Converting a network into a small-world network: A fast algorithm for minimizing average path length through link addition

Max Ward[1,*] and Amitava Datta[1]

1 Computer Science and Software Engineering, University of Western Australia, Perth, WA, Australia

*** max.ward-graham@research.uwa.edu.au**

Abstract

The average path length in a network is an important parameter for measuring the end-to-end delay for message delivery. The delay between an arbitrary pair of nodes is smaller if the average path length is low. It is possible to reduce the average path length of a network by adding one or more additional links between pairs of nodes. However, a naïve algorithm is often very expensive for determining which additional link can reduce the average path length in a network the most. In this paper, we present an efficient algorithm to minimize the average network path length by link addition. Our algorithm can process significantly larger networks compared to the naïve algorithm. We present a simple implementation of our algorithm, as well as a performance study of it in this paper.

Introduction

Many real-world networks are modeled as a graph, $G = (V, E)$, where V is a set of vertices and E is a set of edges connecting some of the vertex pairs from the set V. Any message sent from a source to a destination propagates through intermediate vertices. A packet incurs *end-to-end delay* if it has to go through many intermediate hops. Since any pair of nodes can act as a source-destination pair, it is desirable that all paths in a network go through as few intermediate hops as possible. In other words, any traffic in the network will incur less average delay if the average path length in the network is low. Quite often the edges of these graphs are considered to be weighted, the weights may indicate the cost of establishing an edge, the latency or other factors depending on the context of the network.

Many networks have regular structures, meaning each node is connected to an equal number of nodes on an average. This makes the average path lengths in real-world networks quite long. In *small-world* networks the degree distribution of the nodes follows a power law and the average distance between any pair of nodes is usually small. These networks are of interest for quite sometime since Milgram's pioneering paper [1]. Watts and Strogatz [2] designed an approach that improves the clustering coefficient of a random network and converts a random network into a small-world network. Comellas and Sampels [3] replaced each node of a network by a mesh network, the number of nodes in the mesh is equal to the degree of the node that is replaced by the mesh. This converted an arbitrary network into a small-world network. Fall [4] showed

that the addition of a few random edges in a network can reduce the average path length of a network significantly. Lu *et al.* [5] showed that an arbitrary network can be made into a small-world network by introducing a scale-free distribution of nodes in a binary tree network.

The small-world nature of a network can be characterized in several different ways. However, our focus in this article is the average path length for all pairs of nodes. Usually a lower average path length reduces the overall communication cost in a network. Though there is strong evidence from previous work by Fall [4] that addition of extra edges can reduce the average path length in a network, the algorithmic nature of this problem has been studied only recently.

Meyerson and Tagiku [6] have shown that the general problem of adding k edges for minimizing the weighted average shortest path lengths between all pairs of vertices is NP-complete. They designed several approximation algorithms for this problem [6]. We are interested in this paper in designing an algorithm that adds a single edge to an existing network for minimizing the average shortest path lengths between every pair of vertices. Though it may seem quite a restricted problem, it has applications in many different areas. Chang *et al.* [7,8] have considered the problem of adding radio-frequency (RF) links in multi-core processor design. Their aim is to reduce the latency of communication in multi-core architectures, by adding extra radio frequency links on top of a regular interconnection scheme like a mesh network. However RF interconnects require much more area and cannot replace the traditional interconnects. Adding even a single RF interconnect can improve latency significantly [7,8]. The weight on the link is the area requirement for an RF interconnect in this case. Ogras and Marculescu [9] consider a long range link over a regular mesh network for designing efficient Network-on-Chip (NoC) in VLSI design. The length of the long link and the volume of traffic between the nodes are the weights on the links in this case. Pickavet and Demeester [10] have designed heuristic algorithms for link restoration in Synchronous Digital Hierarchy (SDH) networks. One of the key parts in their four-phase algorithm is local optimization, where they try to improve the topology in terms of the spare capacity, by adding an extra link. The weight on the link is the spare capacity in this case. Jin *et al.* [11] and Newman [12] have simulated stronger community structures in artificial social networks by link addition. It is possible to strengthen a community by adding an extra link between two nodes v_i and v_j such that they share a relatively large number of friends. The probability of a link addition is the weight on an edge in this case and the probability is determined by counting the number of common friends.

