is r_i.

2.1.2 Conclusions of problem formation

So far, we have divided the problem into sub-problems. In order to find the minimal solution among all vertices, we can examine the vertices in sequence. For each sub-problem, we further divide the problem into $V - 1$ components. Each component has a form of $q(v_j)|p(r_i, v_j)|$ where $i = 1,2,3, \dots, V, j = 1,2,3, \dots, V - 1$. Integer i is used to loop over all vertices and examine the best solution for graphs that are rooted at each of the vertices. Integer j is used to loop over all other vertices while examining a particular vertex as the root.

At this point, we need to prove two assumptions (discussed below). If we can prove the two assumptions, our problem is conceptually solved because there are existing, famous algorithms to solve single source shortest path problems with unweighted, undirected edges. For example, the breadth-first search (BFS) algorithm provides a solution for this problem with a runtime of $O(V + E)$. After applying the BFS, we simply need to loop over every vertex to examine the graphs rooted at particular vertices. This operation adds a term of $O(V)$. Therefore, conceptually, we have solved our naïve algorithm, which has a runtime of $O(V \times (V + E))$.

2.2 Properties of Optimum Solutions

Here we provide the proof for assumption 1, individual minimal solutions for all components for equation (1) occur at the same time under the same conditions.

The input is a connect graph $G = (V, E)$. We need to prove that any of the vertices' optimal solutions do not require other vertices to sacrifice their optimal choices. In other words, when we decide a certain path is the shortest path for a certain vertex v from the root r, we need to prove that all other vertices that are connected through this path also obtain their shortest paths at the same time.

To express this idea mathematically, we use the following expression. For all $u, v \in V$, where $p(r, v) = p(r, u) + p(u, v)$, $p(r, v)$ is given as the shortest path from r to v, we need to prove $p(r, u)$ and $p(u, v)$ are also the shortest paths from r to u, and from u to v, respectively. Without loss of equality, we can simply prove $p(r, u)$ is the shortest path from r to u.

Proof of Optimum Properties

Suppose we have a different path $p'(r, u)$ from r to u that provides a shorter path than $p(r, u)$. We derive an inequality $p'(r, u) + p(u, v) < p(r, u) + p(u, v)$. Since $p(r, u) + p(u, v) = p(r, v)$, we further derive the inequality $p'(r, u) + p(u, v) < p(r, v)$.

Figure 1. Optimal solution for vertex v.

Here, we observe a contradiction. The inequality $p'(r, u) + p(u, v) < p(r, v)$ conflicts with our assumption that $p(r, v)$ is given as the shortest path from r to v.

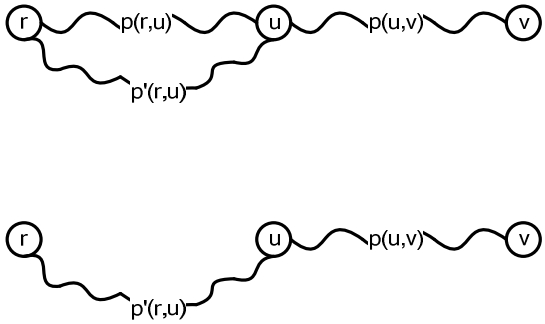

Figure 2. Alternative shortest-path for vertex u.

Therefore, we proved that the alternative path $p'(r, u)$ from r to u cannot provide any shorter path than $p(r, u)$. Therefore, $p(r, u)$ is the shortest path from r to u.

Consider the entire graph G, since any individual solution of the shortest path from the given root to a certain vertex does not require any other vertices to sacrifice their individual best solutions, we can safely say that the overall optimization is the combination of the individual optimizations for every vertex. Therefore, we have proved our first assumption.

2.3 Properties of Spanning Tree

Here we provide the proof for assumption 2. When the overall optimal solution of the first assumption occurs in equation (1), the output graph T_{min_i} is still a connected graph and forms a spanning tree structure.

The input is a connect graph $G = (V, E)$. We need to prove that after the shortest paths of all vertices are drawn in the graph G, the output sub-graph forms a spanning tree structure. In other words, we need to prove the properties of a spanning tree hold for this sub-graph.

In the mathematical field of graph theory, a spanning tree T of a connected, undirected graph G is a tree composed of all the vertices and some (or perhaps all) of the edges of G. Informally, a spanning tree of G is a selection of edges of G that form a tree spanning every vertex. That is, every vertex lies in the tree, but no cycles (or loops) are formed. On the other hand, every bridge of G must belong to T.[3] Based on this definition, in order to prove the sub-graph is a spanning tree structure, we need to prove all three of the properties hold.

Proof for Property 1: The sub-graph is a connected graph.

Suppose the output sub-graph is an unconnected graph. As we know the input of the problem is a connected graph G, so this change from connected graph to unconnected graph can happen only during the procedure of drawing the shortest paths. We take in input graph G. Based on the assumption 1, we put down the shortest paths for all vertices and delete all other edges that are not used in forming the network of the shortest paths.

This situation implies that we delete some certain edges in order to obtain the shortest path network from the given root to all other vertices.

