8-1-2015

Constructing BFS Trees Using Tokens To Balance Speed and Network Traffic

Michael Spencer
University of Nevada, Las Vegas, michael.spencer.nv@gmail.com

CONSTRUCTING BFS TREES USING TOKENS TO BALANCE SPEED AND NETWORK TRAFFIC

By

Michael Spencer

Bachelor of Science in Computer Science
University of Nevada, Las Vegas
2004

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
May 2015

UNLV | GRADUATE COLLEGE

We recommend the thesis prepared under our supervision by

Michael Spencer

entitled

Constructing BFS Trees Using Tokens to Balance Speed and Network Traffic

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

Department of Computer Science

Ajoy Datta, Ph.D., Committee Chair

Lawrence Larmore, Ph.D., Committee Member

Yoohwan Kim, Ph.D., Committee Member

Emma Regentova , Ph.D., Graduate College Representative

Kathryn Hausbeck Korgan, Ph.D., Interim Dean of the Graduate College

May 2015

Abstract

Constructing BFS trees rooted at each node of a network helps solve many problems. Reliable communication to other nodes is easily managed and metrics such as the network diameter, shortest path between any two nodes, the center, the radius, and others can be easily computed. A traditional way to form a BFS tree from each node is for all nodes to construct their trees in parallel. While this is the fastest way to accomplish this task, it also requires a large amount of network traffic. In this thesis, we present a way to use a token passing algorithm to form a BFS tree from each node in the network within a desired network traffic limit. We will analyze how the algorithm works on several network topologies and determine the amount of tokens necessary to form BFS trees from each node as quickly as possible without stressing the network more than a desirable limit.

Table of Contents

List of Tables

List of Figures

Chapter 1 - Introduction

In graphs (a network of nodes and edges), there are several problems that can be solved by constructing a BFS tree from each node. Problems such as determining the diameter of the network, electing a leader, and determining the shortest path between all pairs of nodes (APSP) are much easier when these structures are present [1]. Unfortunately, when every node in a network begins constructing a BFS tree simultaneously, the communication between nodes can be very intense. Parallel execution is the fastest way to construct these BFS trees, it takes only O(diameter) rounds, but it uses O(n*m) messages (where n is the number of nodes and m is the number of edges). This is a very dense amount of messages to send in such a short time.

In many networks, having this level of sustained traffic is not desirable. So, it would be beneficial if there was a way to construct BFS trees from every node without using such a dense amount of traffic (i.e. stretch the message out over time to lessen the impact). Also consider that each network has unique requirements. It would be nice if a message rate limit could be identified, and have the BFS trees construct within this bound. In this thesis, I will present a way to accomplish that.

In the paper by Holzer and Wattenhofer [1], they present a new way to establish BFS trees in a network while spreading out the traffic over time. The basic idea of the solution is to pass a token around the network. When a node receives the token, it can begin constructing a BFS tree from itself. This solution works great at keeping the network traffic density low compared to parallel BFS formation. However, it takes a long time to complete the construction of the BFS trees (2*diameter + n rounds). In my thesis, I modify Dr. Holzer's algorithm to cover many different levels of network traffic. Obviously, there is a tradeoff between speed and message rate. For each increment of extra communication rate allowed, the BFS trees can usually be constructed faster.

In this thesis, I will go beyond theory to actually implement and test the original parallel BFS construction algorithm, as well as the multiple token algorithm presented in this thesis. I will show that depending on the topology and size of the network, you can determine how many tokens should be used to form BFS trees as fast as possible while keeping your traffic levels under a desired limit.

1.1 Problems Solved With Many BFS Trees

There are several types of problems that can be solved just by having a BFS tree constructed from each node. There are too many to cover in this thesis, but I will give an overview of some of the most popular problems.

1.2 Diameter

Calculating the diameter is efficient when each node has a BFS tree. Once the trees have formed, each node uses their tree to calculate the maximum distance between itself and any other node in its tree. Each node then uses its tree to send messages to determine the maximum of those distances. The resulting calculation is the diameter. The messages can actually be sent as the BFS trees are being constructed, so there isn't any additional time required. Running the algorithm in parallel, this process would run in O(diameter) time and use O(n * m) messages [2].

The multiple token algorithm in this thesis completes in diameter rounds after a token reaches the last node. Depending on how many tokens are used, this could take more than n rounds (for a single token) and almost as fast as diameter rounds if a lot of tokens are used. So, O(n + diameter) rounds is an accurate max representation and Ω(diameter) is the fastest it could finish. The amount of messages is the same, they are just spread out over a larger number of rounds if it takes longer, reducing the rate or density of communication.

1.3 Leader Election

Electing a leader in a network where the number of nodes and the diameter are not known is another problem that can be solved when every node has a BFS tree. First, each node constructs their BFS trees. During the convergecast, the leaves send a message to their parent with their ID. Each node passes along the maximum ID it sees. When the root of each tree receives the max ID in its tree, it compares it to its own ID. If its own ID is larger than the max it has received from its children, then it elects itself the leader, otherwise it broadcasts that it isn't the leader [2].

Since this algorithm uses the convergecast that is already part of constructing the BFS trees, the time and message complexity is the same as calculating the diameter.

1.4 All Pairs Shortest Path

All pairs shortest path (APSP) is a very popular problem to solve in distributed computing. It has several well-known solutions that are decades old such as the Floyd-Warshall algorithm [3, 4], the Bellman-Ford algorithm [5], and Dijkstra's algorithm [6]. There have also been many successful attempts to improve on these classic APSP algorithms. Such as scaling the Floyd-Warshall algorithm for parallelization [7, 10, 11], and improving the algorithm to use cache better [8, 9].

APSP is often a smaller problem that is used to help solve larger ones. For example, transitive closure, finding a regular expression denoting the regular language accepted by a finite automation, and inversion of real matrices and optimal routing [7]. The applications for APSP are very widespread. Such as geographical information systems, networking systems, robotics, intelligent transportation systems, and bioinformatics applications that can benefit from good solutions to the APSP problem [10, 11].

All of the above methods are useful in different ways. But, in a network where every node has a BFS tree constructed, the depth of the node in each BFS tree is known, and therefore all distances are known. Which means constructing a BFS tree from each node solves the All Pairs Shortest Path problem.

Chapter 2 - Token algorithm

2.1 Model

The network used in this thesis, is an undirected graph G = (V, E) with edges that have no weight. An edge represents a way to communicate directly between two nodes. For formulas in this thesis, the number of nodes are represented by **n**, and number of edges by **m**. It is assumed that each node has a unique identifier. At the beginning of an algorithm, the nodes have no knowledge of the network beyond their immediate neighbors. The algorithms in this thesis are designed for a synchronous network, but could easily be adapted for an unsynchronized network.

2.2 Breadth First Search Tree Algorithm

From Nancy Lynch's textbook [2], a directed spanning tree is defined as a directed spanning tree of a directed graph G = (V, E) to be a rooted tree that consists entirely of directed edges in E, all edges directed from parents to children in the tree, and that contains every vertex of G. A directed spanning tree of G with root node i is **breadth-first** provided that each node at distance d from i in G appears at depth d in the tree. Every strongly connected digraph has a breadth first directed spanning tree.

The algorithm to construct a BFS tree is as follows [2]:
At any point during execution, there is some set of processes that is "marked" initially just i_o. Process i_o sends out a search message at round 1, to all of its outgoing neighbors. At any round, if an unmarked process receives a search message, it marks itself and chooses one of the processes from which the search has arrived as its parent. At the first round after a process gets marked, it sends a search message to all of its outgoing neighbors.

This algorithm's time complexity is O(diameter) rounds and it generates O(m) messages.

2.3 Holzer/Wattenhofer Algorithm

This is the algorithm presented in the paper by Holzer and Wattenhofer [1]. First, a BFS tree is constructed from a root node. Once the initial BFS tree is completed, a token (referred to as a pebble in their paper) is passed to each node. When a node receives the token for the first time, it waits one round. The next round, the node begins forming a BFS tree and passes the token to a neighbor.