Recently Gaur *et al.* [13] have studied several deterministic link addition strategies for converting an arbitrary unweighted and undirected network into a small-world network. They found that the best strategy is to add an additional (long) link in an arbitrary network to minimize the average path length of the network. Given a graph $G = (V, E)$, Gaur *et al.* [13] considers an $O(V^2 \log V)$ algorithm for the all-pairs shortest path algorithm for unweighted and undirected graphs. This algorithm is based on Dijkstra's shortest path algorithm and not an original contribution by Gaur *et al.* as they have stated explicitly in their paper [13]. There is a possibility of introducing $O(V^2)$ new edges in a connected graph. The all-pairs shortest path algorithm is run after introducing each new edge and the average path length is computed. Hence the all-pairs shortest path algorithm needs to be run $O(V^2)$ time and the overall complexity of determining which link addition results in the minimum average path length is $O(V^2 \times V^2 \log V) = O(V^4 \log V)$.

In this paper we present a more efficient algorithm for the problem of introducing a new link for minimizing the average path length. Our algorithm, unlike that considered by Gaur *et al.* [13], works on weighted, directed graphs. Hence our algorithm is more general than the algorithm considered by Gaur *et al.* [13], and, as we will show, it is also

asymptotically faster in the worse-case. We present our algorithm, its time complexity and also experimental results in this paper. Our algorithm runs in $O(V^2E + V^3)$ time, which has a worst-case time complexity of $O(V^4)$ as opposed to the naïve algorithm that takes $O(V^5)$ time for weighted directed graphs.

Methods

An efficient algorithm for link addition that minimizes the average path length

In general, the shortest path between vertices v_i and v_j is the path that has the minimum sum weight. In the simplest case each edge has a unit weight, or is unweighted, and the shortest path is the minimum number of edges between v_i and v_j. The *single-source shortest path* (SSSP) algorithm finds the shortest paths between a given vertex v_i and all other vertices in the graph; and the *all-pairs shortest path* (APSP) algorithm finds the shortest paths between all pairs of vertices. We refer the reader to the book by Cormen *et al.* [14] for a review of shortest path algorithms. We will use the Floyd-Warshall algorithm as the basis of our new algorithm, as it is known to be one of the most efficient algorithms for this problem [14]. Asymptotically faster algorithms are known for sparse graphs, but the Floyd-Warshall algorithm is simple to implement and understand, and has very low constant factors in practice. It should be noted that Floyd-Warshall can be trivially replaced by a faster algorithm in our algorithm. We refer to the problem of *link addition that minimizes the average path length* as the *MinAPL* problem as in [13]. A naïve application of the Floyd-Warshall algorithm can solve this problem in the following way. We first compute the APSP of a graph G by executing the Floyd-Warshall algorithm in $O(V^3)$ time, and compute the average path length of G. Next, we introduce all possible $O(V^2)$ additional links and compute the average path length of the resulting graph by using the Floyd-Warshall algorithm and determine the link whose addition minimizes the average path length. This algorithm will take $O(V^2 \times V^3) = O(V^5)$ time. We show in this paper that it is possible to solve this problem in $O(V^4)$ time resulting in a much faster algorithm.

A fast algorithm for the MinAPL problem

We first explain a simple property for the addition of a link in a graph. This property will help us in designing the efficient algorithm for the MinAPL problem. The proof of the following lemma can be found in the book by Cormen *et al.* [14].

Lemma 1. *Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to R$, let $p = v_1, v_2, \ldots, v_k$ be a shortest path from vertex v_1 to v_k. For any i and j such that $1 \leq i \leq j \leq k$, let $p_{ij} = v_i, v_{i+1}, \ldots, v_j$ be the subpath of p from vertex v_i to vertex v_j. Then p_{ij} is a shortest path from v_i to v_j.*