Here we observe a contradiction. The value of $p(u, v) = \infty$, if it is unconnected. We cannot obtain this solution while looking for the shortest paths. As we proved earlier, the best solution of this problem contains the best solutions of its sub-problems. So regardless of whether or not $p(u, v)$ is a sub-path or a full path, this is a contradiction.

Therefore, we have proved the first property of a spanning tree. The output sub-graph is a connected graph.

Proof for Property 2: The sub-graph contains all vertices.

Suppose the output sub-graph does not contain all vertices. This is to say the output graph contains at least two sub-parts. This configuration is clearly a contradiction with the first property we just proved.

Therefore, we have proved the second property of a spanning tree. The output sub-graph contains all vertices of the input graph.

Proof for Property 3: The sub-graph does not contain cycles.

Suppose the output sub-graph contains at least one cycle. We denote the cycle is formed by $p(u, v), p(v, w),$ and $p(u, w)$, where $u,\ v,\ w \in V$, $p(u, v),\ p(v, w),\ p(u, w)$ are paths contained in the output sub-graph, and u, v, w are vertices within the existing cycle. For each of these three paths,

[3] http://en.wikipedia.org/wiki/Spanning_tree_(mathematics)

there might be other vertices on the same path, so the cycle could be formed with any number n of vertices, where $n \geq 3$.

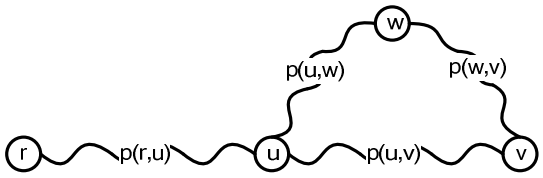

Figure 3. Configuration of cycles in sub-graphs of G.

Let us assume $p(r,u)$ and $p(r,v)$ are proved as the shortest paths as we described in the proof section for assumption 1. Therefore, from the perspective of the root vertex r, there are two ways to reach w, namely $p(r,u) + p(u,w)$ or $p(r,u) + p(w,v)$.

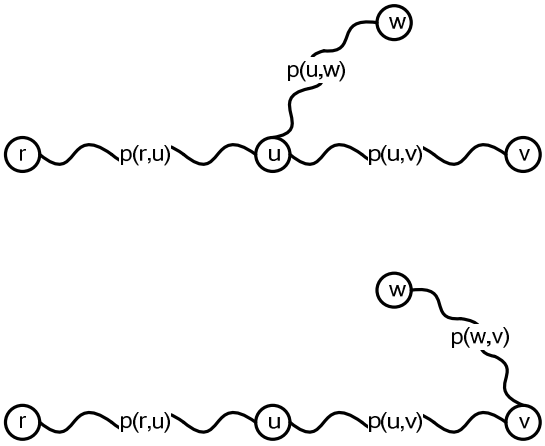

Figure 4. Multiple paths from root r to vertex w.

Here we observe a contradiction. We should have already deleted any multi-path options while investigating the shortest path from the root r to all other vertices. There should be only one path from the root to the certain vertex w, and this path is the shortest way to connect the root with this vertex. Even if path $p(r,u) + p(u,w)$ is the same length as path $p(r,u) + p(w,v)$, a correct algorithm of selecting the shortest path would randomly eliminate one of these two paths. If $p(u,w) > p(u,v) + p(w,v)$, the first path in the diagram above would be selected; otherwise the second path would be selected.

Therefore, we have proved the third property of a spanning tree. The output graph does not contain any cycles.

2.4 Preliminary Conclusions

Previously, we have proved our assumptions in the problem formulation, and we can now safely divide the problem into two parts.

1. We examine the vertices, one at a time, in order to find the graph rooted at a particular vertex. This graph provides the overall minimal solution.

   The overall minimal solution is a set of edges and vertices (V_{min}, E_{min}) that form a tree structure $T_{min} = (V_{min}, E_{min})$ in $G = (V, E)$, and the tree structure is rooted at the vertex r_{min} such that the function $f(T_{min}, r_{min}) = \sum q(v_{min})|p(r_{min}, v_{min})| \leq f(T, r) = \sum q(v)|p(r, v)|$ for all T in G, and all r in V.

2. We search for the single source shortest path for the graph rooted at a chosen vertex. In other words, we convert the problem of looking for the minimal solution for equation (1) into looking for the minimal solution for equation (2).

$$f'(T_{min_i}, r_i) = \sum |p(r_i, v_{min})| \\ = |p(r_i, v_1)| + |p(r_i, v_2)| + \cdots + |p(r_i, v_{V-1})| \tag{2}$$

   As we proved the correctness of this conversion, we can safely plug in the numbers of equation (2) into equation (1) once the solution of equation (2) is found.

We also examined the runtime of this naïve algorithm in the problem formulation section. Since we can solve the single-source-shortest-path problem of graphs with unweighted undirected edges at a runtime of $O(V + E)$ using well-known tree traversal algorithms, such as the BFS algorithm, we can further solve our problem with a runtime of $O(V \times (V + E))$.

3 Algorithm Design

3.1 Key Idea

As we discussed in the previous section, the problem is now divided into two parts. The parts are fairly independent from each other, and we proved the correctness of each of these two parts. Therefore, in order to improve the runtime of the overall algorithm, we need to consider two possibilities:

1. Improving run-time performance for the tree traversal algorithm.
2. Reducing the number of vertices to examine as root vertices.

