Comparison of Different Search Algorithms in Unstructured Decentralized Peer-to-Peer Networks

CSC 466 - Overlay and peer-to-peer Networking

Spring 2025 Semester Project

GitHub | Website

Mathew Terhune
V00943466
Computer Science
University of Victoria
Victoria BC Canada
mterhune@uvic.ca

Ali Gaineshev
V000979349
Computer Science
University of Victoria
Victoria BC Canada
ggaineshev@gmail.com

Holly Gummerson
V00986098
Computer Science
University of Victoria
Victoria BC Canada
hgummerson@uvic.ca

1.INTRODUCTION

In the proposal phase of this project, we aimed to compare search algorithms such as random walks, flooding and other query techniques in unstructured decentralized peer-to-peer (P2P) networks. Our group wanted to focus on how network topology and querying mechanisms influence search efficiency, scalability and resource consumption. To accomplish this, we planned to use a network simulation tool to replicate the same functionality on a smaller scale P2P network based on Gnutella architecture. This approach was introduced to allow us the ability to conduct tests and control the environment. Further, we planned to implement and compare multiple search methods and document metrics such as query success, latency and bandwidth consumption. Based on these metrics, we ultimately intended to evaluate the effectiveness of each approach, and possibly identify optimizations. In this phase, we effectively chose our topic, and under the professors guidance NS3 as the simulation tool. This initial phase allowed us to complete research on Gnutella like architecture, and choose an acceptable angle to gain proper insights. Additionally, we completed a basic deliverable schedule to ensure that our proposed research could be completed per the scope of the class.

For the first biweekly update, we aimed to gather information, download initial dependencies and choose preliminary metrics for our tests. We took inspiration from research done by Dorrigiv et al. (2007), looking into different search algorithms in peer to peer networks including k-random walkers, and flooding [2]. We further researched the idea of Random walks in unstructured P2P networks using research proposed by Gkantsidis et al. (2006), which showed promising results on certain topologies and network types. The final algorithm we chose to look into was introduced to us by Cuenca-Acuna et al. (2003) regarding the use of Gossiping. With these initial papers, we gathered preliminary information on the methods we planned to implement in our simulation. During this time, we were able to download the NS3 software and begin to select the metrics we wanted to track. During this time, there were significant challenges due to NS3 being complicated software. When starting to work with NS3, we had to overcome build, config and general environment errors to allow the software to run on our local machines. Along with this, there is a significant learning curve to writing and running simulations which had to be learned through documentation and online resources. Due to this initial roadblock, we highlighted the fact that the metrics were proposed, and subject to change as we got familiar with NS3. Theoretically, the concepts and metrics we chose were applicable; however, converting them to our custom built simulation and architecture was a complicated task. The following phases are a result of trial and error through different roadblocks.

Following the first biweekly update, we completed the midterm phase of this project. The aim of this phase was to establish a solid foundation for our project structure and further refine our approach to the problem. This included identifying the key functionalities needed as well as validating our initial assumptions regarding NS3. By referencing work completed by Hu and Liu (2010) as well as Frigo et al. (date) we started the application and packet structure for our custom simulation [4][5]. Further, by studying The Annotated Gnutella Protocol Specification v0.4 we designed the initial packet structure, with slight modifications due to the scope and abilities of NS3 [6]. To test and verify our packet structure, we developed an initial chain structure and baseline sending and receiving. This phase came with a host of problems acclimatizing to NS3 and its routing protocols. By completing this basic version of the code, we were able to move on to further refining and more complex topologies.

In the third biweekly phase, we had set goals to develop a comprehensive test suite, introduce dynamic nodes, implement network flooding and finalize our metrics. Throughout this period we were able to generate a binary tree topology which allowed us to test more thoroughly and validate flooding. There were a few challenges to this portion including working NS3 built in netdevices to multiple IP addresses, making multiple queries to be sent and received incorrectly. During this time we also worked on major packet structure changes to accommodate our growing application and future metrics. This included adding a sink node field, as well as a path vector for a query hit. These changes allowed us to test flooding and verify the topologies connections and sending were successful. The final work we completed in this phase was the beginning of dynamic nodes, which included the important discovery of NS3 scheduling rules. We discovered that due to the nature of the simulation, we would instead have to schedule a node to disable and re-enable later. This phase was very important in discovering key issues and roadblocks in our initial ideas for the project. After working in depth on the code, we planned to finalize gathering the statistics, continue with dynamic node churn and implement different strategies to compare to flooding.

In this final phase, we refactored our code and were able to successfully implement two more search methods to the simulation. We choose k-random walkers which were in the proposal, as well as normalized flooding introduced to us by Dorrigiv et al. (2007) [2]. Using this research as a guideline, we modified these two search algorithms for our application and tested them for correctness. Further, we created more topologies such as cluster, k-regular and megagraph to compare these methods in different situations. Finally we gathered statistics and initial insights based on our results. This phase came with challenges regarding netAnim (NS3 built in animation tool), and its manual node placement barriers. Further, our initial statistics were interesting, however did not allow us to create the visuals we intended. Using the feedback given from our presentation and the errors we found, we completed the goals set in our proposal, leaving room for future work and improvements.

2. BACKGROUND

We believe that a thorough understanding of the underlying structures and architecture of unstructured decentralized P2P systems is essential for this project. This includes formal and semiformal explanations of Gnutella, query search optimizations and ns-3.

2.1 Unstructured Decentralized Peer-to-Peer Networks

Peer-to-peer networks are a distributed network of participants (often called peers) directly sharing resources and communicating with each other, without relying on a central server. This allows for the device to act as both the client and server. Each computer or peer in the network can directly communicate with other peers to share files, data or processing power. P2P networks are commonly used for file sharing applications, some forms of online gaming and certain decentralized applications. In 1999, Napster was the first system to utilize the need for avoiding a central server, and instead go through multiple peers that already held the file locally [7]. This type of file sharing is self-scaling, because of the aggregate download capability. This system was P2P, but still included a centralized search facility based on file lists provided by the peer upon joining the system. After Napster was shut down due to legal issues, new decentralized systems grew in popularity. An unstructured decentralized P2P system is one that distributes both the download and search capabilities. In these systems, no rule defines where data is stored and the network topology is arbitrary. Peers are totally equal in the sense that they are not taking advantage of heterogeneity [1]. The advantages of this purely decentralized system include increased resilience and cost-effectiveness, due to eliminating reliance on a single point of failure of authority. If one node fails, the network can continue to operate as long as other nodes remain active. These networks don't require expensive central servers or infrastructure, making the cost reduced and distributed among the participants [8]. Another advantage of these systems is the ability to self organize, allowing peers to leave and join the network freely.

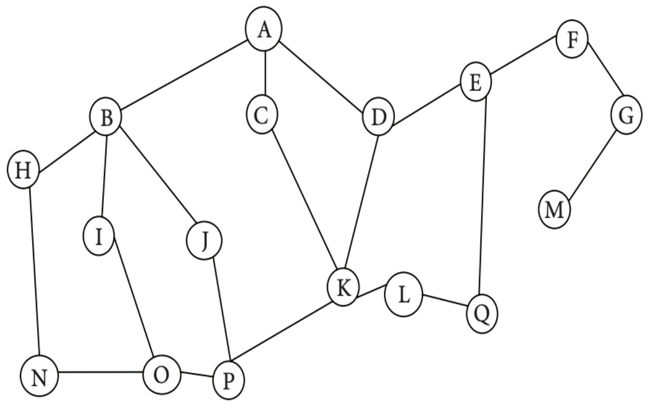


Figure 1. An Example Unstructured P2P Network [18]

2.1.1 Gnutella

In 2000, Gnutella was created and used an unstructured overlay with flooding and query routing. It was created after Napster's copyright legal issues, and attempted to distinguish between exchanging music and the general sharing of files [9]. In this overlay network, each peer that runs the Gnutella software knows about some set of machines that are also running Gnutella. This idea corresponds to graph edges creating ways to communicate. For the purposes of this project we will focus on the search ideas in this kind of network, omitting connectivity functions.