We denote an edge between two vertices v_k and v_l by e_{kl} and the weight of e_{kl} as $w(e_{kl})$. We denote the average shortest path length of G as $APL(G)$. For a graph $G = (V, E)$, the main task in the MinAPL algorithm is to add an edge between a pair of vertices (v_k, v_l) such that $e_{kl} \notin E$ (we call such an edge an *augmenting edge*) and compute the average shortest path length, $APL(G')$ for the new graph $G' = (V, E \cup e_{kl})$. Then an edge $e_{pq} \notin E$ is the solution for the minAPL problem when,

$$APL(G') = \min_{\forall e_{kl} \notin E} APL(G = (V, E \cup e_{kl})) \text{ where } G' = (V, E \cup e_{pq}) \quad (1)$$

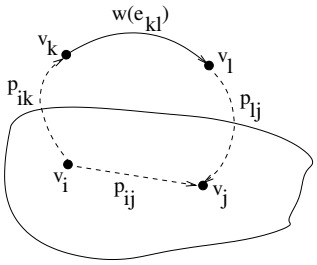

Fig. 1. Illustration for Lemma 2.

So the main task in our algorithm is to introduce all the augmenting edges e_{kl} one by one and compute the average path length of the resulting graph every time and choose the edge that gives the minimum average path length. The heart of our algorithm is the following lemma. We denote the shortest path from vertex v_i to v_j as p_{ij}, and its weight as $w(p_{ij})$.

Lemma 2. *Suppose $G = (V, E)$ is our original graph. Consider a directed edge $\overrightarrow{e_{kl}} \notin E$ and the resulting augmented graph $G' = (V, E \cup e_{kl})$ after the introduction of e_{kl}. Consider a pair of vertices v_i and v_j. The weight of the shortest path from v_i to v_j in G' is: $min\{w(p_{ik})) + w(e_{kl}) + w(p_{lj}), w(p_{ij})\}$.*

Proof. We refer to Fig. 1 for an illustration. Suppose the shortest path from v_i to v_j has changed in G', then this new shortest path must go through e_{kl}. Also, this new shortest path will consist of the shortest path from v_i to v_k, e_{kl} and the shortest path from v_l to v_j. Note that the difference between the graphs G and G' is only in the augmenting edge e_{kl}. From Lemma 1, a subpath of a shortest path is a shortest path. Hence p_{ik} and p_{lj} remain the shortest paths in G' between the vertex pairs (v_i, v_k) and (v_l, v_j) respectively. So the new shortest path must consist of the weights $w(p_{ik}) + w(e_{kl}) + w(p_{lj})$, if $w(p_{ij}) > w(p_{ik}) + w(e_{kl}) + w(p_{lj})$. □

We can take advantage of Lemma 2 in the following way. If we compute all-pairs shortest paths in the original graph G, we have the weights $w(p_{ik})$ and $w(p_{lj})$ that we can use for determining the new shortest paths in G' for all vertex pairs like (v_i, v_j). We first run the Floyd-Warshall algorithm on the input graph G and store the weights of the all pairs shortest paths in a matrix ASP. The entry $ASP[i, j], 1 \leq i, j \leq |V|$, stores the weight of the shortest path from v_i to v_j. Note that $ASP[i, j] \neq ASP[j, i]$ in general as G is a directed graph. Next we introduce the augmenting edges separately and iteratively and compute the new average path length for each augmented graph after the introduction of an augmenting edge. The algorithm outputs the augmenting edge that minimizes the average path length among all possible augmenting edges. The ASP matrix can be populated in $O(V^3)$ time using the Floyd-Warshall algorithm. The average path length of G can also be computed at the same time. Our algorithm is stated in Algorithm 1.

The correctness of the algorithm follows from Lemma 2. Since there are $O(V^2 - E)$ augmenting edges, the outer for loop executes $O(V^2 - E)$ times. The inner for loop examines every edge in G, hence it executes $O(E)$ times for every iteration of the outer loop, giving an overall complexity of $O(V^2E + V^3)$, which is $O(V^4)$ in the worst case. The second term in the complexity is due to the complexity of the Floyd-Warshall algorithm that needs to be run before our algorithm. The complexity of executing the Floyd-Warshall algorithm naïvely for each augmenting edge will be $O(V^5)$ as stated in the previous section.