3.1.1 Improving Run-Time Performance for the Tree Traversal Algorithm

This challenge is currently a well-known research topic. As we discussed earlier, there is simple linear-time algorithm for BFS traversal in a graphs using dynamic programming.[4] This algorithm keeps a set of appropriate candidate nodes for the next vertex to be visited in a FIFO queue. Furthermore, in order to discover the unvisited neighbors of a node from its adjacency list, it marks the nodes as either visited or unvisited.

[4] T. H. Cormen, C.E. Leiserson, and R.L. Rivest. Introduction to Algorithms. McGraw-Hill, 1990.

Ongoing research has concluded that, unfortunately, even a massive amount of RAM is insufficient to significantly increase performance for large graphs on a modern PC. Existing approaches are simply non-viable. The main cause for such poor performance of this algorithm on massive graphs is the amount of I/O that incurs. Remembering visited vertices needs $O(V)$ I/O in the worst case, and the unstructured indexed access to adjacency lists may result in $O(E)$ I/O. Researchers continue to work on improving the memory usage performance for BFS implementations.[5]

These are improvements, however, regarding the space needed for Memoization algorithms in dynamic programming. By improving the computational model, tree traversal implementations manage to optimize the usage of the memory space. But as the run-time for the algorithm, it is still linear, and that is probably the best algorithm for tree traversal so far.

Intuitively, to determine if an edge is needed in the final single source shortest path for an input graph, we need to access the edge at least once. This operation would simply give a linear time algorithm, let alone the comparisons and queue accessing operations.

Therefore, the tree traversal portion of the algorithm we propose runs in linear time applying the BFS tree traversal algorithm. For our run-time analysis, we do not consider the memory allocation requirement for a particular algorithm.

The naïve BFS algorithm runs at $O(V + E)$ for all kinds of graphs. For this implementation, only a connected graph is considered as valid input. We can, therefore, further differentiate the higher and lower terms in this analysis. As we know, for the worst case run-time analysis, the lower term is thrown away, and only the higher terms are kept in the final equation.

In this case, a connected graph requires the number of edges to be at least the number of vertices minus one. The input is also an undirected graph, so the upper bound of the number of edges is, therefore, quadratic to the number of vertices. This case occurs when the input graph is a complete graph. We can express this relationship in a mathematical form:

$$V \leq E \leq \frac{V \times (V - 1)}{2}$$

It is clear that V could be the lower term in the analysis $O(V + E)$. V could be at most the same as E. Ether V is a lower term of E or V is the same as E. V could be omitted in the final analysis. Therefore, we have a BFS tree traversal algorithm with a run time of $O(E)$ for any kind of connected graph input. Therefore our naïve algorithm could be written as $O(V \times E)$.

$$O(V + E) = O(E)$$
$$O(V \times (V + E)) = O(V \times E)$$

3.1.2 Reducing the Number of Vertices to Examine as Root Vertices

The run-time performance of our algorithm would improve if we could eliminate some vertices as possible candidates for the root. To observe an increase in performance, we would need to be able to

[5] Deepak Ajwani, Ulrich Meyer, and Vitaly Osipov. Improved external memory BFS implementations

examine and eliminate some vertices using an algorithm faster than $O(V \times E)$. If that is possible, we would, therefore, be able to improve our overall algorithm.

Based on these thoughts, we propose one possible way to improve this part of the algorithm. We do not have to examine the sub-problems that are rooted at leaf vertices, where the roots do not have the greatest vertex weights. If this proposal is proved to be correct, we can skip this kind of leaf vertices and examine only the remaining. In order to prove this proposal, we need to prove the following two assumptions:

1. A sub-graph rooted at a leaf vertex with a degree of one can never provide the overall minimal solution if its vertex weight is not the greatest in the input graph. This can be expressed in a mathematical form:

$$T_{min_{leaf}}(V_{min}, E_{min}) \neq T_{min}(V_{min}, E_{min})$$

$$\sum q(v_{min})\left|p\left(r_{leaf}, v_{min}\right)\right| \neq \sum q(v_{min})|p(r_{min}, v_{min})|$$

2. We can find and mark off these kinds of leaf vertices using a operations faster than $O(V \times E)$.

Proof for Assumption 1

Suppose we have a sub-graph $T'(V', E')$, where V' denotes the vertices and E' denotes the edges. Suppose T' is generated from a input graph $G = (V, E)$, where V denotes the vertices in G and E denotes the edges in G. Suppose $T'(V', E')$ and vertex r consist the solution of our problem for the input $G = (V, E)$. Suppose leaf vertex r is the root of tree $T'(V', E')$. We also suppose p_r is not the greatest vertex weight in all of the vertices.

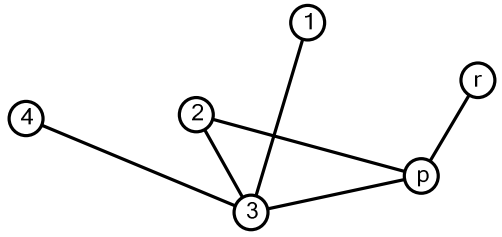

Figure 5. Input graph $G = (V, E)$.

