COMP 490 Project Report: Broadcasting in NetworkX

by William Zijie Zhang

December 19, 2023

Abstract

Broadcasting is an information disseminating problem in a connected network of transmitting a message from an originator vertex to all other vertices as quickly as possible. It is well known that finding the broadcast time for any random vertex in an arbitrary graph is NP-complete. However, it has been proven that this problem can be solved in polynomial time for a certain class of graphs. The dissemination process is as follows: the originator begins by placing a series of calls along the communication lines of the network. Every time the informed nodes help the originator in distributing the message. Every call is assumed to take place in discrete units of time and only involves two adjacent vertices. The applications of broadcasting span many fields such as internet messaging, replicated databases, and network design. However, prior to this project, there were no readily available broadcasting algorithms through open source software.

1 Introduction

This project report documents the contributions I have made to NetworkX during the Fall 2023 semester at Concordia University, Montreal. The primary focus of the project was to bridge the gap between theory and implementation for broadcasting algorithms. However, before being able to implement some of these algorithms I first had to gain a deeper understanding of NetworkX and the problem we were trying to solve.

1.1 Why is Broadcasting challenging?

The goal of broadcasting is to find a sequence of calls from its vertices that can distribute a message as fast as possible. Although there are theoretical proofs that broadcasting is NP-complete for an arbitrary graph and a given originator, it can be helpful to gain an intuition on why the problem is so difficult to solve. The main challenge of broadcasting is rooted from the insane amount of possible calls but more importantly, it can be difficult to classify the quality of certain calls. For example, starting from the beginning, which of its neighbors should the originator distribute the message to at the first time unit? Given the lack of global information, many neighbors can seem like equally good options. The first idea that comes to mind would be that the originator should send the message to its highest degree neighbor so that it can also help distribute the message. However, this simple heuristic works only on some occasions and isn't able to distinguish between vertices that are mostly isolated and vertices that connect to other clusters of nodes.

In the broadcasting problem there is a clear trade-off between being able to verify an optimal broadcast time and connectivity in the graph. On the one hand, highly connected graphs have the ability to broadcast the message quickly as most vertices will be able to help the originator in passing the message. However, on the other hand, the large amount of connections makes it hard to compute the exact broadcasting time of a particular vertex. Since then, a lot of research has been made on broadcast graphs [HL19] which minimize the number of edges required while maintaining the lowest possible broadcast time.

1.2 Why NetworkX and what does it do?

NetworkX is a Python package for the creation, manipulation, and study of complex networks. The graphs in NetworkX are represented as a "dictionary of dictionaries of dictionaries" where the layers respectively store the nodes, adjacency information and edge attributes [HSS08]. In addition, the labels of simple graph nodes in NetworkX are stored as Python sets, which allow near instant lookup time but have the restriction that there can be no duplicate values. The main benefit of using NetworkX over other graph analytic library is that it is easy to learn and allows fast prototyping. For example, in a few lines of code, a user could generate a large random network, pass it to some algorithm of choice, and output visualizations of the whole process.

I wanted to also briefly mention why Python is perfectly suited for a graph analysis library. First of all, its simplicity allows flexible representation of networks along with concise expressions of graph algorithms. In addition, the growing Python open source community allows NetworkX to use and provide features from numerical linear algebra (using adjacency matrices of graphs) and plotting libraries (using Matplotlib and GraphViz).

2 Algorithms: Approximations and Heuristics

The graph algorithms in NetworkX are separated into many categories such as connectivity, isomorphism and trees. However, the category that best fits the broadcasting problem would be *"Approximation and Heuristics"* since most algorithms are unable to compute the exact broadcast time of a vertex or graph. Other types of approximation algorithms include many famously difficult graph theoretical problems such as the travelling salesman problem, finding Ramsey Numbers, and computing Steiner trees. Just as a clarification on these terms, approximations algorithms provide theoretical guarantees on being reasonably close to optimality while heuristics can be seen as intuitive steps that have no such guarantees. In the following section, we will be exploring both an exact algorithm and a heuristic for broadcasting.

2.1 Broadcasting in trees

The first algorithm I implemented is known as "finding the broadcast center" in trees. The broadcast center of a graph G denotes the set of vertices having minimum broadcast time. This algorithm was first introduced in an important paper titled: "Information dissemination in trees" [SCH81]. In this paper, the authors also proved that broadcasting from an originator in arbitrary networks is NP-complete as it can be related to the 3-dimensional matching problem. However, the algorithm for finding the broadcast centers of a tree is linear with respect to the number of vertices in G. The intuition behind this algorithm is to take advantage of unique paths between vertices to have guaranteed broadcast time of sub-trees of G. As a byproduct, the broadcast center algorithm can also calculate the minimum broadcast time BT of the tree. Perhaps non-trivially, the paper also details that the broadcast time from any originator in the tree is equal to the distance of that vertex from the closest broadcast center plus BT.