Algorithm 1 Algorithm MinAPL

Input: A directed graph $G = (V, E)$ where w_{ij} is the weight of edge e_{ij} between vertices v_i and v_j. and $e_{ij} = \infty$ if there is no edge between v_i and v_j.

Output: An augmenting edge e_{kl} that minimizes the average shortest path length in G.

```
function MinAPL(G)
    ASP, minAPL ← Floyd-Warshall(G)
    edge ← ∅
    for all augmenting edges e_kl ∉ E do
        sum ← 0
        paths ← 0
        for all vertex pairs v_i, v_j in G do
            sp ← min(ASP[v_i, v_j],
                ASP[v_i, v_k] + w(e_kl) + ASP[v_l, v_j])
            if sp ≠ ∞ then
                sum ← sum + sp
                paths ← paths + 1
        APL ← sum/paths
        if APL < minAPL then
            minAPL ← APL
            edge ← e_kl
    return edge
```

Results

We have done extensive experiments for evaluating the performance of our algorithm. All experiments were run on a Virtualbox Ubuntu 14.04 on an i7 3630QM processor. The algorithm was implemented using C++ and the GNU C++ compiler. For each of the experiments we have ensured that the output of our algorithm matches the output from the naïve algorithm exactly. All our graphs were generated through random introduction of edges ensuring that the graph was connected. First we generated random graphs with 40 vertices with different edge densities as a fraction of the total possible edges of the complete graph. The set of augmenting edges was defined as the remaining edges not added to the graph. The performance comparison of the naïve algorithm with our algorithm is shown in Fig. 2. The weights on the edges are randomly generated with a value between 1 and 100. Our algorithm runs much faster for both sparse and dense graphs. If the total number of edges for the complete graph is represented as C, our algorithm runs in less than a milliseconds for $0.3C$ augmenting edges, while the naïve algorithm runs in about 38 milliseconds. Similarly, for $0.7C$ augmenting edges, our algorithm runs in 2.3 milliseconds, whereas the naïve algorithm takes over 100 milliseconds. Though we have run our algorithm several times for each graph, the running times were almost identical for each run and hence we have not shown any confidence intervals in the graphs.

Next, we evaluated the run times of the two algorithms for larger graphs. We fixed the number of augmenting edges to $0.4C$ while increasing the number of vertices. The results are shown for the naïve algorithm and our algorithm in Fig. 3. Our algorithm outperforms the naïve algorithm significantly in this case as well. For example, the runtime of the two algorithms are 46.54 and 4472.14 milliseconds respectively for a graph with 100 vertices. Similarly, for a graph with 400 vertices, the respective runtimes are 11873.8 and 4.18×10^6 milliseconds respectively.

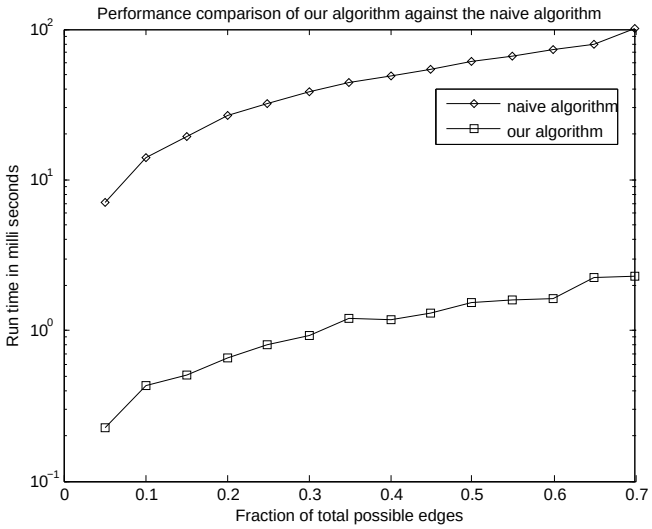

Fig. 2. A comparison between the naïve algorithm and our algorithm for graph with 40 vertices and different fraction of the total edges. The runtime is plotted in a logarithmic scale.

Discussion

We have presented a simple and efficient algorithm for adding an edge in a graph for minimizing the average shortest path length of the graph. The algorithm improves upon the naïve algorithm significantly and can be used for addition of edges in large graphs for minimizing the average path length. As an example, our algorithm finds an augmenting edge that minimizes the average path length of a graph with 1000 vertices in about 7 minutes.