This algorithm's time complexity is O(n). The construction of all BFS trees stops at most diameter rounds after they were started. The runtime of the algorithm is determined by the time needed to build the initial BFS tree, O(diameter) plus the time needed by the last BFS tree that is initiated by the pebble O(n). Since diameter <= n, O(n) [1].

2.4 Single Token Algorithm

For the single token version of my algorithm, I modify the Holzer/Wattenhofer algorithm to decrease the amount of time it takes to complete. In the Holzer/Wattenhofer algorithm, the token doesn't start traversing the network until the root node's BFS tree has finished constructing. In round 1, the root node communicates to its neighbors when it requests to be their parent in its BFS tree. In my modified version of the algorithm, the token is passed to a neighbor in round 2, rather than waiting for the initial BFS tree to finish construction. Since the token starts one round after the BFS construction, the token will never have to wait for the BFS construction in order to find a neighbor because it's always one step behind. So, there is no reason to wait until BFS construction is complete to begin passing the token. As in the Holzer/Wattenhofer algorithm, when a node receives a token, it begins forming a BFS tree. The next round, it passes its token a neighbor from the main BFS tree.

The algorithm's time complexity is still O(n), although it is faster than the Holzer/Wattenhofer algorithm (by about diameter rounds). Since there is no wait time for the initial BFS tree construction, the total rounds is simply n + diameter rounds (the Holzer/Wattenhofer algorithm is diameter + n + diameter).

For every node in the network, there will be a different BFS tree formed. It's important to note that the token will only be passed to neighbors of the BFS tree of the node that initially received the token. For example, if a node has 3 network neighbors, but the BFS tree of the start node only has 2 neighbors for this node (presumably because that third node was chosen as a neighbor to a different node), then the third node will never receive a token from the current node (but it will receive a token from a different neighbor eventually). Using the start node's BFS neighbors instead of the network neighbors keeps the tokens from becoming trapped in a cycle.

In the diagram below, a yellow node is in the BFS tree of node 1, the orange hexagon is the token, and a green node is in the BFS tree of node 2 and node 1.

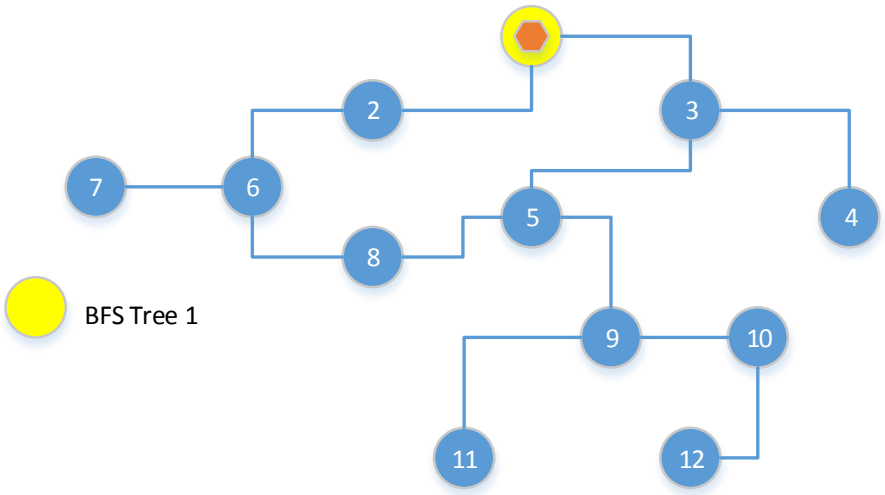

Figure 1: Single token round 1

In figure 1, the algorithm begins. Node 1 receives a single token and asks its neighbors if it can be their parent in its BFS tree (BFS tree 1).

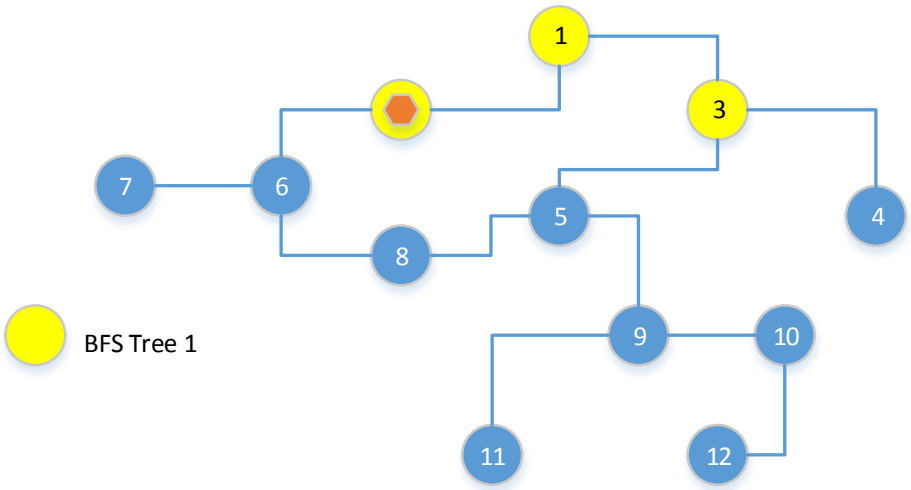

Figure 2: Single token round 2

In figure 2 the following things happen:

1. Nodes 2 and 3 agree to be node 1's children in BFS tree 1.
2. The token is passed to node 2.
3. Node 2 asks node 1 and node 6 if it can be their parent in BFS tree 2.
4. Node 2 also asks node 6 if it will be its child for BFS tree 1.
5. Node 3 asks node 4 and 5 if it will be its child for BFS tree 1.

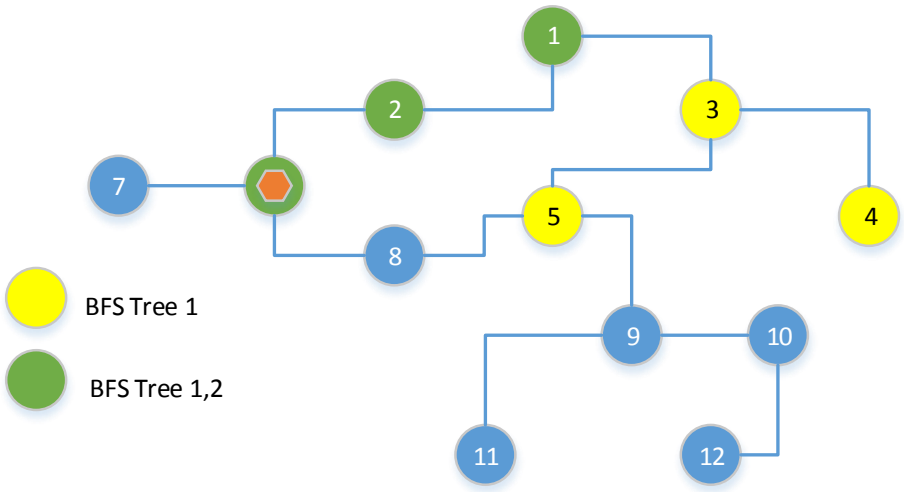

Figure 3: Single token round 3

In figure 3 the following things happen:

1. Node 6 has agreed to be node 2's child for both BFS tree 1 and 2.
2. Nodes 4 and 5 have agreed to be node 3's children for BFS tree 1.
3. The token has been passed to node 6.
4. Node 6 asks node 2, 7, and 8 if it will be its children for BFS tree 6.
5. Node 6 asks node 7 and 8 if it can be their parent for BFS trees 1 and 2.
6. Node 5 asks nodes 8 and 9 if they will be its children for BFS tree 1.
7. Node 1 asks node 3 if it will be its child for BFS tree 2.

The algorithm will continue like this until all BFS trees have been constructed.

2.4 Multiple Token Algorithm

This algorithm is similar to the single token algorithm, but it has modifications to handle multiple tokens. As in the single token version, a designated start node is chosen. This node receives all of the tokens before the algorithm begins. Once the algorithm starts, the initial node begins its BFS formation algorithm and then distributes its tokens to all of its BFS neighbors evenly. This means each node now needs to keep track of how many times it sends a token to a neighbor. When a node receives a token, it begins forming a BFS tree and then it also passes any tokens it has to each BFS neighbor evenly.