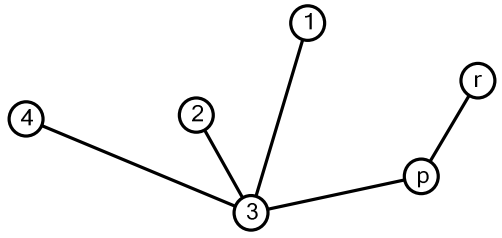

Figure 6. Solution graph $T'(V', E')$ rooted at vertex r.

Therefore the solution can be expressed in a mathematical form:

$$f(T', r) = \sum q(v)|p(r, v)| \\ = q_1 \times p_1 + q_2 \times p_2 + \cdots + q_{V-2} \times p_{V-2} + q_{parent} \times 1 \quad (3)$$

According to the problem formulation, the summation of $q_i \times p_j$ for T' rooted at r is the smallest of all possible $q_i \times p_j$ for graphs rooted at other vertices. This expressed in a formal form is:

$$f_r(T', r) = q_1 \times p_1 + q_2 \times p_2 + \cdots + q_{V-2} \times p_{V-2} + q_{parent} \times 1 \\ \leq f(T_{other}, r_{other}) = \sum q(v)|p(r_{other}, v)|, \text{for all possible } r_{other}.$$

At the same time we can calculate another set of numbers. If we choose the parent vertex of the leaf vertex to be the root instead of r, we have a summation of:

$$f_{parent}(T', parent) = \sum q(v)|p(r_{parent}, v)| \\ = q_1 \times (p_1 - 1) + q_2 \times (p_2 - 1) + \cdots + q_{V-2} \times (p_{V-2} - 1) + q_r \times 1 \quad (4)$$

Note that equation (3) and equation (4) are solutions for two different roots on the same sub-graph. The form of the summation also looks very similar. In order to get to the leaf root, all other vertices need to go through the parent vertex. If we choose the parent vertex to be the root, all other vertices take one less step to get to the root.

Here we observe a contradiction. By comparing equation (3) and equation (4), we can see that equation (4) always provides a smaller summation based on our assumptions in the first paragraph of this proof.

$$Equation\ (3) - Equation\ (4) = q_1 + q_2 + \cdots + q_{V-2} + q_{parent} - q_r$$

Since we supposed that q_r is not the greatest vertex weight of all vertices, at least one of q_i, where $i = 1, 2, \ldots, V-2$, is larger that q_r. Therefore $q_i - q_r$ is a positive value. A positive value plus a series of positive values gives a positive value. Therefore $Equation\ (3) - Equation\ (4)$ always gives a positive value. The summation of equation (4) always provides a smaller overall value.

Therefore, we have proved that if the leaf vertex does not have the greatest vertex weight, graphs rooted at its parent vertex always provide better solutions compared to graphs rooted at the leaf vertex. In other words, if the leaf vertex does not have the greatest vertex weight, it can never be chosen as the root. We do not have to examine this kind of leaf vertices if we can detect them.

Proof for Assumption 2

In order the find and mark off these kinds of leaf vertices, we need to do two things. First, we need to find the leaf vertices. Second, we need to determine if the leaf vertex has the greatest vertex weight in this graph. These two operations are fairly independent.

1. Looking for leaf vertices can be done by simply going through all vertices in the adjacency list and determine if a certain vertex is of more than one degree. This takes $O(V)$ time.
2. Determining if the leaf vertex has the greatest vertex weight can be done by simply going through all vertices and record the vertex with the greatest vertex weight. This also takes $O(V)$ time.

Therefore, in order to decide which vertices are the ones that can never be chosen as the root, we can simply go through all vertices and pick up the ones with a degree of one. Within the same loop, we can also record the vertex with the greatest vertex weight. After this loop, we can check to see if the vertex with the greatest weight is one of the leaf vertices. If it is not, then the set of leaf vertices is kept the same. If it is, then this vertex is removed from the set because, at this point, it is unknown whether or not this vertex should be chosen as the root. After these operations, we are left with two sets of vertices, and one of them contains vertices that can never be chosen as the root. We would, therefore, examine vertices in the other set.

So far, we have proved the second assumption that the operation to determine and mark off the vertices that can never be chosen as the root takes shorter time than $O(V \times E)$. It takes $O(V)$ time.

3.1.3 Summary of the algorithm optimization

Let us use a simple example to summarize the key ideas of the algorithm optimization. In the simple input graph below, we will use the vertex weights to name the vertices.

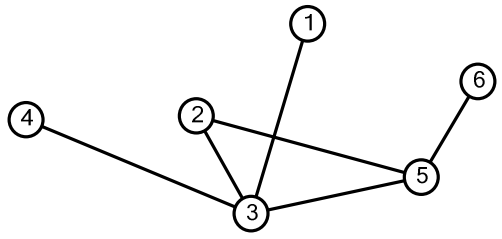

Figure 7. Simple input graph.

Vertex #	Degree	Weight
4	1	4
2	2	2
3	4	3
1	1	1
5	3	5
6	1	6

The algorithm proceeds as follows.