Supporting Information

Source Code

The source code for our algorithm, and the testing framework for it, is available on GitHub. It can be found here `https://github.com/maxhwardg/best-link-addition`.

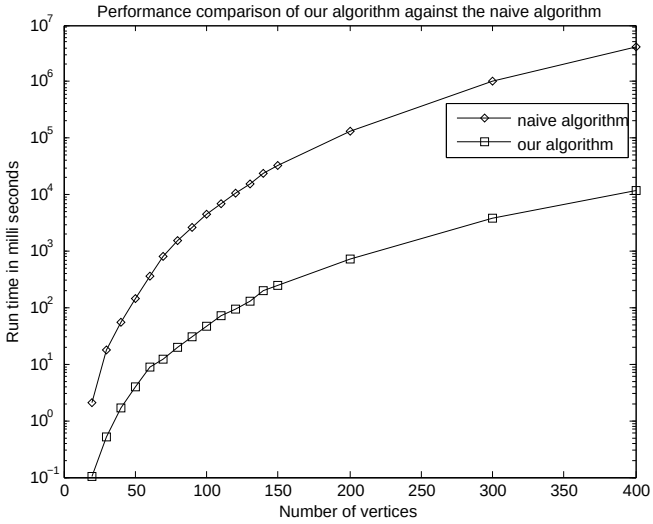

Fig. 3. The performance comparison between our algorithm and the naïve algorithm for different graph sizes. Each graph has a fraction of 0.6 of the total edges. The augmenting edge is chosen from the remaining fraction of 0.4 of the total edges. The runtime is plotted in a logarithmic scale.

Acknowledgments

Cras egestas velit mauris, eu mollis turpis pellentesque sit amet. Interdum et malesuada fames ac ante ipsum primis in faucibus. Nam id pretium nisi. Sed ac quam id nisi malesuada congue. Sed interdum aliquet augue, at pellentesque quam rhoncus vitae.

References

1. Milgram S. The small world problem. Psychology today. 1967;2(1):60–67.
2. Watts DJ, Strogatz SH. Collective dynamics of 'small-world'networks. nature. 1998;393(6684):440–442.
3. Comellas F, Sampels M. Deterministic small-world networks. Physica A: Statistical Mechanics and its Applications. 2002;309(1):231–235.

4. Fall K. A delay-tolerant network architecture for challenged internets. In: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications. ACM; 2003. p. 27–34.

5. Lu ZM, Su YX, Guo SZ. Deterministic scale-free small-world networks of arbitrary order. Physica A: Statistical Mechanics and its Applications. 2013;392(17):3555–3562.

6. Meyerson A, Tagiku B. Minimizing average shortest path distances via shortcut edge addition. In: Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques. Springer; 2009. p. 272–285.

7. Chang MF, Cong J, Kaplan A, Naik M, Reinman G, Socher E, et al. CMP network-on-chip overlaid with multi-band RF-interconnect. In: High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on. Ieee; 2008. p. 191–202.

8. Chang MCF, Socher E, Tam SW, Cong J, Reinman G. RF interconnects for communications on-chip. In: Proceedings of the 2008 international symposium on Physical design. ACM; 2008. p. 78–83.

9. Ogras UY, Marculescu R. " It's a small world after all": NoC performance optimization via long-range link insertion. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on. 2006;14(7):693–706.

10. Pickavet M, Demeester P. A zoom-in approach to design SDH mesh restorable networks. In: Heuristic Approaches for Telecommunications Network Management, Planning and Expansion. Springer; 2000. p. 107–130.

11. Jin EM, Girvan M, Newman ME. Structure of growing social networks. Physical review E. 2001;64(4):046132.

12. Newman ME. The structure and function of complex networks. SIAM review. 2003;45(2):167–256.

13. Gaur N, Chakraborty A, Manoj B. Delay Optimized Small-World Networks. Communications Letters, IEEE. 2014;18(11):1939–1942.

14. Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to algorithms. vol. 6. MIT press Cambridge; 2001.