2.1.1 Explanation and Code

In the following subsection I will be explaining some interesting NetworkX and Python optimizations that were used for this algorithm. The complete implementation of the broadcast center algorithm can be found in my pull request: NetworkX#6928.

The broadcast center algorithm works as follows: label the pendant vertices with broadcast value of 0. Then iteratively label the pendant vertices of the inner tree in the following manner: sort in descending order the labeled neighbors of v and take the maximum of their values summed with the index. The intuition behind the *getMaxBroadcastValue* function is that if we were to broadcast from v we would first pass the message to neighbors with highest broadcast value so that they can help distribute the message faster to their own neighbors. Taking advantage of the low-connectivity in trees, we are guaranteed that the broadcast value of v is exactly the largest sum of *values[u]* $+ i$.

```
def _get_max_broadcast_value(G, U, v, values):
    adj = sorted(set(G.neighbors(v)) & U, key=values.get, reverse=True)
    return max(values[u] + i for i, u in enumerate(adj, start=1))
```

Once all vertices have been labeled with their corresponding broadcast values, it is time to find the broadcasting centers of the tree. In [SCH81] the authors proved that the broadcast center will always be in the form of a star graph. We also know that the last vertex that is being labeled will be the hub of the star graph. The intuition behind this fact is that the first broadcast from a center could be reversed without loss of generality. Therefore, by changing the order of broadcasts at the beginning, we can still maintain the same remaining sequence of calls. The *getBroadcastCenter* function performs the same sorting of labeled neighbors as above, however, now we are concerned with finding j the first index that would give a broadcast time equal to BT.

```
def _get_broadcast_centers(G, v, values, target):
    adj = sorted(G.neighbors(v), key=values.get, reverse=True)
    j = next(i for i, u in enumerate(adj, start=1) if values[u] + i == target)
    return set([v] + adj[:j])
```

The following code section is used to find the broadcast time of a tree. It uses the broadcast centers BC and minimum broadcast time BT from the previous algorithm. We know that the broadcast time of a graph corresponds to the maximum broadcast time of all its originators [HL19]. Therefore we have to calculate the distance of all vertices from the broadcasting center. An efficient way to do that is to first compute all the distances from the broadcasting centers and store them in $distFromCenter$. Then we can easily compare the distance of a vertex with all the centers and keep the distance from the closest center to that vertex. Finally, we return the largest distance in $distFromCenter$ summed with the minimum broadcast time BT from the broadcast center algorithm.

```
    BT, BC = tree_broadcast_center(G)
    dist_from_center = dict.fromkeys(G, len(G))
    for u in BC:
        for v, dist in nx.shortest_path_length(G, u).items():
            if dist < dist_from_center[v]:
                dist_from_center[v] = dist
    return BT + max(dist_from_center.values())
```

2.2 Tree-Based Algorithm (TBA)

The second algorithm I tried to implement was a heuristic algorithm that computes a maximum bipartite matching between informed and uninformed nodes at every discrete time step. This heuristic performs exceptionally well in many classes of graphs, namely the *de Bruijn* and grid graph [HS06] , which are actively used in the design of efficient networks. However, because of time constraints, namely with the review of the broadcast center algorithm, I wasn't able to complete the implementation of this heuristic.

The TBA algorithm is as follows: vertices are first separated into two regions, the bright region representing informed nodes and the dark region representing uninformed nodes. The authors also introduce the concept of a dark border which refers to uninformed nodes that are neighbors of informed nodes. The main idea behind the TBA heuristic is to estimate the broadcast time of nodes in the dark border. The algorithm then calculates a *maximumNumberWeight* matching between vertices in the bright region and vertices in the dark border. This matching must both try to maximize the total estimated broadcast time and the number of such matchings. The paper introduces two matching methods: *sortMatching* and *listMatching* which have trade-offs between time and space complexity. In addition, there could be many vertices from the bright region that connect to a single vertex in the dark border, hence affecting the weight count of the latter vertex. To deal with this overestimation, a refinement can be applied that would divide the expected weight of a node in the dark border by the number of its parents.

3 Graph Generators

In NetworkX the user is also able to generate many graphs such as the Erdős-Rényi random graph or the Harari connectivity graph. These graph generators can receive parameters that dictate the number of vertices and edges along with the probability of connecting two random vertices in the resulting graph. This allows NetworkX users to rapidly test the validity of a conjecture or run experiments on a wide variety of graphs. The following section will be detailing two graph generators that were missing from NetworkX and their importance in the theory of graphs.