1. Examine the vertices and determine the ones that can never be chosen as the root.

   In this case, the candidate vertices that can never be chosen as the root (leaf vertex set) are as follows:

Vertex #	Degree	Weight
4	1	4
1	1	1
6	1	6

And the candidate vertices that require further examination are as follows:

Vertex #	Degree	Weight
2	2	2
3	4	3
5	3	5

2. Determine which vertex has the greatest weight. If it is in the leaf vertex set move it to the other set. If it is not in the leaf vertex set, then leave the two sets with no change.

   Vertices that can never be chosen as the root (leaf vertex set without the heaviest vertex) are as follows:

Vertex #	Degree	Weight
4	1	4
1	1	1

And vertices that need further examination are as follows:

Vertex #	Degree	Weight
2	2	2
3	4	3
5	3	5
6	1	6

We have only four vertices to check, and we know that the final solution will be a graph rooted at one of these four vertices.

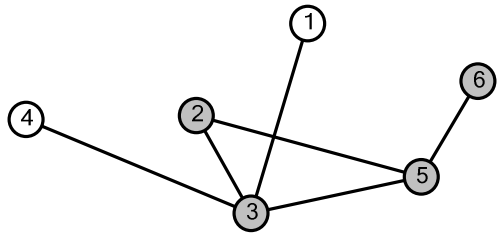

Figure 8. Vertices that need further examination.

Now, we loop through vertices, 2, 3, 5, and 6. For each of them, we use a BFS to find the single source shortest path tree rooted at the particular vertex and calculate the summation for equation (1).

The tree traversal algorithm proceeds as follows:

1. This is the best solution for graphs rooted at vertex 2. The solution is $f(T,r) = \sum q(v)|p(r,v)| = 4 \times 2 + 3 \times 1 + 1 \times 2 + 5 \times 1 + 6 \times 2 = 30$.

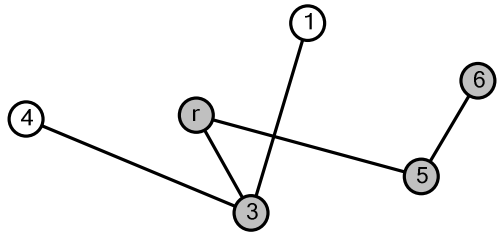

Figure 9. Solution for graphs rooted at vertex 2

2. This is the best solution for graphs rooted at vertex 3. The solution is $f(T,r) = \sum q(v)|p(r,v)| = 4 \times 1 + 2 \times 1 + 1 \times 1 + 5 \times 1 + 6 \times 2 = 24$.

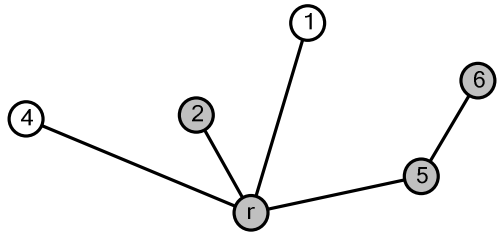

Figure 10. Solution for graphs rooted at vertex 3

3. This is the best solution for graphs rooted at vertex 5. The solution is $f(T,r) = \sum q(v)|p(r,v)| = 4 \times 2 + 2 \times 1 + 1 \times 2 + 3 \times 1 + 6 \times 1 = 21$.

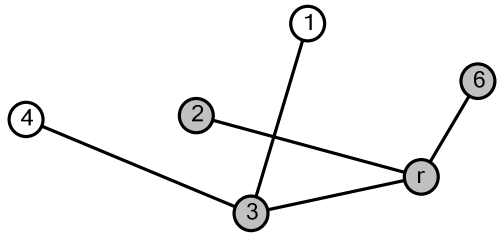

Figure 11. Solution for graphs rooted at vertex 5

4. This is the best solution for graphs rooted at vertex 6. The solution is $f(T,r) = \sum q(v)|p(r,v)| = 4 \times 3 + 2 \times 2 + 1 \times 3 + 3 \times 2 + 5 \times 1 = 32$.

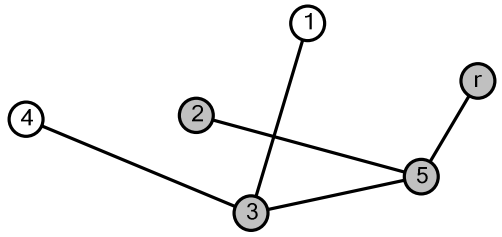

Figure 12. Solution for graphs rooted at vertex 6

After looping over all possible candidates that were selected at the beginning, we simply determine the best solution from this local solution, and that will be the best overall solution. For this example, the solution is a tree structure rooted at vertex 5, which is shown is Figure 9.

3.2 Pseudo Code for the algorithm

In this algorithm, we need to use several data structures for input, output, and temporary data storage. These data types include a first in first out queue, a fixed-size matrix (*Matrix*), and a *VertexNode* object, which stores the properties of each vertex.

The queue implemented by this application, *Queue*, needs to support the standard queue operations. In the pseudo code we call them *push* and *pop*, where push takes a parameter of a *VertexNode*.