2.1.2 Flooding

When a peer in a Gnutella network wants to find an object or a file, it will send a query message outwards into the network in search of that object. This message specifies the file's name and is sent to its neighbours in the overlay. If one of the neighbours has the file or object specified in the message, a response is sent back with a query response message, specifying where the object can be downloaded. The flooding (F) aspect comes when a peer cannot satisfy the query, and it forwards the message onto each of its neighbours except for the sender. This process repeats until either a query hit is found, or the TTL runs out (ensuring the flood does not continue indefinitely). In Gnutella, each node maintains a record of the query messages it has seen recently. This allows for smarter forwarding and faster cut off for forwarding loops. Furthermore, this allows the query hit a path to forward the response upstream to the neighbour that originally sent the query message.

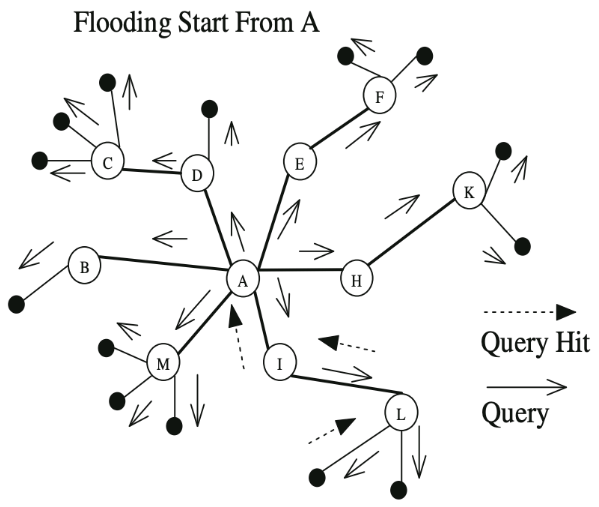


Figure 2. Sample flooding algorithm from A [17]

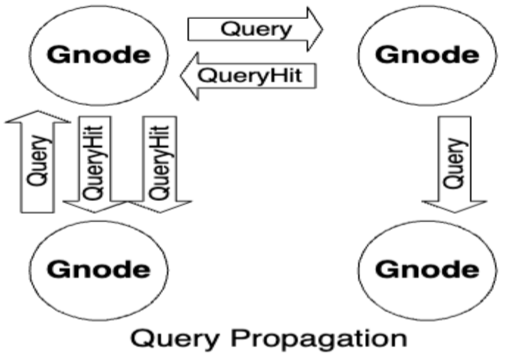


Figure 3. Simple Gnutella query propagation [16]

Although flooding has the property that it is guaranteed to find the desired object in the fewest possible hops, its forwarding mechanism does not scale well as a system grows [9]. Flooding assumes no knowledge about the topology of the network or resource distribution which is beneficial for dynamic systems which are continuously changing. Due to the load on each node growing linearly with the total number of queries in the system, which is growing with the system size, this approach quickly becomes a scaling issue [7]. There has been research done to optimize search in unstructured decentralized peer-to-peer systems which follows either topology or search algorithm changes. In the next section we will explore the background of search strategies used in this project.

2.2 Proposed Search Optimizations

After Gnutella was produced the negative aspects of flooding on scaling were revealed. From there optimizations and studies were made finding ways to improve the scalability and limit the overhead in these decentralized systems. In the scope of our project, we choose to modify two approaches which we will discuss more in detail below. The first is normalized flooding, followed by random walks. These approaches take stances to reduce messages sent compared to traditional flooding.

2.2. 1 Normalized Flooding

Normalized flooding works the same as pure flooding where each node, if it is not a query hit, forwards the message to all of its neighbours except the sender. The only difference in normalized flooding is that the node will only send to a subset of its neighbours. Let δ be the minimum degree, or number of neighbours. If the node has more than δ neighbours, then it sends the message only to δ nodes in the neighbourhood [2]. During this process, the subset of neighbours is chosen uniformly at random.

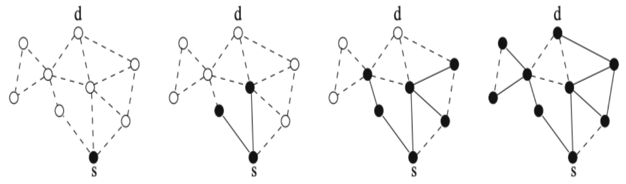


Figure 4. Normalized flooding search strategy (δ = 2) [12]

This search algorithm aims to reduce the amount of packets being sent in the system, and reduce duplicates. A study by Rani Kumari et al. (2008) found that random walks showed similar qualitative behaviour to normalized flooding, except for a lower resilience to clusters in networks [12]. Further the normalized flooding showed its best performance when the amount of nodes joining is high, and leaving is low. To recap, the advantage of this algorithm is that the node will send to at most δ neighbours, which reduces message overhead. Some possible disadvantages to this include the randomness of node selection, as well as the possibility that low degree nodes are left out from receiving queries (affecting fairness) [13]. We will be referring to it as NF throughout the paper.

2.2.2 Random Walks

Random walks (RW), or Markov chains, were an alternative that was explored to flooding. The general idea is that instead of forwarding a request to all of the nodes' neighbours, it will instead be forwarded to a random neighbour from the list. This algorithm is thought to scale well with the network as the method has small message complexity [2]. The random walk or Markov chain has many theoretical properties which make it an interesting candidate for P2P networks. This includes the memorylessness property which allows for previous states to be irrelevant in predicting the subsequent states.

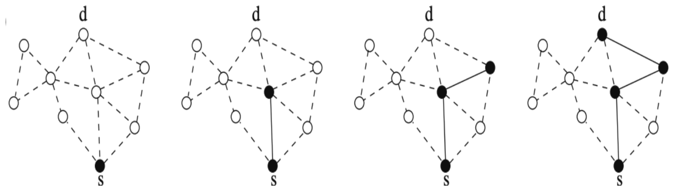


Figure 5. Single random walk search strategy [12]

The idea of random walks in these networks has been studied by multiple authors. Many variants have been produced from this, including two-level random walks. In this algorithm a querying node selects k1 random walks with TTL1=1. When the TTL1 finishes, each thread will then generate k2 threads which perform k2 random walks from that node with TTL2=I2 [13]. This scheme explained by Thampi and Sekaran generates less duplicate messages, but has longer searching delays. Another more simplistic method is the k independent random walker algorithm, which we choose to implement for this project.

2.2.3 Random Walkers

In this algorithm, simultaneous random walks are simulated from the same origin. The search is successful once the file is located by anu of the individual random walks. This method can also be known as the k-walkers algorithm. If the object is ont found it is forwarded to a randomly selected neighbour until the TTL is terminated. They suggested that this algorithm is a sort of middle ground between flooding with high overhead and a single random walk which has a lower success rate. The advantages are that this algorithm will only send out k query messages, reducing message overhead and improving the scalability of the system. When the network grows, the message overhead will not grow linearly with it. Secondly, having the multiple walkers running parallel will increase the chances of encountering a query hit, without having to contact every node. Further, if one walker fails, there is resilience in the fact that others can succeed. Some of the disadvantages of this would be the lack of guarantee in discovery. The random walk's success depends on the random path and TTL values which do not guarantee a hit. Moreover due to the memoryless property and random factors, there is a chance that walkers may overlap nodes and waste resources [15].