The algorithm's time complexity is still O(n), although it is faster than both the Holzer/Wattenhofer algorithm and the single token algorithm. Since there is no wait time for the initial BFS tree construction, the total rounds is simply n + diameter rounds (the Holzer/Wattenhofer algorithm is diameter + n + diameter). In the case of the multiple tokens, the n will usually be less than the single token n because multiple tokens are traveling for each round.

The first round is identical to the single token version.

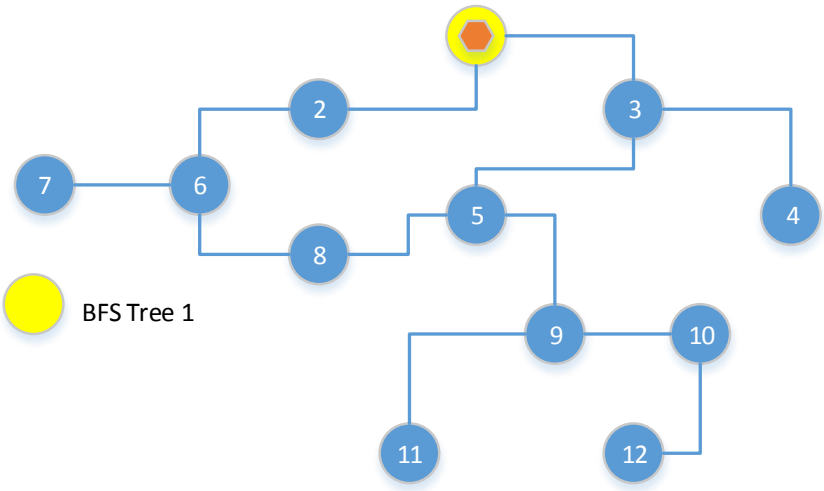

Figure 4: Multiple token round 1

In figure 4, the algorithm begins. Node 1 receives two tokens and asks its neighbors if it can be their parent in its BFS tree (BFS tree 1).

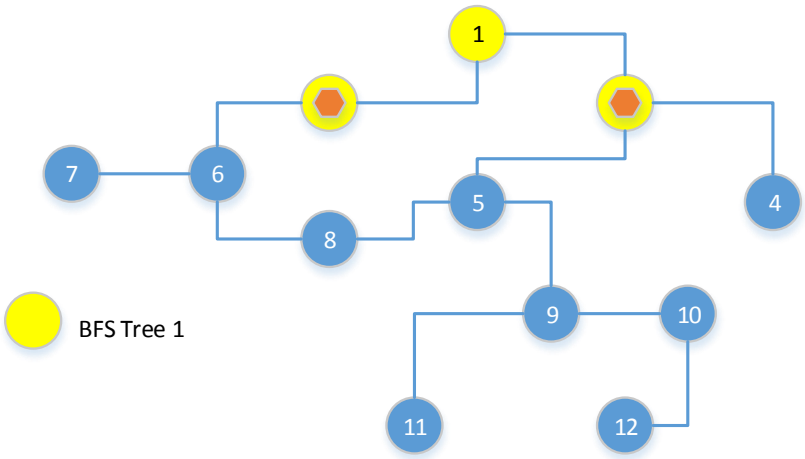

Figure 5: Multiple token round 2

In figure 5, the following things happen:

1. Nodes 2 and 3 agree to be node 1's parent in BFS tree 1.
2. Node 1 passes tokens to nodes 2 and 3.
3. Node 2 asks node 1 and node 6 if it can be their parent in BFS tree 2.
4. Node 2 also asks node 6 if it will be its child for BFS tree 1.
5. Node 3 asks nodes 1, 4, and 5 if they will be its children in BFS tree 3.
6. Node 3 also asks node 4 and 5 if it will be its child for BFS tree 1.

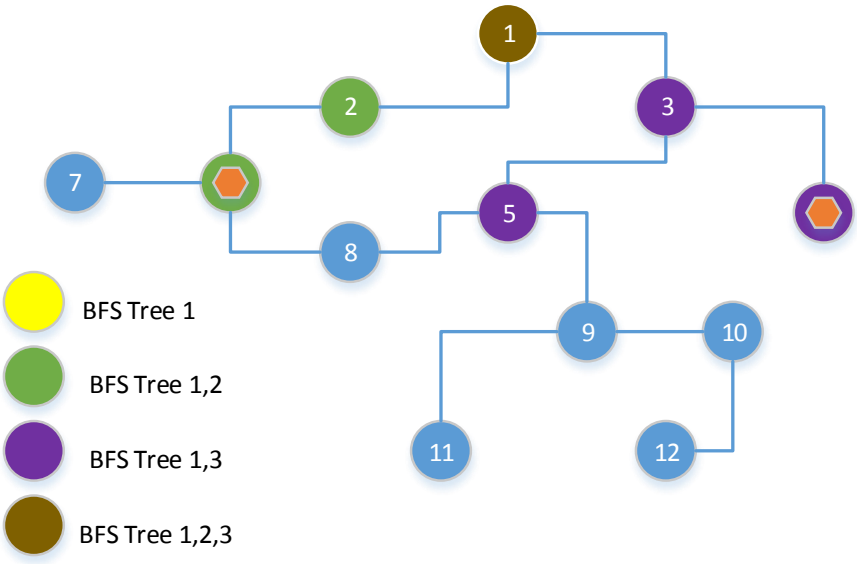

Figure 6: Multiple token round 3

In figure 6 the following things happen;

1. Node 6 has agreed to be node 2's child for both BFS tree 1 and 2.
2. Node 4 and 5 have agreed to be node 3's children for BFS tree 1.
3. The tokens have been passed to nodes 6 and 4.
4. Node 6 asks node 2, 7, and 8 if it will be its children for BFS tree 6.
5. It also asks node 7 and 8 if it can be their parent for BFS trees 1 and 2.
6. Node 5 asks nodes 8 and 9 if they will be its children for BFS tree 1.
7. Node 5 also asks nodes 8 and 9 if they will be its children for BFS tree 3.
8. Node 1 asks node 3 if it will be its child for BFS tree 2.
9. Node 1 also asks node 2 if it will be its child for BFS tree 3.
10. Node 4 asks node 3 if it will be its child for BFS tree 4.

In order to achieve an even distribution of tokens, each node counts how many times it sends a token to each neighbor. When a node has a token to pass, it chooses the neighbor with the fewest receptions. Also, when a node receives a token, it adds one to its counter for the neighbor it received the token from. This keeps the token from going back to a node until all nodes on that BFS branch have received a token.

It goes without saying that if a node has only one neighbor, when it has a token to pass, it just passes it back to its neighbor the next round. These leaf nodes do not need to track how many tokens are sent to each neighbor.

Adding just a second token makes a noticeable difference in the amount of things happening in each round. For example, in round 3 above, the single token algorithm generated 7 items, while two tokens generated 10 items. Each round continues to grow in communication until the BFS trees start completing. In a later chapter, we will see that this is represented in the form of network traffic (communication) increases for each token added.

2.5 Algorithm Pseudo code

Once per round, the "PassTokensToNeighbors" function would be called for each node. "ProcessEndOfRound" is called at the end of the round for each node.

Algorithm 1 Pass Tokens To Neighbors

```
function PassAllTokensToNeighbors(Node theNode, int round)
  while (theNode.TokenCount > 0) do
    if (theNode.neighborCount == 1) then
      theNode.BFSNeighbors[0].ReceiveToken()
    else
      Node neighborWithMinTokens <- GetMinReceptionNeighbor(theNode)
      neighborWithMinTokens.TokenSentCount++
      neighborWithMinTokens.ReceiveToken(round)
    end if
  end while
end function
```

Algorithm 2 Find neighbor that has received the fewest tokens

```
function GetMinReceptionNeighbor(Node theNode)
  Node minNeighbor = theNode.BFSNeighbors[0]
  For each (BFSNeighbor aNeighbor in theNode.BFSNeighbors) do
    if (aNeighbor.TokenSentCount < minNeighbor.TokenSentCount) then
      minNeighbor = aNeighbor
    end if
  Next
  return minNeighbor
end function
```

Algorithm 3 A node receives a token from a neighbor

```
function ReceiveToken(int round)
  dockedTokenCount++
  if (haventReceivedTokenYet == true) do
    haventReceivedTokenYet = false
    BeginBFSConstruction()
  end if
end function
```

Algorithm 4 Executed by each node at end of round. Moves docked tokens into "ready to send" state for next round.

```
function ProcessEndOfRound()
  TokenCount += dockedTokenCount
  dockedTokencount = 0
end function
```

Chapter 3 - Implementation

To test the algorithms and theories in this thesis, I created a synchronized network simulator in .net. I added a BFS construction algorithm as well as the token passing algorithm. I also designed various network topologies and programmed in the code for creating them. I tested and verified the accuracy of the BFS formation algorithms using various examples from text books and online references. After each run, I recorded the results which are reviewed in the next chapter.