The matrix object needs to support at least two operations. The method $matrix.isMarked(v_1, v_2)$ checks if the corresponding spot of these two input vertices is marked, and it returns a Boolean type. The method $matrix.mark(v_1, v_2)$ marks the corresponding spot in the matrix of the input vertices.

The *VertexNode* is a fixed-size object. It contains three fields, the weight of the vertex, the degree of the vertex in the input graph, and the hop of this vertex to a certain root vertex. The object needs to provide accessors to all three fields and one mutater, *setHops*, to update the hop property. We also have a fixed size array of *VertexNode*. V denotes the size of the array.

```
JADE-BFS-ALGORITHM(Matrix M)
 1    int maxWeight ← 0;
 2    int solutionSum ← 0;
 3    Matrix input ← the input adjacency M;
 4    Matrix solution ← populate with 0;
 5    VertexNode root ← null;
 6    Queue candidateSet ← Ø
 7    for every v ∈ V {
 8        if v.getWeight > maxWeight {
 9            maxWeight ← v.getWeight
10        }
11    }
12    for every v ∈ V {
```

```
13          if v.getDegree ≠ 1  or  v.getWeight = mWeight {
14                candidateSet.push(v);
15           }
16    }
17    while candiateSet ≠ Ø {
18          Matrix localSolution ← populate with 0;
19          int localSum ← 0;
20          Queue temp ← Ø;
21          VertexNode vroot ← candidateSet.pop;
22          temp.push(vroot);
23          vroot.setHops(0);
24          for every v ∈ V  and  v ≠ vroot {
25                v.setHops(∞)
26          }
27          while temp ≠ Ø {
28                VertexNode u ← temp.pop;
29                for every v ∈ V and v ≠ vroot {
30                      if input.adjacent(v, u) {
31                            if v.getHops = ∞ {
32                                  v.setHops(1 + u.getHops);
33                                  localSolution.mark(v, u);
34                                  temp.push(v);
35                            }
36                      }
37                }
38          }
39          for every v ∈ V {
40                localSum ← localSum + v.getWeight × v.getHops;
41          }
42          if localSum < solutionSum {
43                solutionSum ← localSum
44                solution ← localSolution
45                root ← vroot
46          }
47    }
```

3.3 Correctness Proof

The correctness of lines 1 to 16 is proved in the algorithm design section, Key Idea part 2. We proved the assumption that by eliminating the leaf vertices that do not have the greatest vertex weight improves the overall performance of the algorithm. After this optimization, the algorithm has a better average expectation run-time.

The correctness of using BFS single source shortest path tree as a solution of our problem is proved in the problem formulation section. We proved two assumptions. We proved that to optimize the overall summation of equation (1), we simply need to optimize its components. None

of the vertices need to sacrifice their shortest paths in order to generate the overall smallest summation of equation (1). We also proved that the output of components optimization for equation (1) is a connected graph. Therefore we proved that the problem can be converted into a BFS single source shortest path tree problem.

The correctness of the BFS search consists of two parts. We need to prove the output distance correctly exists. We also need to prove that the output path for each vertex is the shortest path possible. There are many literatures of the proof of BFS tree traversal. We provide a simple one here.

1. Prove that BFS algorithm outputs the correct path from the root to a certain vertex.

   This is an obvious observation since the child vertices are always derived from the parent vertices, and the parent vertices are always derived from the grandparents, which are one more step closer to the root vertex. If any of these links are broken, the loop at line 17 would terminate and return an unconnected graph. So, every established path should be correct.

2. Prove that BFS algorithm outputs the shortest path from the root to a certain vertex.

   Assume that we have an output tree graph rooted at a vertex r generated using the BFS algorithm. We also assume that the path from r to v going through u shown on the graph is not the shortest path. Instead there exists an alternative path that provides an even shorter path from r to v going through w. We can visualize the situation as the graph below:

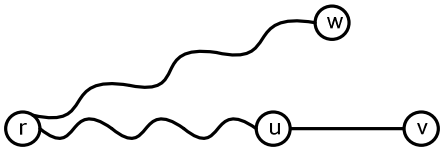

Figure 13. The output path of BFS algorithm: $p(r,u) + p(u,v)$.

An alternative path is shown below.

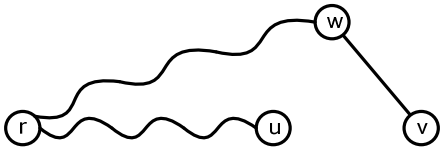

Figure 14. Assume an alternative path $p(r,w) + p(w,v) < p(r,u) + p(u,v)$.

Vertices u and w both can be the parent of vertex v. Since going through w provides a shorter path, and our input graph is a undirected graph, we can learn that $p(r,w) < p(r,u)$. This is to say that w has a shorter hop distance to the root r compare to u.

According to the algorithm, the order the vertices were pushed into the queue is based on the hop distance of the vertices. Vertices that have shorter hop distances need to be pushed in to the queue sooner. The feature of the queue data structure tells us that the elements that are pushed in sooner are also popped out sooner.