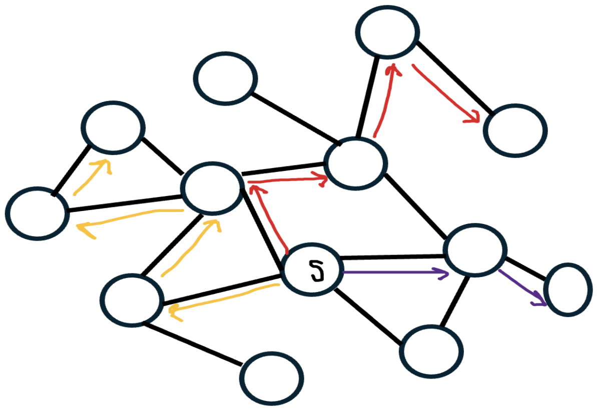


Figure 6. Simple random walkers algorithm with k=3 from s

2.3 NS-3

Ns -3, first released in 2008, is a discrete-event simulator which is targeted toward research and educational use. It is distinct from its predecessor ns-2 due to its modern and extensible platform. The goal of this software is to develop a free and open source environment suitable for networking research needs, aligned with modern networking research [3]. The code is written in C++, and contains a network animator to give direct visuals for simulations run. Ns-3 supports research on both IP and non-IP based networks, as well as a real-time scheduler which were factors when considering our environment. We chose this software for the project due to its commitment to a solid simulation core that is well documented and easy to debug.

2.3.1 NetAnim

NetAnim is an offline animator tool provided by Qt toolkit which animates a previously executed simulation using a XML file [3]. This file is provided after running a simulation on ns-3, and generating a trace file. This software allowed us to watch the packet transfers and network play out. It displays nodes, links and packets being sent and received between each. Along with this, it provides information about time and general statistics which could be used for analysis. Throughout this project, we used NetAnim to display our graphs and ensure consistency between our nodes. Further it enhanced our debugging and understanding of the networks we were creating. The real-time graphical feedback particularly helped us troubleshoot packet routing issues and gave us confidence that our implementation matched our expectations. We will be utilizing the simulations created by NetAnim throughout the paper to demonstrate key concepts and our results.

2.3.2 Sending and Receiving

Ns-3 supports the sending and receiving of data packets through the ns3::PacketSocket class, as well various ns3 public classes. To send a packet, the data is first wrapped inside a Packet object. In our project we proceed to attach a custom packet header which holds specific metadata. We do this to ensure that regular Packet functionality is available, while being able to customize with our own logic. Each node in the system has a socket bound to one of its assigned IP addresses. This socket is used for both sending and receiving packets. To ensure that we can listen to incoming messages, we register it using a SetRecvCallback(). This function allows the node to listen to incoming packets, and handle the logic using our custom logic. To send packets, SendTo() is used, which specifies the destination IP and port. The packet can be forwarded through this logic only if the IP and socket are correctly bound. Using the provided classes from ns-3 and integrating it with our logic allowed us to mimic the decentralized message passing.

2.3.3 Socket Binding

Here we will outline more in depth the use of ns-3's socket binding in our application.

```
void
P2PApplication::StartApplication()
{
   Ptr<Node> node = GetNode();
```

```
    Ptr<Ipv4> ipv4 = node->GetObject<Ipv4>();

    // first interface is a loopback one. Avoid it.
    for (uint32_t i = 1; i < ipv4->GetNInterfaces(); i++)
    {
        TypeId tid = TypeId::LookupByName("ns3::UdpSocketFactory");
        Ptr<Socket> socket = Socket::CreateSocket(GetNode(), tid);
        Ipv4Address address = ipv4->GetAddress(i, 0).GetLocal();

        if (address != Ipv4Address::GetZero())
        { // Only bind to valid addresses
            InetSocketAddress local = InetSocketAddress(address,
m_port);
            socket->Bind(local);

socket->SetRecvCallback(MakeCallback(&P2PApplication::RecievePacket
, this));
            m_sockets.push_back(socket);
        }
    }
```

Figure 8. Socket functionality in application.cc

Outlined in figure 8 is the code which is called upon the application starting. Here we use the Socket class to create a UDP socket for the current IP. This socket can both send and receive packets. In our project, we skip loopback interfaces to ensure the sockets will only bind to real used IPs. Once bound, each socket is set to listen to incoming packets using the SetRecv CallBack as outlined above. This allows incoming packets to get forwarded to our handler RecievePacket function to be dealt with in a custom fashion. By utilizing the ns-3 built in classes to allow our application

2.3.4 Nodes in ns-3

To further understand how our code is processed in the software, we will dive into how nodes are structured in ns-3 and how we utilized it for our custom application. Each simulation node outlined by figure 9 is a container with multiple components which give it its properties.

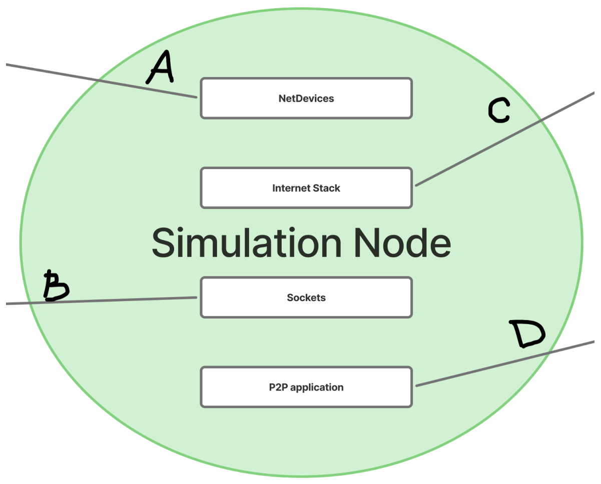


Figure 9. Diagram of a simulation node in ns-3

A. NetDevices simulate the physical network cards and manage a data-link layer transmission.

B. Sockets as outlined in the previous subsection facilitate the sending and receiving of packets across the differing interfaces. Sockets are bound to each Ipv4 interface, each of which are used to send and receive via UDP. They bind to a specific IP and Port, and are essential to our simulation packet movement.

C. Internet Stack is used to provide IP, UDP and routing capabilities to the node. It further handles Ipv4 interfaces, each of which is assigned a unique IP.

D. P2PApplication is our custom application code, which handles our logic such as search algorithms, sockets, message reception, neighbour tracking and local IPs. It handles all P2P behaviour and is installed on each node.

Having each node structured like this enables functionality for the essential portions of our simulation.

2.3.5 Node Connections in ns-3

In ns-3 each connection between nodes is typically modelled using a point-to-point link, which represents a direct communication channel between two nodes. This idea is similar to a virtual ethernet cable. When creating a simulation, it is important to note that each node can have multiple network interfaces, and each interface is assigned a unique IPv4 address. In simple terms, a node will have multiple IP addresses if it's connected to more than one link (which is common for more complicated topologies). Figure 10 outlines this idea, as node 0 uses one IP address to talk to node 1 and a different IP to communicate with node 2.

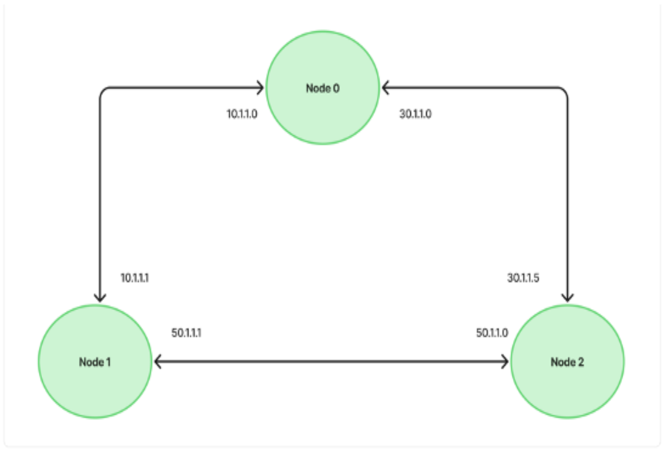


Figure 10. Example node connection in ns-3

In our implementation, each node stores a list of its neighbours and its own IP addresses to locally keep track of its connections.