3.1 Cube-Connected-Cycle

The first graph generator we are concerned with is a network topology that was designed for usage in parallel computing [PV81]. The cube-connected-cycle is formed by taking a hypercube with 2^n vertices and replacing each vertex with a cycle of length n. This creates a graph that is 3-connected while maintaining the connectivity properties of the hypercube. Interestingly, in the context of broadcasting, the cube-connected-cycle is known for having close to optimal broadcast time. To construct this graph in NetworkX, we would begin by storing vertices as tuples of the following form: (x, y) where $0 \leq x < 2^n$ and $0 \leq y < n$. Clearly, x represents the index of the original hypercube and y represents the index of the newly inserted cycle.

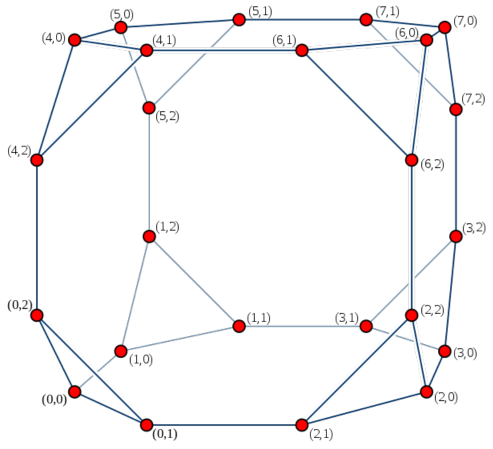

Figure 1: The Cube-Connected-Cycle with $n = 3$

The next step would be to add the edges connecting these vertices. The edges that belong to the cycles are simple, we must create the following maps: $(x, y) \rightarrow (x, y + 1 \bmod n)$ and $(x, y) \rightarrow (x, y - 1 \bmod n)$. Now we must map each vertex through its correspondent accross the original hypercube. This can be done by the following map: $(x, y) \rightarrow (x \oplus 2^y, y)$ where $\oplus$ denotes the bitwise "exclusive or" operation (reminder: we put 0 to denote same binary digit between the operands and 1 if they are different). In the example above with $n = 3$, the tuple $(2, 2)$ goes to $(6, 2)$ since "0010" XOR "1000" gives us "0110" which is simply 6 in binary.

3.2 Kneser Graph

The following generator I implemented is called the Kneser graph. It was developed by the german mathematician Martin Kneser and popularized by László Lovász when he proved the Kneser conjecture [GR01] with topological methods. Due to it's symmetrical construction, it is natural to think that the Kneser graph would also be a good topology for the broadcasting problem. However, Aram Khanlari, a recently graduated Master's student of Professor Harutyunyan, tried to find such broadcasting properties in the Kneser graph without success.

The Kneser graph is the graph whose vertices are the k-element subsets of n elements, and two vertices are connected if their corresponding subsets are disjoint. It follows clearly that the Kneser graph will have $\binom{n}{k}$ vertices each with degree $\binom{n-k}{k}$. From the handshake lemma we can conclude that the number of edges in the Kneser graph will be $\binom{n}{k} \cdot \binom{n-k}{k} \cdot \frac{1}{2}$. The well-known Petersen graph with 10 vertices and 15 edges is a special case of the Kneser graph with $n = 5$ and $k = 2$ (see image below). In the following paragraph I will be detailing my implementation of this graph generator. You can find the full code sample in my pull request: NetworkX#7146.

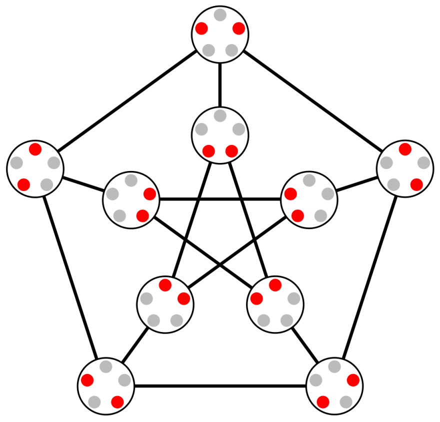

Figure 2: The Kneser Graph $K(5, 2)$

The vertices can easily be generated by making use of Python iteration tools and taking all combinations of length k from the list of values between 0 and $n - 1$.

```
G = nx.Graph()
vertices = list(itertools.combinations(range(n), k))
G.add_nodes_from(vertices)
```