I ran the algorithm over multiple levels of binary trees and ternary trees, several sizes of star networks, and four different mesh networks (combinations of large and small, sparse and dense). For each network, I first ran the parallel version, then I started with a single token and incrementally added a token for each run. I continued to increase the token count until the rounds could not be reduced anymore.

3.1 Rounds

During each run, I capture the number of rounds it takes to fully construct a BFS tree from every node. Before the rounds begin, a node is selected and is given all of the tokens. Then the simulation begins giving each node the time it needs to complete its processing for the round. When all nodes have completed their processing, the round ends. If there are some nodes who have not finished constructing their BFS tree, then the round counter is incremented, and a new round begins. This continues until all nodes report their BFS trees are constructed. This final round count is the value that is reported in the charts.

3.2 Network Traffic

This is an attempt to determine the message density or rate as the algorithm runs. It is calculated by counting the number of BFS requests each node receives from a neighbor. Each time a neighbor requests a node to be their child, the internal counter is incremented for that node (the one that receives the request). When all nodes report that their BFS tree is constructed, the request counters for all nodes are added together,

then divided by the number of nodes and divided again by the total rounds (requests per round per node). This final result is stored in the chart.

Traffic = Total Requests / Total Rounds / Total Nodes

3.3 Technical Details

The application is written in .net (specifically vb.net). I may convert it to C# at a later time. A conversion to C# would not impact or change the data results in this thesis. It was written using Visual Studio 2013. It should be accessible by Visual Studio 2010 or Visual Studio 2013.

To test the network simulator and the BFS construction code, I created the code necessary to generate random networks within specifications. For example, you can create a network with at least 5 nodes, but not more than 30 nodes, with at least 2 neighbors for each node, but not more than 5 neighbors for each node. This random network generation code is not used for this thesis, since I am using specific networks to collect results. The code is still accessible in the project.

The network simulator stores each node in a List (of nodes). I chose this data structure because it is easy to work with. Unfortunately, when the network gets large (more than 1000 nodes), the performance begins to slow down. If more performance is desired, converting the List data structure to an array should suffice. While more difficult to maintain and debug, the performance benefits would make this conversion beneficial for large network tests.

In any given round, a node can receive a BFS request for any BFS tree. This problem makes the code complex and difficult to follow at times. Each BFS request includes the BFS tree ID which is the node ID of the root node for that BFS tree. So, each node must maintain a list of BFS neighbors for each node ID in the network. This means each node requires enough space to store n * m (m is the number of network neighbors).

The user interface allows quick selection of the options for the simulation. The user selects the type of network to generate, along with how big to make it (if the network is

programmed to be sizeable). The user also chooses how many tokens to use or if it should be fully parallel.

When the simulation completes, on the left panel it shows the original network by listing each node and its neighbors. It also shows the BFS trees that are generated for each node in the same manner (listing each node and its BFS neighbors for that tree). The right panel shows the number of rounds it required to complete the run and the traffic generated during the run.

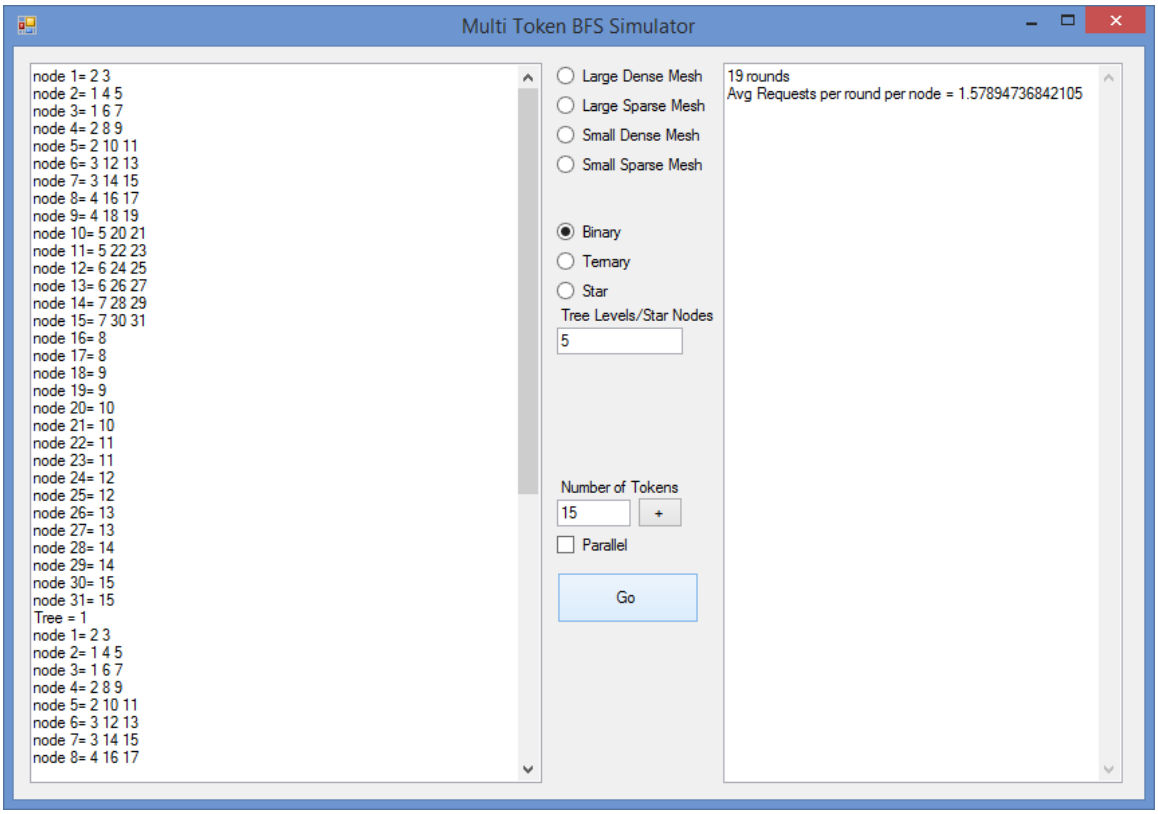

Figure 7: Visual result of an example run

3.4 Code Availability

The code is publicly available on GitHub. I hope to update the code after the thesis to make it easier for others to read and make it run faster. It is available at the following address:

https://github.com/MichaelSpencerNV/MultiTokenBFSConstruction

Chapter 4 – Results

4.1 Binary Tree

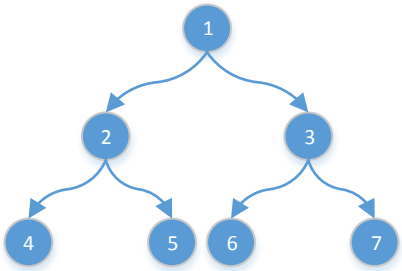

Figure 8: Binary tree example

For the binary tree, I ran the algorithm over trees of depth 4 (15 nodes) through 8 (255 nodes). The results show that the difference in speed between the fastest multiple token method and the fully parallel formation is roughly equal to the depth of the tree. For example, the parallel formation for a level 4 tree completes in 10 rounds. No matter how many tokens are added to the multi-token algorithm, it cannot complete the algorithm any faster than 13 rounds. This is because a token can only travel once per round. So even if all the tokens travelled in a straight line to the bottom of the tree, it would take at least as many rounds as the depth of the tree to reach the bottom.

In the case of the binary tree, a single token is very slow. Adding a second token cuts the time down roughly in half, while incurring a network traffic impact of a little less than double. Adding a third token has only a marginal positive impact on the time for small trees, but as the network gets larger, the improvement is greater as well.