3. OUR GOAL

3.1 Description

The primary goal of our project was to study search algorithms in unstructured decentralized P2P networks. We wanted to understand the core differences in performance, scalability and resilience between common search strategies. To explore this, we built a Gnutella-like simulation on ns-3, implementing and comparing: flooding, random walkers and normalized flooding. We created a custom packet, application, topologies and various utility functions to simulate the unstructured decentralized network. By doing this, we were able to gather metrics and analyze the differences in performance. The following sections outline the simulation we built, as well as the custom topologies. We will detail the implementation as well as design and scope issues faced.

3.2 Scope

Given the potential scale of this project, we chose to limit the scope for deliverables to remain feasible. While building this project, we found ourselves modifying and taking out certain components which initially seemed achievable but later proved to be too time consuming given our constraints. This section outlines the final scope of our projects

3.2.1 Graphs

The topologies we used in our simulations were carefully selected to maintain simplicity and uniformity. Our group chose to place the following constraints on our graphs.

1. Connected: Every node in the graph is reachable from any other node, ensuring there are no isolated components.

2. No Forests: We exclude disjoint sets of trees or multiple disconnected components

3. **Bidirectional Connections**: All connections are undirected, meaning that communication can occur in both directions.

4. **Acyclic**: All graphs disallow cycles or loops.

3.2.2 Search Only

As our primary goal of this project was to study the search algorithms in querying we choose not to include basic gnutella like behaviour such as Ping and Pong which are involved in the peer discovery process. Due this being a simulation and time constraints, we choose to simply have the peers connected upon the simulation start. Further, ns-3 was new to the group so the choice to avoid that and focus only on the searching mechanisms allowed for resources to be honed into that area.

3.2.3 ns-3 Restrictions

Working with NS-3 gave us scope in our original ideas to make the project something that is feasible given the time frame. There are two main reasons that we had to modify our original expectations. The first being that we were new to the software, and the programming needed came with a larger learning curve then expected. C++ (the language which we programmed in), the different classes which we mobilized, and how simulations are actually run gave us a lot to learn in a short time span. Because of this we were made to lessen the workload and re-think what project could accomplish. Secondly what we were building was a simulation, meaning that most of the functionality was predetermined. Given that we were creating a peer to peer network it was difficult to implement dynamic behaviour due to the scheduling needed. We initially wanted to have nodes joining and leaving freely, but with the NS-3 constraints and difficulties with the peer-to-peer links we chose different methods.

3.2.4 Algorithms Chosen

Due to time constraints and the many possible ways we could learn for the algorithms chosen, we had different variations from when we initially planned to the final product. Due to the time constraints and programming abilities we have we opted for flooding, random walkers and normalized flooding. These three algorithms were generally more simple and gave us time to test with different topologies to ensure the functionality was correct. If we had more time and experience working with ns-3 we would have chosen either expanding ring, or modified the random walks to have a bias like described in certain papers. We wanted to ensure a working and correct product in this paper thus opted for these three. Shrinking the scope for this project allowed us to have working algorithms that still tested different network properties.

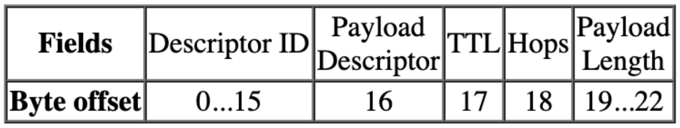
4. CUSTOM GNUTELLA-LIKE APPLICATION

The main portion of our project consists of a custom built Gnutella like application, which we used to simulate different search methods. To accomplish this we used NS-3, and wrote different files for routing logic, packet structure, topologies and simulation. In this section we will discuss what we built in detail. This includes implementation as well as design choices made. By doing this we hope to give a strong foundation to our results and gathered statistics later in the paper. The code for this application can be found on the GitHub linked on this paper, as well as the website.

4.1 Packet Structure

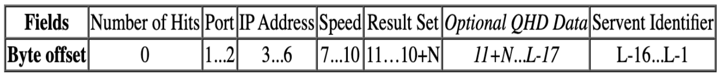
To send and receive packets in a Gnutella like way, we created a custom packet structure to hold metadata and control internal logic. Our packet information is contained on a separate file and is attached to each Packet object as described in an earlier section. Using The Annotated Gnutella Protocol Specification v0.4 as a guide, our group implemented the logic that allows us freedom in our routing logic [6]. Since this project is based on Gnutella, our initial attempt was to remain close to that implementation. After working with the code we decided to modify some parts to better suit our needs, and make further additions easier. These parts include avoiding data transfer and not including ping/pong functionality. Further we added additional fields and modified the last hop areas to make routing more efficient and simplified.

In these packets, the payload contains different fields depending on a ping, pong, push, query or query hit. The descriptor ID is a 16-byte string that uniquely identifies the descriptor in the network and allows for detection of cycles, the payload descriptor the message type given in bytes (ex. 0x00=Ping). The TTL is time to live and represents the number of times a descriptor will be forwarded by Gnutella servants before its removal [6]. The Hops represent the number of times the descriptor has been forwarded. Finally the payload length details the length of the descriptor immediately following the header. For Simplicity we opted to modify this idea, and combine the descriptor metadata into the descriptor header.



Fields	Descriptor ID	Payload Descriptor	TTL	Hops	Payload Length
Byte offset	0...15	16	17	18	19...22

Figure 11. Gnutella Descriptor Header structure from [6]



Fields	Number of Hits	Port	IP Address	Speed	Result Set	*Optional QHD Data*	Servent Identifier
Byte offset	0	1...2	3...6	7...10	11…10+N	*11+N..L-17*	L-16...L-1

Figure 12. Gnutella Query Hit Descriptor Payload [6]

In our modified packet class, we have increased the size of the descriptor header, and included more general fields. Table 1 is a table representation of the packet we created.

Table 1. Custom packet structure

Field	**Type**	**Bytes Used**
payloadDescriptor	uint8_t	1
descriptorId	uint32_t	4
senderIP	Ipv4Address	4
destIp	Ipv4Address	4
sinkNode	uint32_t	4
m_ttl	uint8_t	1
hops	uint8_t	1
path	vector<Ipv4Address>	4 * N

As seen in table 1, we took the important fields such as descriptor id, payload descriptor, TTL, and hops from the Gnutella header, as well as Ip address from the query hit descriptor. We modified the payloadDescriptor header to include the following types: PING, PONG, QUERY, QUERY_HIT, QUERY_RW, QUERY_NF. These types correspond with an integer and are used to identify how the internal logic of each node will process the packet. We included a senderIP to keep track of the initial packet which sent the packet (used primarily for the query hit functionality). The destIp field was added to simplify the routing logic, so packets can be forwarded with more ease. Path.size and path were created for query hit functionality as well as to avoid cycles. By keeping a vector of each address kept we could see the packets last hop. This is used in the logic to check before forwarding, as well as tracing back for the reverse forwarding on a hit. Finally because of our design choices, a sinkNode field was added which contains the node ID which 'contains the wanted file'. We added this to avoid unnecessary checking and data transfer due to the scope issues outlined above.

By designing our packets this way, we were able to control routing with our own logic and better keep track of the metrics needed for later evaluation. Although it is not fully Gnutella-like, this modified packet was based on that framework and works in a similar fashion.

4.2 Application

The application is the core of our simulation. It encapsulates all logic for packet handling, search, and inter-node communication within our peer to peer network. At a high level it performs the following tasks.