Here we observe a contradiction. Vertex w has a shorter hop distance than u. Vertex w is pushed into the queue sooner than u. Vertex w is also popped out of the queue sooner. Therefore the neighbors of vertex w are examined before the neighbors of vertex u. If vertex v is connected to both w and u, w should have marked v before u did. This is a contradiction of the assumption that Figure 11 is the output of the BFS algorithm while going through w provides a shorter hop distance for v from r.

Therefore, we proved that the BFS algorithm always gives the shortest path for a certain input graph rooted at a certain vertex.

4 Computational Complexity Analysis

4.1 Time Complexity

The time complexity of the algorithm in section 3.2 is analyzed as follows.

1. The loop from line 8 to line 11 simply takes $O(V)$ time. This loop determines the heaviest vertex.
2. The loop from line 12 to line 16 simply takes $O(V)$ time. This loop builds the candidate-vertex set for the examination of their single source shortest path trees. The leaf vertices that do not have the greatest vertex weight are eliminated.
3. In the worst case, the outer loop from line 17 to line 47 takes $O(V)$ time. The worst case happens when there are no leaf vertices or there is only one leaf vertex and it is the heaviest vertex at the same time. If this case occurs, we need to examine the single source shortest path tree for all of the vertices. Since we cannot eliminate the possibility of this situation, such as when the input is a complete graph, we cannot promise a faster worst-case run-time compare to the naïve algorithm.
4. The loop from line 24 to line 26 takes $O(V)$ time. The vertices that are not the current chosen root are assigned a hop value as infinite after this loop.
5. The loop from line 27 to line 38 takes $O(V)$ time. Every vertex that is not the root vertex is pushed into the temporary queue once.
6. The loop from line 29 to 37 takes $O(D)$ time, where D denotes the degree of a certain vertex. The vertices are poped off from the temporary queue one after the other. After popping out each one of the vertices, we mark its unmarked neighbors. The number of unmarked neighbors is at most the number of degree of this particular vertex.

7. Combining 5 and 6, the overall run-time from lines 5 to line 6 is $O(V) \times O(D)$, which equals $O(E)$.

8. The loop from line 39 to 41 takes $O(V)$ time. After we updated the hop value of all vertices after the BFS tree traversal, we loop over all vertices and sum up the value of their weights multiplied by their hop value. This is the local minimal solution for our problem. It is the local output of equation (1).

9. Combining 3, 4, 7, and 8, the overall worst-case run-time from line 17 to line 47 takes $O(V \times E)$. The reason of the conversion is based on the consideration that V is a lower term compare to E in a connect graph as we discussed earlier.

$$O(V) \times \big(O(E) + O(V)\big) = O\big(V \times (V + E)\big) = O(V \times E)$$

10. Combining 1, 2, and 10, the overall worst-case run-time of this algorithm is $O(V \times E)$. We omit the terms from 1 and 2 because $O(V) + O(V)$ compared to $O(V \times E)$ is a much lower term.

$$O(V) + O(V) + O(V \times E) = O(V \times E)$$

We can consider everything else takes constant time locally.

Although the worst-case run-time of this algorithm is the same as the naïve application of the BFS tree traversal, the average expectation is dramatically increased. We managed to use a fast operation to reduce the size of the problem at the beginning of the algorithm. In the example of the previous section, we examined only 4 out of 6 vertices by applying this algorithm.

On the other hand, we cannot eliminate the possibility of graphs with no leaf graphs. Our algorithm cannot promise a faster output for all kinds of input. Therefore, the worst case analysis is still $O(V \times E)$.

4.2 Space Complexity

As we discussed earlier, dynamic programming is used for the algorithm. The program maintains a certain amount of data while computing. This data is stored in abstract data types that do not impair the performance of the algorithm.

4.2.1 Input space requirement

For our algorithm, we need a source matrix to store the adjacency relationships between vertices of the input graph $G = (V, E)$. This matrix takes $V \times \frac{V}{2}$ space. It stores *TRUE* or *FALSE* depending on whether the two vertices are connected. Since the input graph is an undirected graph, the matrix is symmetric divided into two halves by the diagonal from top-left to the right-bottom. Therefore, we need only half of the full matrix to store this adjacency relationship information.

Then, we need an array type to store the properties of each of the vertices. The array has V elements. Each element stores a created abstracted data type *VertexNode*. Within this data type, we keep three fields, the weight of this vertex, the degree of edges for this vertex, and the hop distance from the vertex to a certain chosen root vertex. The first field is immutable. The second field, also

immutable, indicates the degree of edges in the input graph. The third field is a mutable field that keeps track of how many hops away a vertex is from a certain root at a certain time for this particular vertex. Therefore, this *VertexNode* type of array takes a space of $V \times 3$.

Therefore, the overall space requirement for the input of our problem is

$$O\left(V \times \frac{V}{2} + V \times 3\right)$$

4.2.2 Output space requirement

The output of this algorithm is an adjacent matrix with V rows and V columns. The matrix stores *TRUE* or *FALSE* depending on whether or not two vertices are connected. This matrix is a modified from the input matrix. The input matrix can be fairly full. For example, this can occur when it is a connected graph. There is only edge number upper bound, $V \times \frac{V-1}{2}$. This extreme case occurs when the input graph is a complete graph. The output graph demonstrates a tree structure, so the number of edges is one less than the number of vertices, $E = V - 1$.