When the tree is of a sufficient size (level 6 and up), the number of tokens required to change either the rounds or the traffic level is usually a power of 2. This is likely because each node has exactly two children (in the ternary tree, multiples of 3 occur). It is also because as the network gets larger, the sub trees become larger. The tokens have to travel through an entire sub tree and back before they can go to the other child's sub tree for a given node.

For display purposes (to save space), in the table, I skip the token amounts when there is no change between rows.

Using this table, if you had a level 6 binary tree network and wanted to form BFS trees from every node without experiencing more than 2 requests per node per round on average, you would use 6 tokens. The network would have BFS trees constructed from every node in 39 rounds and each node would only receive 1.59 BFS requests per round on average. If you increased the token amount to 8, the algorithm would complete 10 rounds faster, but generate 2.14 messages per node per round.

Binary Tree										
Level	4		5		6		7		8	
Tokens	Rnds	Trffc	Rnds	Trffc	Rnds	Trffc	Rnds	Trffc	Rnds	Trffc
1	35	0.40	69	0.43	135	0.46	265	0.48	523	0.49
2	21	0.67	39	0.77	73	0.85	139	0.91	269	0.94
3	21	0.67	35	0.86	61	1.02	111	1.14	209	1.22
4	15	0.93	25	1.20	43	1.44	77	1.64	143	1.78
6	15	0.93	25	1.20	39	1.59	65	1.94	115	2.21
8	13	1.08	19	1.58	29	2.14	47	2.68	81	3.14
12	13	1.08	19	1.58	29	2.14	43	2.93	69	3.68
16	13	1.08	17	1.76	23	2.70	33	3.82	51	4.98
24	13	1.08	17	1.76	23	2.70	33	3.82	47	5.40
32	13	1.08	17	1.76	21	2.95	27	4.67	37	6.86
64	13	1.08	17	1.76	21	2.95	25	5.04	31	8.19
128	13	1.08	17	1.76	21	2.95	25	5.04	29	8.76
Parallel	10	1.40	13	2.31	16	3.88	19	6.63	22	11.55

Table 1: Binary tree results

4.2 Ternary Tree

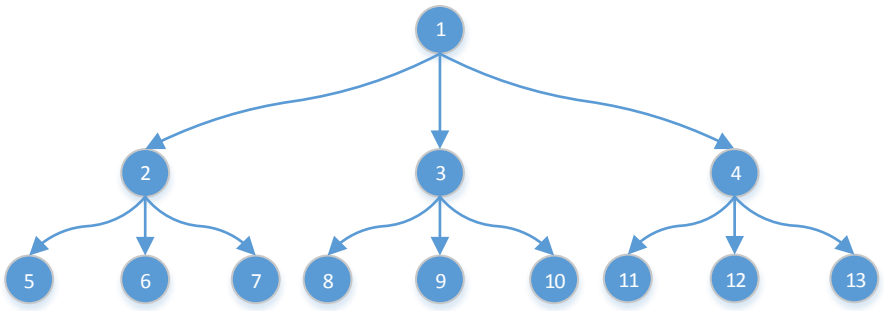

Figure 9: Ternary tree example

For the ternary tree network, I ran the algorithm on levels 2 (13 nodes) through 5 (364 nodes). Beginning at 6 tokens, all increments that generate new data is always a multiple of three (in some cases a power of 3). This is likely because the network is essentially divided into three sections, so the root can pass an equal number of tokens into each sub tree. On the next level of the tree, when each of those nodes can pass an even number of tokens to their children, we see another bump in new data.

In the ternary tree, adding a second token makes a large difference (about 30% faster), but not quite as much as the binary tree. The traffic also increases, but much less compared to the increase with the binary tree's second token. In the binary tree, adding a third token didn't make a big difference but in the ternary tree, we see another big jump (again about 30% faster). This makes sense now that each node has three children (vs two) to spread out the tokens to.

Ternary Tree								
Level	2		3		4		5	
Tokens	Rounds	Traffic	Rounds	Traffic	Rounds	Traffic	Rounds	Traffic
1	29	0.41	85	0.46	249	0.48	737	0.49
2	19	0.63	49	0.80	133	0.90	379	0.96
3	13	0.92	33	1.18	89	1.35	253	1.43
4	13	0.92	33	1.18	85	1.41	229	1.59
5	13	0.92	31	1.26	73	1.64	193	1.88
6	11	1.09	23	1.70	53	2.26	137	2.65
9	9	1.33	17	2.29	37	3.24	93	3.90
12	9	1.33	17	2.29	37	3.24	89	4.08
15	9	1.33	17	2.29	35	3.43	77	4.71
18	9	1.33	15	2.60	27	4.44	57	6.37
27	9	1.33	13	3.00	21	5.71	41	8.85
45	9	1.33	13	3.00	21	5.71	39	9.31
54	9	1.33	13	3.00	19	6.32	31	11.71
81	9	1.33	13	3.00	17	7.06	25	14.52
162	9	1.33	13	3.00	17	7.06	23	15.78
243	9	1.33	13	3.00	17	7.06	21	17.29
Parallel	7	1.71	10	3.90	13	9.23	16	22.69

Table 2: Ternary tree results

4.3 Star Network

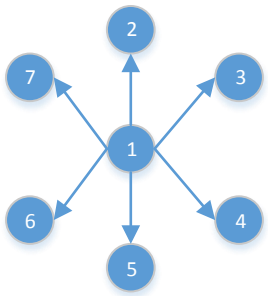

Figure 10: Star network example

The star network was interesting because no matter the size of the network, the parallel run always finishes in 4 rounds. This is because every node is connected to the root node directly. Unlike other networks, given enough tokens, the multiple token algorithm can almost reach the same speed as the full parallel algorithm. This is because the diameter of the star network is always 2.

When the network is large enough, each token makes a slight difference in speed and traffic. So, with this network topology, you can actually get a large number of choices for speed and network traffic. The star network turns out to be the most friendly network topology to the multiple token algorithm in this thesis because of how predictable it is.

Star Network						
Nodes	15		40		127	
Tokens	Rounds	Traffic	Rounds	Traffic	Rounds	Traffic
1	31	0.45	81	0.48	255	0.49
2	17	0.82	43	0.91	129	0.98
3	13	1.08	29	1.34	87	1.45
4	11	1.27	23	1.70	67	1.88
5	9	1.56	19	2.05	55	2.29
6	9	1.56	17	2.29	45	2.80
7	7	2.00	15	2.60	39	3.23
8	7	2.00	13	3.00	35	3.60
9	7	2.00	13	3.00	31	4.06
10	7	2.00	11	3.54	29	4.34
11	7	2.00	11	3.54	27	4.67
12	7	2.00	11	3.54	25	5.04
13	7	2.00	9	4.33	23	5.48
14	5	2.80	9	4.33	21	6.00
16	5	2.80	9	4.33	19	6.63
18	5	2.80	9	4.33	17	7.41
20	5	2.80	7	5.57	17	7.41
21	5	2.80	7	5.57	15	8.40
26	5	2.80	7	5.57	13	9.69
32	5	2.80	7	5.57	11	11.45
39	5	2.80	5	7.80	11	11.45
42	5	2.80	5	7.80	9	14.00
63	5	2.80	5	7.80	7	18.00
126	5	2.80	5	7.80	5	25.20
Parallel	4	3.50	4	9.75	4	31.50