1. Starting the simulation.
2. Initializing Sockets.
3. Launching Search Algorithms.
4. Sending and Forwarding Packets.
5. Recording Statistics.

For this section of the report, our group has chosen to give high level descriptions of how the code functions. At the top of the report we provided a link to the github for further implementation details.

4.2.1 Start Application - Neighbours

This function is invoked at the beginning of the simulation where it performs several key initialization steps. It begins by initializing each node and retrieving its IPv4 protocol stack. For every valid IP address assigned to a node, a corresponding UDP socket is created, bound to a unique IP and port pair. A receive callback is then registered, enabling the application to dynamically handle incoming packets. Each socket is also stored for later use when sending messages. This setup ensures that every node can communicate across all its interfaces and is fully prepared to participate in the simulation from the outset.

4.2.2 Send Packet

Packet transmission in our application follows a structure that is consistent across each of the search algorithms which we implemented.

Stages in packet transmission

1. Initialize<AlgorithmName>()
2. SendPacketFromSRC( )
3. <AlgorithmName>ExceptSender( )

Each of our algorithms are initialized from a unique entry point function such as InitializeFlood( ), IntializeNormalizedFlood( ), …. These functions are used to initialize the search process by selecting all or a subset of neighbors which are connected to the source node and calling SendPacketFromsrc( ) to begin the search process.

SendPacketFromsrc( ) is a core helper function used by all search algorithms to begin the communication process. This function performs, creating a peer to peer packet object, assigning the important metadata: (Message type, sender's IP address, TTL, Sink Node ID) Afterwards this is then encapsulates in a NS3 packet object and is sent via the appropriate socket utilizing NS-3's built in SendTo( ) function. From a high level, this function acts as a standardized mechanism to send packets from the source node to selected neighbors depending on the search algorithm.

Once an intermediate node has received a packet (covered in the following section in detail) the forwarding is handled by algorithm specific functions. Examples of this being FloodExceptSender( ), NormalizedFloodExceptSender( ) …, these determine how to propagate the packet forward in the network.

As an example, if there was a flood scheduled to be performed on a network, the simulation would be initialized and set up, followed by a called to IntialFlood( ) which would work through this specific algorithms logic, calling SendPacketFromSrc( ) to begin the searching process. Following this at each intermediate node, FloodExceptSender( ) is being called as packets are received to continue the flood propagation outwards.

4.2.3 Receive Packet

On the other side of packet transmission is packet retrieval. The ReceivePacket(Ptr<Socket> socket) function is another core component to the custom application logic. This function encapsulates the logic for interpreting and reacting to incoming messages within the network, and is responsible for routing and validation of packets.

After receiving a packet the node undergoes a series of checks to figure out how to handle the packet. Initially, the node will check if it is disabled, which terminates any transmission. Following this, to receive the incoming packet from the socket the node extracts our custom P2P packet header and begins to determine what will be done with this packet. Using the metadata that is stored within the packet if there is a loop detected via checking the path that the packet has already taken it will be dropped due to our simulations being acyclic, checks if the packet has returned to the original sending node, a successful search is logged where the system marks the query as complete. However, if this is not the case and there exists still neighboring nodes to forward the packet to the packet is forwarded depending on the metadata message type. Lastly, this function also can be thought of as a centralized control method for deciding what search is being performed. Depending on what the type variable is a specific function will be called to continue propagation of the packet.

Table 2. Message type for query propagation

Type	Associated Algorithm
QUERY	Flood
QUERY_RW	Random Walk
QUERY_NF	Normalized Flood

4.3 Network Sim

This file can be thought of as the driver of our simulation. While functions such as startApplication() begins the simulation and begins the packet transmission process, network-sim.cc is the true entry point for our project. Within here is where all pre-staging occurs.

1. Command line argument handling .
2. Network Simulator 3 initialization.
3. Network creation.
4. Node Creation.
5. Adding P2P Application to each created node.
6. Setting simulation start and stop time.
7. Peer addresses are set.
8. Events are scheduled and executed.

Additionally, this also encapsulates the functionality of network building as it is a stage which needs to be set up prior to the simulation being beginning.

4.4 Network Builder

After Network-sim.cc interprets how to handle command line arguments, the network builder determines which style of topology will be created. (The topologies are covered in further detail in subsequent sections) One of LINEAR, TREE, or REGULAR, will be selected and built using a corresponding function. Otherwise an error will be returned.

4.4.1 Network Topology Creation

While the specifics of how each topology is implemented will be covered in detail in the following section, there is a series of steps that occurs prior to the arrangement of connections which are important to understand. NS-3 handles many built in functions to streamline the creation of networks but does not handle everything for its user. As an example, nodes are able to be created using a function such as

```
NodeContainer Nodes;

Nodes.Create(numberOfNodes);
```

but, leaves it to the user to initialize all information regarding the specific nodes. In the following section we will cover information about how the nodes are set up as the same process is accomplished for each of the topology types.

4.4.2 Node Setup / Initialization

After the nodes have been created, getting each prepared follows a sequential process involving several of NS-3's built in functionalities which will be covered. We will start with explaining what each function is and why it is important to the overall setup.

4.4.3 InternetStackHelper

Serves the purpose of installing an internet protocol stack onto a container containing nodes. This adds all necessary protocol layers to a node including the IP layer, UDP/TCP transport protocol, and routing table management. Without this nodes within the simulation would be able to communicate.

4.4.4 Network Interfaces

A NetDevice in NS3 represents a network interface card that acts as a point of communication to receive and send packets for a node. Each node will have one or more of these depending on the number of neighbors that it is maintaining connections with.

4.4.5 Ipv4AddressHelper
Functions as a simple IPv4 address generator. This acts like a local number incremented.

4.4.6 PointToPointHelper
Creates connections between two nodes. Simply this takes in two nodes which are already set up and initialized and creates a point to point link between them.

4.4.7 Connections
NS-3 allows users to orient connection creation between two nodes assisted our group to automate the process of creating different types of topologies. After the initial setup and creation of our simulation we transitioned some effort into creating external python files that make use of the NetworkX library for the creation of graph types.

4.5 Graph Representation
While it is easy to picture a graph in your mind after referring to a picture, how graphs are implemented and stored within our code requires some previous understanding.

A graph from a mathematical point of view can be represented as V = {v1, v2, v3 , … , vn} a set of vertices and E = { (u,v) | u,v element of V, (u != v) } a set of edges that are unordered pairs of vertices that represent connections between two vertices.

For the purpose of our project we needed to decide how to store the network itself and from doing research online we had decided to make the choice of using an adjacency matrix to maintain our node connections.

4.5.1 Adjacency List
An adjacency list can be thought of as a way to represent a graph using a series of unordered lists that represent connections between vertices. Suppose we have the graph represented in figure 13. In this graph we have 5 vertices each of which have a varied number of neighbors. This graph can be represented using a 2D vector.

```
std::vector<std::vector<char>> nodeNeighbours(5);
```

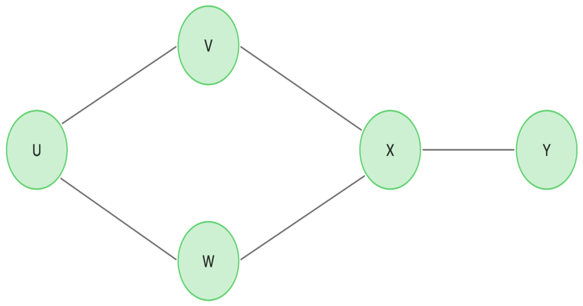


Figure 13. Visual representation of a simple graph

Each node has a corresponding row within the 2D vector that maintains each connection within the graph. So,

Table 3. Tabular representation of a 5 vertex graph

Vertex ID	Connected To
U	U,W
V	U,X
W	U,W
X	V,W
Y	X

From this you are able to build a graph from scratch, or, traverse a graph by taking a vertex and its known neighbors to create a path to traverse along. Using this approach it allows us to have a central location that contains all information about our network.