There are other data types that are used for storing the relationships of a connected graph, but since the input graph is stored as a matrix, it is easy for the implementation to use this same data type. In addition, since we are focused on the runtime of the algorithm, we assume the memory performance of the machines is not a barrier. This is contrary to the problem we mentioned in the beginning of the previous section. The main barrier for BFS algorithm implementation on massive graphs in real life is memory performance. The matrix also takes a pace of $V \times \frac{V}{2}$.

The output also includes an object of type *VertexNode* that indicates what vertex was chosen as root that generated the overall minimal solution. It takes a space of ~ 1, which can be ignored. Therefore, the overall space requirement for the output of our problem is $V \times \frac{V}{2}$.

4.2.3 Memory allocation during the computation

As we discussed earlier, the linear tree traversal algorithm requires a first in first out queue data structure to store the *VertexNodes* during the computation. In the worst case, all of the vertices need to be examined. In this case, there are no leaf vertices or there is only one leaf vertex which, at the same time, has the greatest vertex value. If the worst case occurs, we need a queue of V elements to store the vertices. Therefore the space requirement during the computation is V.

4.2.4 Summary of Space Complexity

As discussed in the three sections above, our algorithm requires $V \times \frac{V}{2} + V \times 3$ for the input, $V \times \frac{V}{2}$ for the output, and V for the temporary space during computation. These three parts are fairly independent. The overall space requirement for our algorithm is:

$$O\left(V \times \frac{V}{2} + V \times 3\right) + O\left(V \times \frac{V}{2}\right) + O(V) = O(V \times V + V \times 3 + V) = O(V^2)$$

4.3 Comparison with Naïve Brute Force Algorithm

The brute force algorithm compares all possible combinations of a tree structure that can be formed from the input graph $G = (V, E)$. For each generated tree structure, the algorithm loops over all vertices and computes the summations for equation (1). Therefore, for each generated tree structure, the algorithm gets a set of possible solutions. Then, it compares these summations. Only the smallest one is kept as the solution for this particular tree structure. By repeating this process on every tree structure generated from the input graph, the algorithm gets a set of possible solutions for all sub-graph tree structures. Then, it compares these summations, which are the best solutions for the tree structures. The smallest summation at this step is the minimal summation for equation (1) for the problem. The corresponding tree structure and the root vertex constitute the solution for the problem.

4.3.1 Comparison of Space Complexity

The space requirement for brute rorce algorithm is slightly different from our algorithm. The input still requires a $V \times \frac{V}{2}$ adjacent matrix and a $V \times 3$ array of VertexNode. The output still requires a $V \times \frac{V}{2}$ adjacent matrix and one *VertexNode*, which can be ignored. The difference is that no temporary storage is needed while computing because brute force algorithm does not use dynamic programming. It repeats the duplicate computation if it is needed.

4.3.2 Comparison of Time Complexity

The worst case running time for brute force algorithm is clearly exponential. To determine the possible tree structures in a given input graph, the algorithm needs to examine a combination of C_E^{V-1} sub-graphs. Some of them form tree structures and some of them may form disconnected graphs. To examine if they are connected, we can use BFS, so this give a $C_E^{V-1} \times (V + E)$, which is in this case $C_E^{V-1} \times V$.

Then, the algorithm needs to go through all sub-graphs that form tree structures. Let us call this loop 1. It takes C_E^{V-1} time because there are this many tree graph in the worst case. Within each sub-graph the algorithm need to loop through all vertices assuming that every one of them can be chosen as the root. Let us call this loop 2. It takes V time because every vertex needs to be checked. For each of the chosen roots, the algorithm needs the loop through all other vertices to calculate the summation for equation (1). Let us call this loop 3. It takes $V - 1$ time because the root weight is not considered. In order to determine the hop distances for each of the vertices, the algorithm needs to loop from the child vertex to the root vertex and count the hops. This step has to be done repeatedly for every vertex because dynamic programming is not applied. Let us call this loop 4. In the worst case, this loop takes $V - 1$ time because the longest possible distance from the root to another vertex is the number of edges in the sub-graph. If we ignore the run-time for the comparison and swap, we just consider the four loops. Therefore, the overall run-time needs to be at least:

$$O[C_E^{V-1} \times V + C_E^{V-1} \times V \times (V - 1) \times (V - 1)]$$

As we discussed earlier, we cannot eliminate the possibility of graphs with zero leaf nodes, such as a complete graph. Therefore the upper bound of E is quadratic to V. This equation provides an analysis of an exponential run-time.

5 Conclusion

This paper proposed an algorithm with $O(V \times E)$ run-time to solve the problem of minimizing the traveling cost to a single source vertex through a vertex-weighted, undirected graph. The paper provided a comprehensive problem formulation, algorithm design, and complexity analysis for the problem. We proved the correctness of applying a BFS algorithm to solve the tree traversal portion of this problem. The algorithm was further optimized by eliminating the vertices that can never be chosen as the root. After optimization, the average expectation of the run-time is decreased. The worst-case $O(V \times E)$, however, still exists.