Table 3: Star network results

4.4 Small Mesh Networks

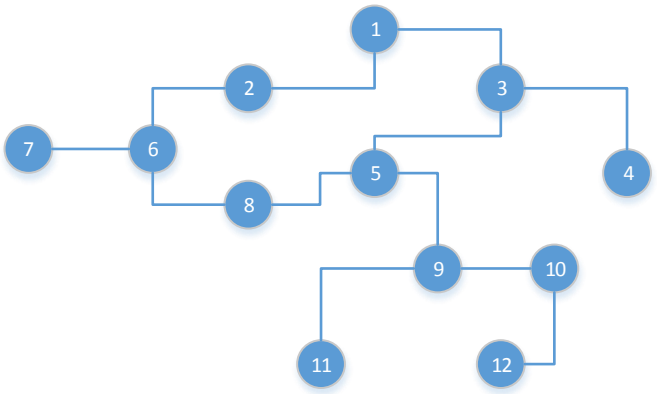

Figure 11: Small, sparse mesh network

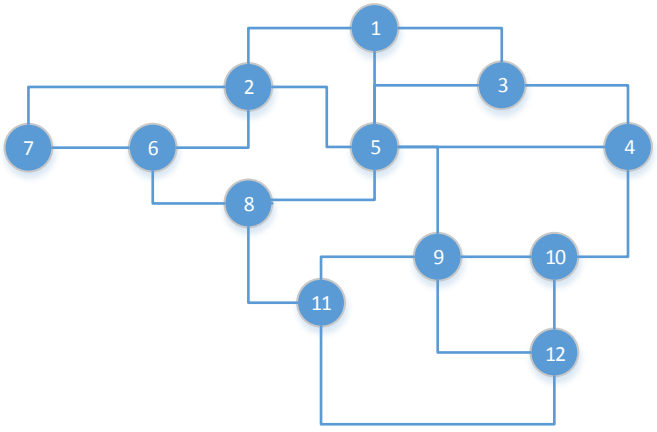

Figure 12: Small, dense mesh network

For the mesh networks, I created four examples. The first example is represented above in figure 10. It's a small, sparse mesh network with only 12 edges and 12 nodes. The second example is a small dense network with 20 edges and 12 nodes.

The increase in edges between the sparse and dense network results in both a faster completion of the algorithm and a higher traffic impact for the same amount of tokens when comparing the sparse and dense results in table 4. Also, it doesn't take very many tokens to close the gap between the multiple token and fully parallel algorithms. 6

tokens for the dense network and 8 for the sparse network finish within two rounds of the parallel speed.

As in the binary tree, the second token makes a large impact (almost as much as in the binary tree), but subsequent token amounts have only a marginal increase in speed and traffic.

Small Mesh Network				
Nodes	Sparse		Dense	
Tokens	Rounds	Traffic	Rounds	Traffic
1	30	0.43	25	1.16
2	18	0.72	15	1.93
3	18	0.72	13	2.23
4	16	0.81	13	2.23
6	16	0.81	10	2.90
8	14	0.93	10	2.90
Parallel	12	1.08	8	3.63