Using an adjacency list in mind, we translated this general format to store IP addresses of the NetDevices being used when assigning links between two nodes.

```
// Vector to store node neighbors
    std::vector<std::vector<Ipv4Address>> nodeNeighbors(numNodes);
```

Figure 14. RegularGraph storage container

Understanding the logic behind this is fundamental to how our application code understands how to send and receive packets. As an example suppose we are attempting to initiate a flood from node U in our previously mentioned graph. Under the assumption that the nodes would be stored in lexicographical order each node can be accessed by using the index from which it is placed in the container.

Table 4. Nodes in comparison to the vector position

Node	2D Vector Position
U	0
V	1
…	n+1

If we were attempting to find which nodes to forward the packet to from node U we would index within the container at position 0 to retrieve a list containing all relevant connections to that node. Furthermore, if we were to create / remove a node from this network the adjacency list would need to be updated correspondingly to ensure consistency. Otherwise dangling nodes would lead to errors with packet logic.

This logic is also pivotal for the automation of our graphs being created. It allows us to take as input a text file that has graph connections in a similar format. (discussed further in detail in a subsequent section)

4.6 Topologies

When working through this project, choosing what topology type to implement was an important task. We aimed to choose topologies that covered a wide variety of connections while allowing us to learn different methods to create them. With this in mind we chose to start out with implementing a simple line or linear graph consisting of 5 nodes. This allowed us to test the initial implementation of our graphs and ensure that our application logic was functioning as intended.

4.6.1 Positioning & Visualization

While network simulator 3 does have a built in visualization tool for their simulations, it chooses to place vertices arbitrarily without consideration of how the structure of the network will look. (As it would be in real life). So, as the number of nodes grows in a graph the space between vertices becomes smaller. However, Binary trees can be visualized simply by using similar logic to construction just with translating it to be positioned within a 2D plane but for graphs such as our megagraph, it becomes convoluted and confusing.

4.6.2 Graph Builder

After building some topology types, our group noticed that the manual nature of building out a graph can take out a significant portion of time. For this reason we search for external tools to assist in graph creation. This led us to doing research and finding the python library NetworkX and creating an automation file graph-builder.py to create a better solution to a problem.

4.6.3 NetworkX

A python package that is designed for the creation, manipulation, and study of the structure, dynamics, and functions of dynamic networks[21].

4.6.4 Tree Graph

The tree graph was the first and most thoroughly planned topology implemented in our project. By definition, a tree is a connected, acyclic graph where a unique path exists between any two nodes. For our implementation, we focused specifically on binary trees, a structured subset of trees where each node can have at most two children: a left and a right child.

Our decision to use binary trees was primarily influenced by the procedural simplicity they offer. Specifically, binary trees can be efficiently represented and constructed using a one-dimensional array, enabling straightforward indexing and connection logic between parent and child nodes.

4.6.5 Binary Tree Construction

Suppose you have a vector consisting of k nodes. To build a binary tree from this vector, iterate through each position i. For each node at position i assign its left child to index 2*i + 1 and its right child is at position 2*i + 2.

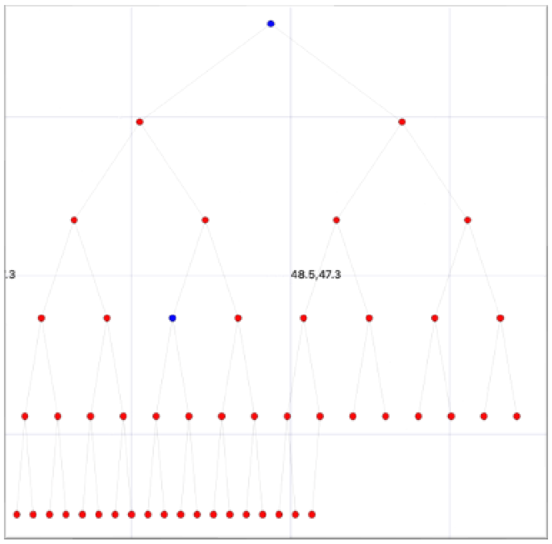


Figure 15. Binary Tree visualization from netanim.

4.6.6 K- Regular Graph

A k regular graph is a simple graph with the property of each vertex having the same degree. More formally, a simple graph is said to be k regular if all vertices within the graph have the same degree r [22]. These types of graphs are built into the NetworkX Python library, allowing us to simply state what type of degree that graphs nodes will be and the probability that two nodes would be connected.

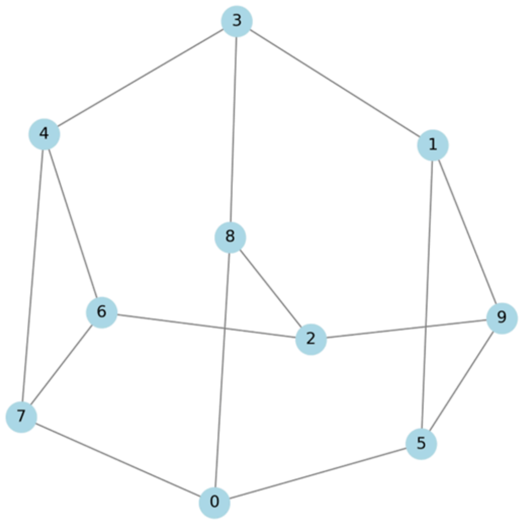


Figure 16. Regular graph

4.6.7 Cluster Graph

Moving forward from regular graphs the following topology which we chose to implement was the cluster graph. This type of graph can be thought of as several regular graphs interconnected to a central hub node.

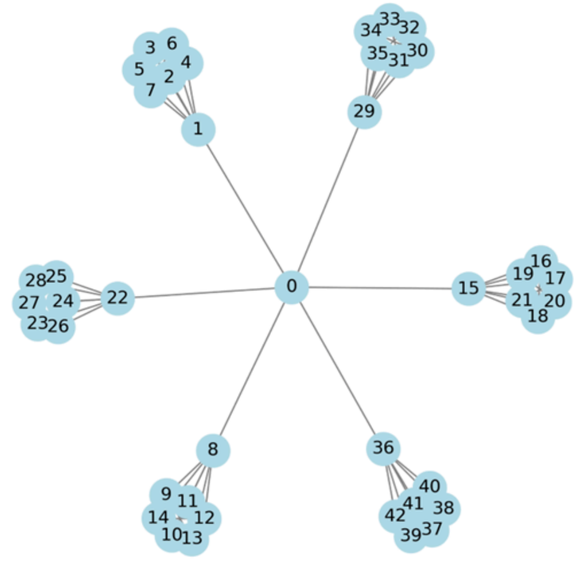


Figure 17. Cluster Graph

4.6.8 Megagraph

Finally the last graph type which we chose to implement and test was a custom megagraph. This topology can be thought of as several cluster graphs attached together.