Now we need to add all the edges in the Kneser graph. There is only one rule we need to follow so the implementation is quite simple. First we loop through all vertices we obtained in the previous step. Then we obtain the complement of v by removing all k-elements of v from the n element universe set. Then we connect v to all other valid k-element subsets that are in the complement. Note that if $n < 2k$ then the Kneser graph will be disconnected because no sets are completely disjoint. See the code example below:

```
for v in vertices:
    complement = set(range(n)) - set(v)
    for t in itertools.combinations(complement, k):
        G.add_edge(v, tuple(t))
return G
```

4 Future Work

This section will detail potential algorithms that remains to be added to NetworkX. In the search of a broadcast graph with 24 vertices and 35 edges, Gabriel Talih explained that the third and final step of the plan was to compute the exact broadcasting time of a generated graph. Despite the fact that this problem is NP-hard, it remains tractable for relatively smaller graphs. In addition, knowing the target broadcast time, we can pass an upper bound on the broadcast time and greatly reduce the amount of computations. Using the fact that all broadcast schemes must be spanning trees, we can simply check the broadcasting time of every spanning tree using the broadcast center algorithm. In [Knu11], Donald Knuth details how to generate all spanning trees of a graph in "Algorithm S" on page 24 of section 7.2.1.6. A small optimization could be to classify all the spanning trees into isomorphism classes and then calculate the minimum broadcast time of the representatives of those classes.

Naturally, a next step for this project would be to complete the implementation of Bin Shao's Tree-Based Algorithm (TBA) along with verifying the performance of the heuristic in many network topologies. There are also two more missing graph generators that have efficient broadcasting properties and remain to be implemented: the shuffle-exchange and the *de Bruijn* graph. In addition, some work can be done on new visualizations of broadcasting heuristics, continuing Cedric Paradis' project on the topic. Finally, a lot of approximation algorithms remain to be explored and implemented.

5 Summary

This project has brought significant progress on open source implementations of broadcasting algorithms. The idea behind the project came up during a Python conference where I was introduced to the NetworkX library and some of its maintainers. During the summer of 2023, I was brainstorming some project ideas and found a unique opportunity to combine Professor Harutyunyan's research in broadcasting and my desire to contribute to open source software. At the beginning, I spent a lot of time doing literature review on broadcasting research papers. I discovered many related topics such as multiple originator broadcasting, the search for broadcast graphs and even attended Aram Khanlari's master's thesis defence. I also learned the fundamentals of NetworkX and the importance it played in network analysis. Soon thereafter, I stumbled upon the seminal paper: "Information dissemination in trees" [SCH81] which laid the foundation for future broadcasting research. I instantly knew that the algorithm presented inside the paper would be a good starting point for my project.

Throughout the following months, I had a lot of discussions with Daniel Schult, one of the maintainers of NetworkX, and managed to implement and properly document the broadcast center algorithm. In this project report, I detailed some important code segments along with the intuition behind why the functions work. The implementation ended up being quite elegant due to many shortcuts introduced by my reviewer. In the following section, I briefly go over the TBA heuristic which I didn't manage to successfully implement due to time constraints. However, I found that such a heuristic definitely had practical significance as it provided close to optimal broadcasting times in an efficient manner. While waiting for code reviews, I also explored the topic of graph generation while discovering graph topologies used in broadcasting. In the project report, I first give a brief historical introduction to two network topologies, then I explain how to generate them for arbitrary large dimensions. Finally, I conclude the report by outlining concrete steps I would take if I had to continue this line of work.

I want to conclude this report by thanking my supervisor Professor Hovhannes Harutyunyan for which his guidance and support have been invaluable towards the completion of this project.

References

[GR01] Chris Godsil and Gordon Royle. Kneser graphs. In *Algebraic Graph Theory*, chapter 7, pages 135–161. Springer-Verlag, New York, 2001.

[HL19] H. A. Harutyunyan and Z. Li. A simple construction of broadcast graphs. In D. Z. Du and C. Tian, editors, *Computing and Combinatorics. COCOON 2019*, pages 240–253, Cham, Switzerland, 2019. Springer.

[HS06] Hovhannes A. Harutyunyan and Bin Shao. An efficient heuristic for broadcasting in networks. *Journal of Parallel and Distributed Computing*, 66(1):68–76, 2006.

[HSS08] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gäel Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA, USA, August 2008.

[Knu11] Donald E. Knuth. *The Art of Computer Programming*, volume 4. Addison-Wesley, Boston, MA, USA, 2011.

[PV81] Franco P. Preparata and Jean Vuillemin. The cube-connected cycles: A versatile network for parallel computation. *Communications of the ACM*, 24:300–309, 1981.

[SCH81] P. J. Slater, E. J. Cockayne, and S. T. Hedetniemi. Information dissemination in trees. *SIAM Journal on Computing*, 10(4):692–701, 1981.