Table 4: Small mesh network results

4.5 Large Mesh Networks

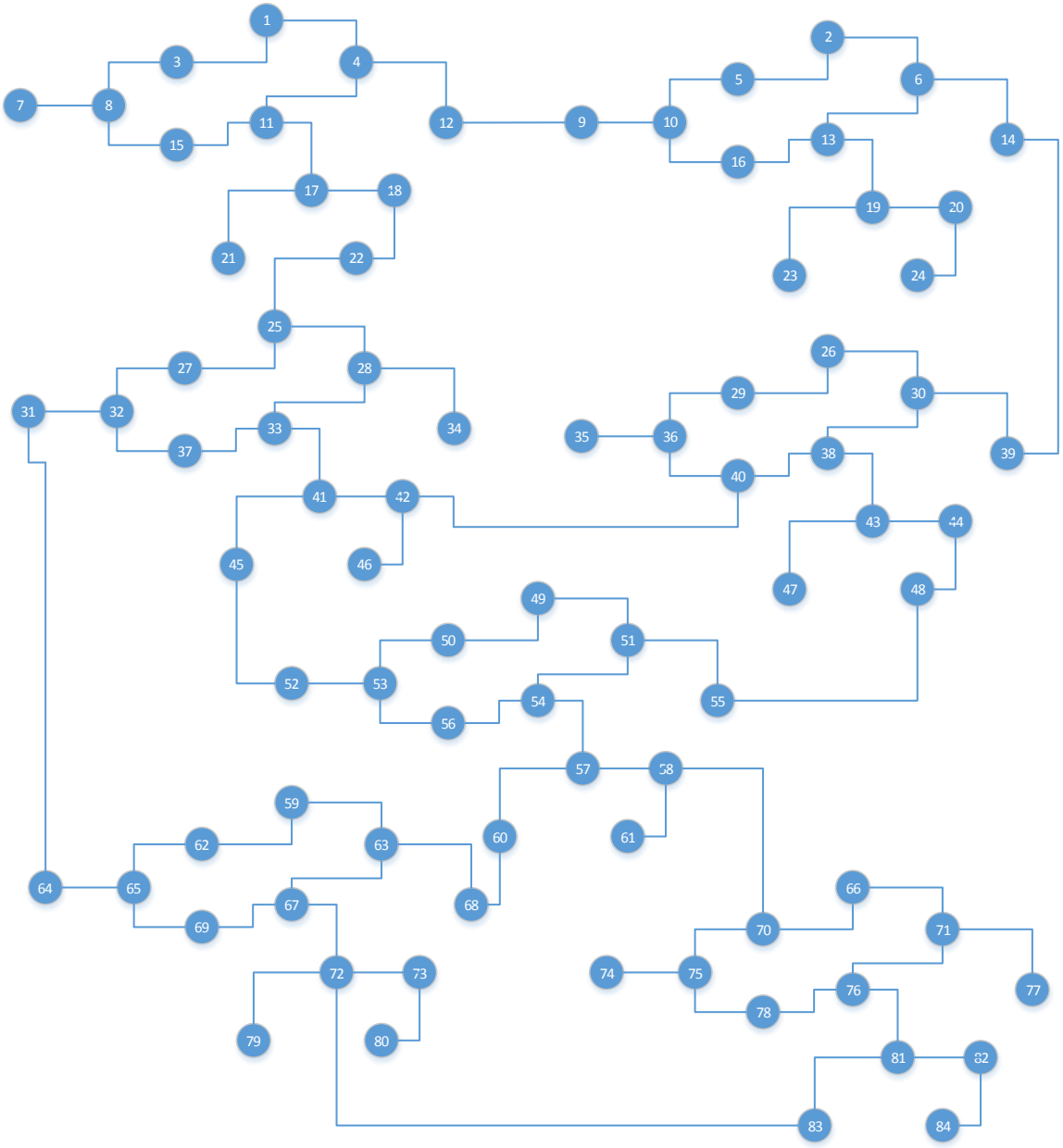

Figure 13: Large, sparse mesh network

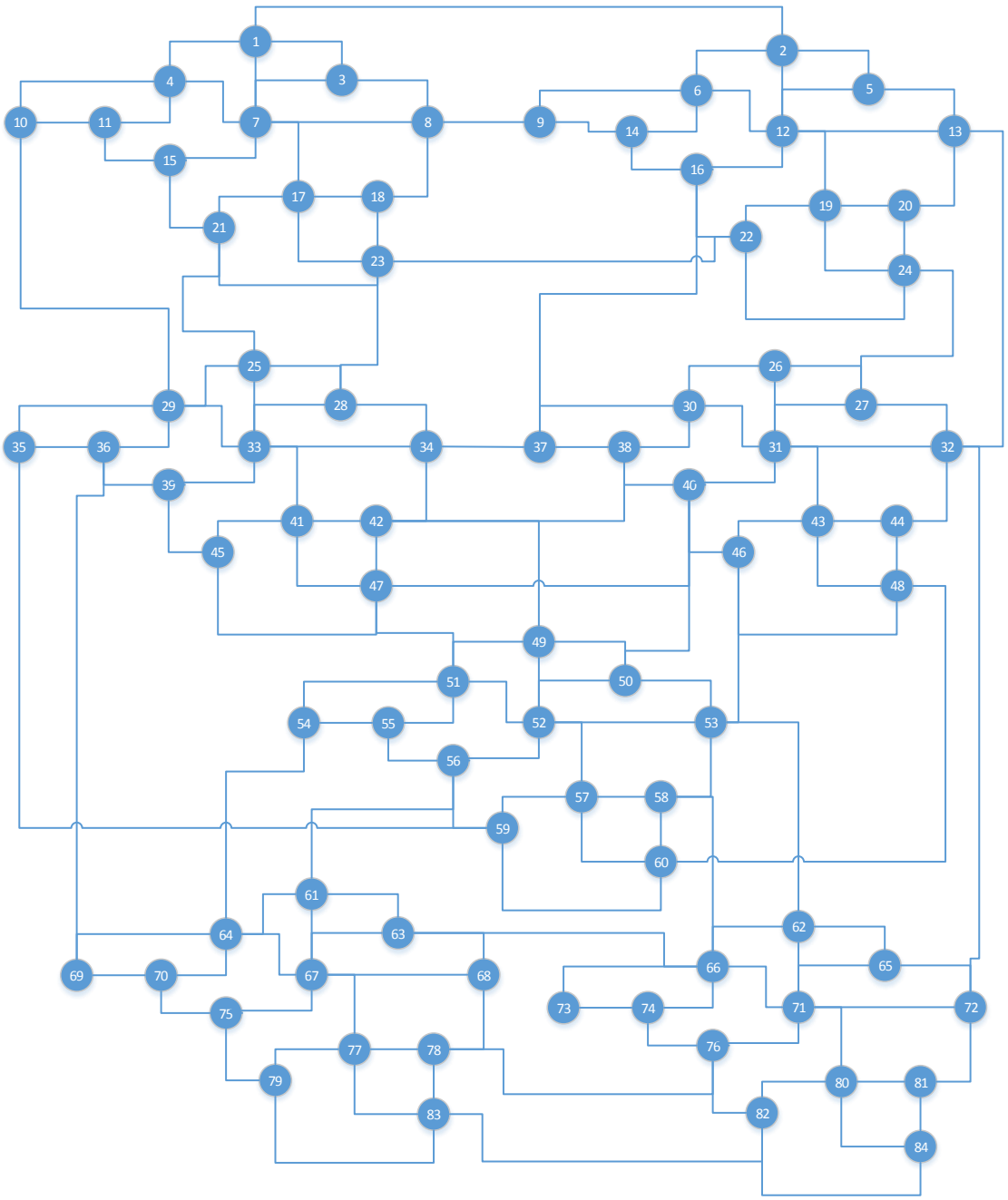

Figure 14: Large, dense mesh network

For the large mesh networks, I again created a sparse and a dense version. Both versions have 84 nodes. The sparse version has 94 edges, while the dense version has 167 edges.

As with previous networks, adding a second token has a large impact. In this case, subsequent tokens continue to make a difference in both networks. The dense network

has a many more increments to choose from, but also has a much larger traffic impact compared to the sparse network at any given token amount.

The fastest sparse network construction with 104 tokens, falls 11 rounds short of the speed of the parallel algorithm. This is due to the size of the network and the many bottlenecks that exist. It takes a while for the tokens, no matter how many, to reach the nodes on the other side of the network. In contrast, the dense network misses the parallel network by only 4 rounds of the parallel network, but it takes 428 tokens to reach the speed.

Large Mesh Network				
Nodes	Sparse		Dense	
Tokens	Rounds	Traffic	Rounds	Traffic
1	194	0.54	176	1.43
2	106	0.99	92	2.73
3	102	1.03	78	3.22
4	82	1.28	62	4.05
5	82	1.28	60	4.18
6	82	1.28	54	4.65
7	80	1.21	48	5.23
8	64	1.64	48	5.23
9	64	1.64	42	5.98
10	62	1.69	42	5.98
12	60	1.75	40	6.28
15	60	1.75	38	6.61
16	58	1.81	38	6.61
20	58	1.81	36	6.97
24	57	1.84	36	6.97
28	57	1.84	34	7.38
32	55	1.91	33	7.61
42	55	1.91	32	7.84
44	53	1.98	32	7.84
56	53	1.98	28	8.96
104	51	2.06	28	8.96
116	51	2.06	26	9.65
212	51	2.06	24	10.46
428	51	2.06	23	10.91
Parallel	40	2.63	19	13.21

Table 5: Large mesh network results

4.6 Rounds vs Tokens

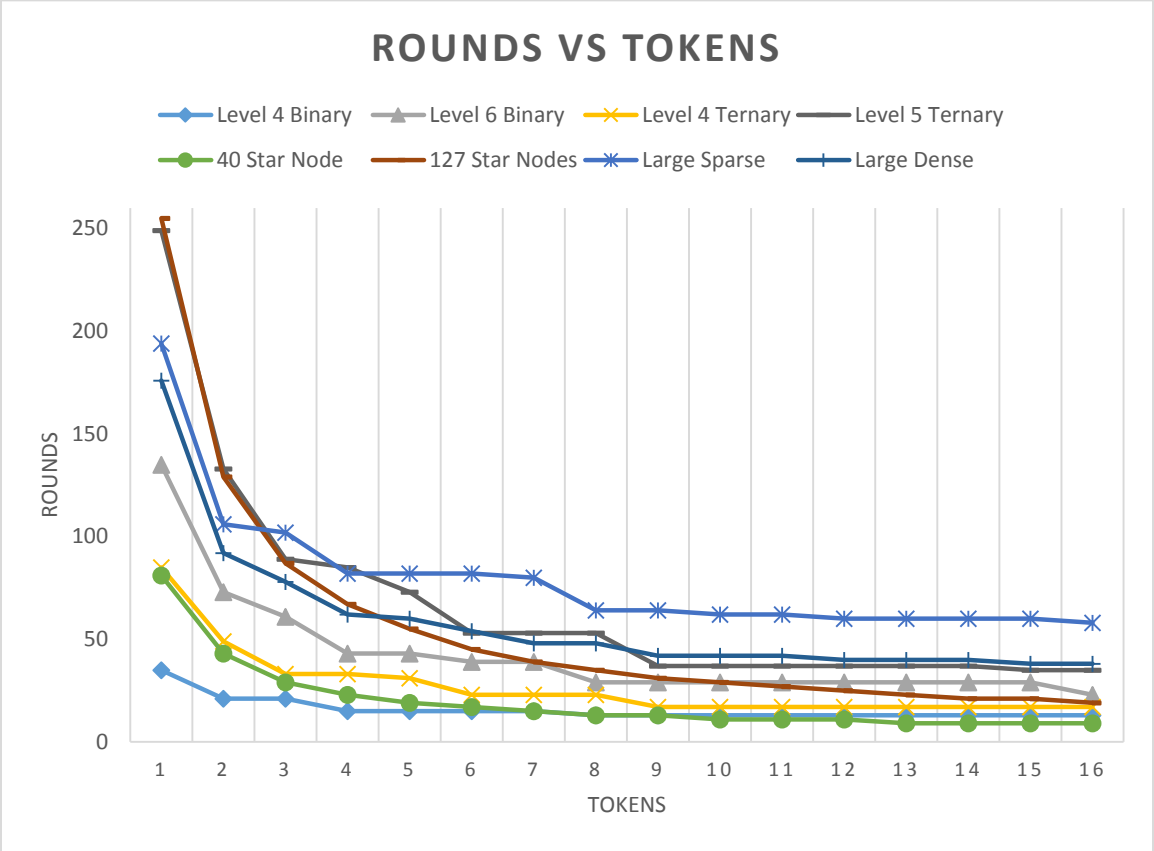

Figure 15: Rounds vs tokens

This chart shows how many rounds it takes for all of the BFS trees to construct, given 1 to 16 tokens. Rounds are on the vertical axis, and tokens on the horizontal. I included a couple examples from each network type. They all follow a similar path, with the largest differences between token amounts of 1 to 7.

The star networks follow a fairly smooth curve, due to the many different levels of network traffic. While the other network topologies tend to plateau for a few token amounts at a time, then drop. This isn't surprising considering the star network can pass the token around to a new node very efficiently, while the other network types commonly have tokens backtracking over nodes that have already received a token.