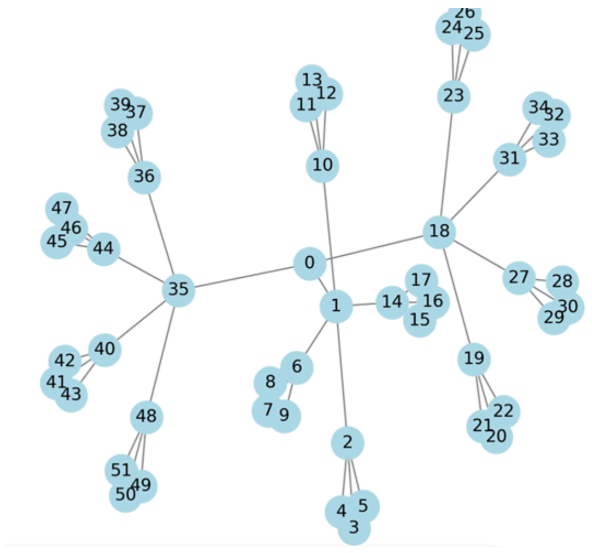


Figure 18. MegaGraph

4.7 Search Methods

The main focus of this project was to explore the different search methods in unstructured decentralized P2P networks. We chose to look at flooding, normalized flooding and random walkers to find potential differences in the above outlined topologies. In this section we will discuss the algorithms and how we implemented them in our custom logic. We will provide details on how they were modified to fit our code and how the routing logic works.

As mentioned in the packet structure section, we added a sinkNode field into the packet header. For each type of query, there is a node that is chosen to be the sink. At each hop, there is a check in place seeing if it has reached the destination. This method added simplicity to the simulation and avoided any ambiguity of results or testing.

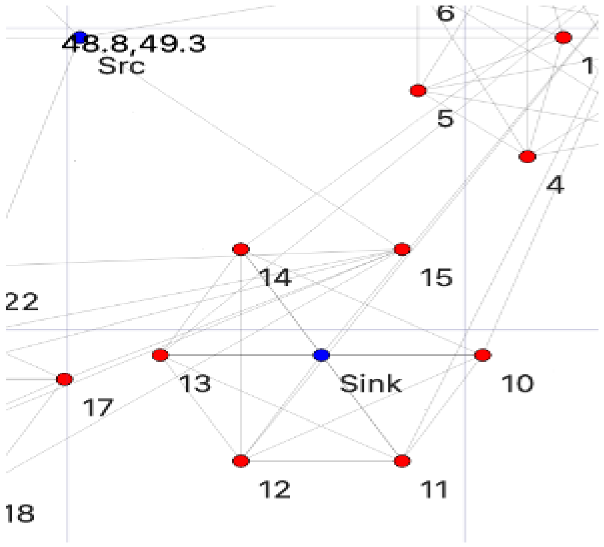


Figure 19. Src and sink node shown in a simulation run

Using this method to confirm a query hit, we utilized the path field of the packet header to perform reverse forwarding like in Gnutella. Although Gnutella uses its neighbours table to do this, we opted for the path vector for simplicity due to the time constraints. Table _ outlines the query hit process each packet goes through after finding the sink node.

Table 5. Query Hit Process

1	Packet is sent to a ForwardQueryHit function
2	payloadDescriptor field is changed to QUERY_HIT
3	Destination field is set to the last hop (top of the path vector), last hop is removed
4	Updated packet header is attached to a new packet object and sent to the last hop destination
5	In the RecievePacket function, the packet is checked ● If it has reached the original sender, stats are updated ● Not at the original sender, back to ForwardQueryHit

This method ensures that the originator gets the response without the need for a centralized routing table. Each of the methods below proceed based on their separate logic until a query hit is found. Once this happens they switch to the universal query hit logic. Below outlines the algorithms we implemented and their differences. In our code, we have an initial and except sender version of each algorithm. This allows for the forwarding logic to be separate from the initial. Further we use a separate function SendPacketFromSource to handle the initial query. By making this distinction the code gains readability and we were able to scale effectively when adding the different methods. Below will cover the forwarding logic in more depth.

4.7.1 Flooding

The flooding method was the first that we implemented and is true to early Gnutella protocol. In this method, the packet is forwarded to all the nodes neighbours, ignoring the sender. In our version upon each hop, the TTL is decreased, hop is increased and the path vector is updated with the previous hop. If there is no query hit (the sink node has not been reached) the packet continues in the flooding logic and is forwarded to more neighbours.

```
void
P2PApplication::FloodExceptSender(P2PPacket p2pPacket, int
excludeIndex)
{
   for (uint32_t i = 0; i < m_ipv4Addresses.size(); ++i)
   {
       // skip the ipv4 the request came from!
       if (i == excludeIndex)
       {
           continue;
       }

       Ipv4Address curIPV4 = m_ipv4Addresses[i];
       Ipv4Address curNeighbor = m_neighbours[i];

       if (curNeighbor != p2pPacket.GetSenderIp())
       {
           // Create a copy of the packet before modifying
           P2PPacket packetCopy = p2pPacket;
           packetCopy.AddToPath(curIPV4);

           Ptr<Packet> newPacket = Create<Packet>();
           newPacket->AddHeader(packetCopy); // Attach the updated
copy

           NS_LOG_DEBUG("rp: forwarding packet from " << curIPV4 <<
" to " << curNeighbor);
           // packetCopy.PrintPath();
           NS_LOG_DEBUG("----------\n");

           m_sockets[i]->SendTo(newPacket, 0,
InetSocketAddress(curNeighbor, m_port));
       }
   }
}
```

Figure 20. FloodExceptSender function in application.cc

Once the initial flood is called at the source node, it will loop through and send a query packet to each of the neighbouring IPs using SendPacketFromSrc. Then those neighbours who receive the query will call FloodExceptSender (fig_). This function forwards the packet to all of the neighbours except the one which it was received from. In Fig_ we can see it loop through each of the addresses completing the same process of updating the packet header, attacking it to a packet object and forwarding to the neighbour socket.

4.7.2 Normalized Flood

Normalized flooding is very similar to flooding in regards to sending to the neighbours, however it has some key differences. The biggest one being that the amount of queries sent and forwarded is capped by the minimum degree of the network. Secondly, each subset of neighbours is chosen uniformly at random. By doing this the amount of packets in the network is lowered and duplicates are lessened. This method was tested to help balance reachability with efficiency. Fig _ outlines the code:

```
void
P2PApplication::NormalizedFloodExceptSender(P2PPacket p2pPacket,
int excludeIndex, int howManyNodes)
{
   // cap howManyNodes to the number of neighbours
   if (howManyNodes > m_neighbours.size())
   {
       howManyNodes = m_neighbours.size() - 1;
   }

   // seed for randomness
   std::random_device rd;
   std::mt19937 gen(rd());

   // Create vector of neighbour size, shuffle indices
   std::vector<int> indices(m_neighbours.size());
   std::iota(indices.begin(), indices.end(), 0);
   std::shuffle(indices.begin(), indices.end(), gen);

   int sentCount = 0;
   for (int i = 0; i < indices.size() && sentCount < howManyNodes;
++i)
   {
       int randomIndex = indices[i];
       // skip the ipv4 the request came from!
       if (randomIndex == excludeIndex)
       {
           continue;
       }

       Ipv4Address curIPV4 = m_ipv4Addresses[randomIndex];
       Ipv4Address curNeighbor = m_neighbours[randomIndex];

       if (curNeighbor != p2pPacket.GetSenderIp())
       {
           // Create a copy of the packet before modifying
           P2PPacket packetCopy = p2pPacket;
           packetCopy.AddToPath(curIPV4);

           Ptr<Packet> newPacket = Create<Packet>();
           newPacket->AddHeader(packetCopy); // Attach the updated
copy

           NS_LOG_DEBUG("NF: forwarding packet from " << curIPV4 <<
" to " << curNeighbor);
           // packetCopy.PrintPath();
           NS_LOG_DEBUG("----------\n");
           m_sockets[i]->SendTo(newPacket, 0,
InetSocketAddress(curNeighbor, m_port));
           sentCount++;
       }
   }
}
```

Figure 21. NormalizedFloodExceptSender function in application.cc

After the InitialNormalizedFlood function is called, the source node shuffles a list of its neighbours and sends the packet to a limited number of them. Then when a packet arrives at an intermediate node the NormalizedFloodExceptSender function repeats this behaviour. It again shuffles the list of neighbours, excludes the sender and forwards the query to a smaller subset of random neighbours. By utilizing the sentCount variable, we are able to send to the correct number of neighbours. Currently we have to hardcode the minimum edge amount, which in future work could be handled more dynamically.