4.7 Traffic vs Tokens

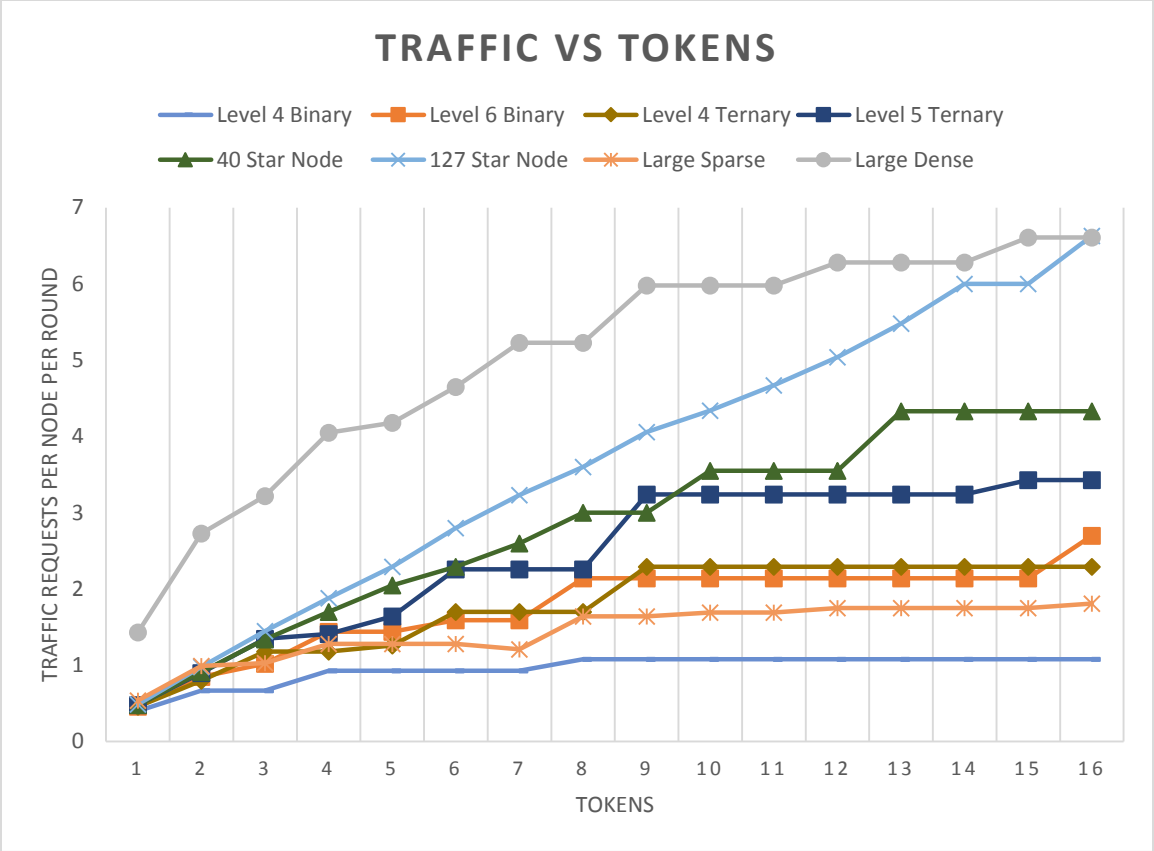

Figure 16: Traffic vs tokens

This chart represents the amount of network traffic for each token amount of 1 to 16 tokens. The vertical axis is BFS requests per node per round and the horizontal axis is token amount.

The large-dense and 127 star network stand out as the highest generators of network traffic. These networks topologies have more densely connected nodes than the others networks. This indicates that the more connected a graph is, the heavier the traffic will be. The chart shows that the token amount can control the traffic even for these dense networks. The biggest difference between sparse and dense networks is how much traffic is increased with each token added. The dense network adds a lot of traffic for each token, while the sparse network sometimes doesn't even increase with an extra token.

Chapter 5 - Conclusions

This thesis contains a new way to construct BFS trees that gives you control over the amount of network impact by choosing a certain amount of tokens to begin the algorithm. I ran the new multiple token algorithm through a custom network simulator across several types of network topologies using different amounts of tokens to confirm that the algorithm is performing as expected. The results show that the amount of network traffic can be chosen based on the token amount.

While constructing BFS trees in parallel is the fastest way to obtain a BFS tree from every node in the network, it requires a lot of network traffic in a short amount of time. In contrast, using the single token algorithm is very light on the network, but it can take a very long time to complete. With the multiple token method described in this thesis, you can choose a point somewhere in between the two extremes. In addition, the results show that if you know a little bit about the network topology, you can even target a specific maximum network limit and be confident you are forming BFS trees as fast as possible within that limit.

Chapter 6 – Future Work

Token Efficiency

In the current algorithm, when tokens make it back to the root, the root continues to pass the tokens around evenly. A possible improvement would be to have the convergecast for the main root BFS tree include the number of total children. When the root receives its children counts, it could begin spreading the tokens more towards the neighbors that have more children (or children that haven't finished convergecast yet). This would not make much difference for smaller networks, but could increase the speed for some networks where the tokens are having trouble reaching the deeper levels when the main BFS tree forms very unevenly.

Token Prediction Simplification

This thesis provides results for several types of networks, and it seems like there is some connection to certain aspects of network topology and the effectiveness of token amounts (i.e. binary tree increases are seen only one powers of two after the initial 5 tokens). It would be interesting if the amount of tokens required to limit the network traffic to some degree could be predicted by only having a few specifications of the network, rather than knowing the actual topology of the network. The ratio of edges to nodes is possibly the strongest indicator. Perhaps there is a formula lurking in this data that could simplify the choice of token amount regardless of the topology?

Unsynchronized Networks

Another area for exploration would be unsynchronized networks. This algorithm should work fine on unsynchronized networks, but the results (time to complete and network traffic) might vary.

Bibliography

[1] Stephan Holzer, Roger Wattenhofer; “Optimal Distributed All Pairs Shortest Paths and Applications”, 2012.

[2] Nancy A. Lynch; “Distributed Algorithms”; Morgan Kaufman Publishers, 1996.

[3] Robert W. Floyd; "Algorithm 97 (SHORTEST PATH)"; Comm. ACM, vol. 5, no. 6, p. 345, 1962.

[4] S. Warshall; "A theorem on boolean matrices", J. ACM, vol. 9, no.1, pp.11-12, 1962.

[5] Richard Bellman; "On a Routing Problem", Quarterly Applications Math., vol. 16, pp. 87-90, 1958.

[6] E.W. Dijkstra; "A Note on Two Problems in Connection with Graphs," Numerische Mathematik, vol. 1, pp. 269-271, 1959.

[7] Srinivasan, T., Balakrishnan, R., Gangadharan, S. A., Hayawardh, V; “A Scalable parallelization of all-pairs shortest path algorithm for a high performance cluster environment”, Parallel and Distributed Systems, 2007 International Conference on, Issue Date: 5-7, Dec. 2007.

[8] Gayathri Venkataraman, Sartaj Sahni, Srabani Mukhopadhyaya; “A Blocked All-Pairs Shortest-Path Algorithm”, SWAT '00 Proceedings of the 7th Scandinavian Workshop on Algorithm Theory, Pages 419-432, 2000.

[9] Joon-Sang Park, Michael Penner, Viktor K Prasanna; "Optimizing Graph Algorithms for Improved Cache Performance", IEEE Trans. Parallel Distrib. Syst., vol. 15, no.9, pp.769-782, 2004.

[10] Tomohiro Okuyama, Fumihiko Ino, Kenichi Hagihara; “A Task Parallel Algorithm for Computing the Costs of All-Pairs Shortest Paths on the CUDA-Compatible GPU”,

Parallel and Distributed Processing with Applications, 2008 ISPA '08. International Symposium on, Issue Date: 10-12, Dec. 2008.

[11] H. Djidjev, S. Thulasidasan, G. Chapuis, R. Andonov, D. Lavenier; “Efficient Multi-GPU Computation of All-Pairs Shortest Paths”, Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, Issue Date: 19-23, May 2014.

Curriculum Vitae

Graduate College
University of Nevada, Las Vegas

Michael Spencer

Degrees:

Bachelor of Computer Science, 2004

Thesis Title: Constructing BFS Tree Using Tokens To Balance Network Speed and Traffic

Thesis Examination Committee:

Chairperson, Dr. Ajoy K. Datta, Ph.D.
Committee Member, Dr. Lawrence Larmore, Ph.D.
Committee Member, Dr. Yoohwan Kim, Ph.D.
Graduate Faculty Representative, Dr. Emma E. Regentova, Ph.D.