4.7.3 Random Walkers

The random walkers algorithm was the third search algorithm we implemented and takes certain aspects of the normalized floodings randomness, however it sends k (defined upon call time) random walkers to search for the sink node. Each of the random walkers is independent of each other and their paths are decided uniformly at random. The previous steps do not influence the next hop. There are some variations of this algorithm that use heuristics to make more informed choices of the next hop, however due to the scope and time constraints we chose to implement this more simplified version. In this version it reduces message overhead compared to flooding, however it sacrifices the probability to find content. Fig _ is the random walk after the initial k walkers are sent.

```
void
P2PApplication::RandomWalkExceptSender(P2PPacket p2pPacket, int
excludeIndex)
{
   if (m_neighbours.size() <= 1)
   {
       NS_LOG_DEBUG("No other neighbors to forward the random walk
to.");
       return;
```

```
    }

    std::random_device rd;
    std::mt19937 gen(rd());

    std::vector<int> validIndices;
    for (uint32_t i = 0; i < m_neighbours.size(); ++i)
    {
        if (i != excludeIndex)
        {
            validIndices.push_back(i);
        }
    }

    if (validIndices.empty())
    {
        NS_LOG_DEBUG("No valid neighbor to send to (excluding
sender).");
        return;
    }

    std::uniform_int_distribution<> dist(0, validIndices.size() -
1);
    int chosenIndex = validIndices[dist(gen)];

    Ipv4Address curIP = m_ipv4Addresses[chosenIndex];
    Ipv4Address nextHop = m_neighbours[chosenIndex];

    p2pPacket.AddToPath(curIP);
    p2pPacket.DecrementTtl();

    Ptr<Packet> newPacket = Create<Packet>();
    newPacket->AddHeader(p2pPacket);

    NS_LOG_DEBUG("Random walk forwarding from " << curIP << " to "
<< nextHop);
    // p2pPacket.PrintPath();
    NS_LOG_DEBUG("----------\n");

    m_sockets[chosenIndex]->SendTo(newPacket, 0,
InetSocketAddress(nextHop, m_port));
}
```

Figure 22. RandomWalkExceptSender function in application.cc

After the source node initiates the k independent walks, each packet is sent to a neighbour. There is a chance that the packets will get sent to the same neighbour, and this increases as the number of neighbours decreases. At each intermediate node, the RandomWalkExceptSender function excludes the sender and forwards the packet to another random neighbour. Here we also decrement the TTL and add to the path vector. This strategy mimics real world random walk strategies while preventing loops or backtracking.

5. METRICS

5.1 Chosen Metrics

Table 6. Metrics Chosen

Metric	Description
Hops	Number of nodes it took to get a Query Hit. We collected average, median, min and max.
Latency	The time it took from source node to sink node. We collected average, median, min and max.
Initial Requests	How many requests it sent from the source node.
Retried Requests	How many retries it took for a source node to get a query hit. Capped at 5 max.
Query Hits	Number of total Query Hits it received. We collected average, median, and max.
Unique Query Hits	Number of Query Hits that came from a unique neighbour.
Query Success Rate	Unique Query Hits / Initialized Requests
Redundant Query Hits	Unique Query Hits - Query Hits
Average Total Work (Intermediate nodes)	Sum of Sent and Received Requests
Wasted Requests	Percentage of how many requests a node

(Intermediate nodes)	sent that did not lead to a Query Hit
Query Hit Efficiency (Intermediate nodes)	Percentage of how many requests a node received that lead to a Query Hit
Zero Work Done (Intermediate nodes)	A node that performed no work at all. Does not include disabled nodes.

5.2 Simulations Ran

During our presentation we opted to choose a wide variety of graph types and test them all. Shortly after creating and running these simulations, we realized that it was hard to analyze and perceive the information in a meaningful way due to several factors. The first was that the way that we oriented our graphs and designed them are very symmetrical. So, as you would scale up the number of nodes and run the same algorithm across them, for an algorithm such as flooding, would result in relatively the same results just on a largers scale. We also realized that some of the metrics which we chose to initially track would not be able to compress into graphs and tables in a meaningful fashion.

Following the presentation and with our newly learned knowledge, we chose to take some time to group and consolidate the data so that we could organize it based on two main categories for analysis.

1. Graph Type: This category represents all the different graph topologies generalized into simple categories;
    a. Cluster
    b. K-Regular, where k is some integer
    c. Tree with some number of nodes
    d. Megagraph

We chose to separate our data into these categories for simplicity and consistency. It allowed us to select and compare different features within graph topology types.

2. Search Type: This category represents each of the different search algorithms which we implemented and ran on the graphs
    a. Flood
    b. Random Walk
    c. Normalized Flood

After selecting these main categories to base our analysis around, using the implementation path that we chose to follow along with the python files with NetworkX we ran the following simulations with the number of nodes in each graph type scaling.

For each of the following graph types we ran each algorithm across 20 different simulations

Table 7. Simulations Ran

Graph Type	Simulations Ran Per Algorithm	Total Simulations
3 Regular 10 Nodes 3 Regular 20 Nodes 3 Regular 30 Nodes 3 Regular 40 Nodes 3 Regular 50 Nodes	20 Flood 20 Normalized Flood 20 Random Walks	300
4 Regular 200 Nodes	20 Flood 20 Normalized Flood 20 Random Walks	60
5 Regular 200 Nodes	20 Flood 20 Normalized Flood 20 Random Walks	60
Mega Graph 3 Clusters Mega Graph 5 Clusters	20 Flood 20 Normalized Flood 20 Random Walks	120
6 Clusters 4 Nodes Each 6 Clusters 5 Nodes Each	20 Flood 20 Normalized Flood	300

6 Clusters 10 Nodes Each 6 Clusters 20 Nodes Each 6 Clusters 30 Nodes Each	20 Random Walks	
Tree with 10 Nodes Tree with 20 Nodes Tree with 30 Nodes Tree with 40 Nodes Tree with 50 Nodes Tree with 60 Nodes	20 Flood 20 Normalized Flood 20 Random Walks	360

After these final simulations gathering the metrics which are covered in the metrics section, we then aggregated the data into a master file which contained all information, allowing for easy filtering and matching of data categories. Additionally, we performed statistical analysis for each algorithm separately to capture individual statistics. In totality we have

Numerical Variables

1. Node ID
2. Unique Query Hits
3. Received Queries
4. Sent Queries
5. Query Hits
6. Forwarded Query Hits
7. Tried Requests
8. Initialized Requests
9. Node Count

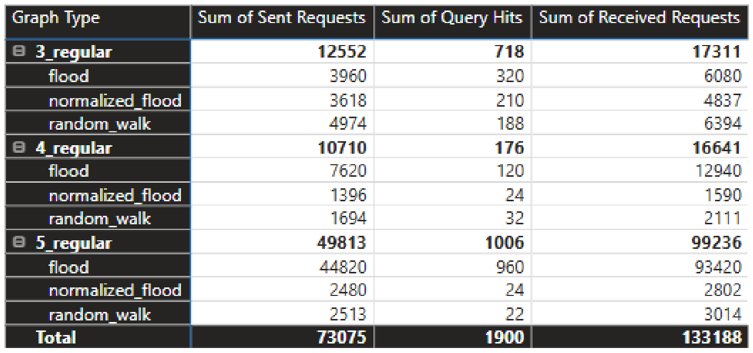
Categorical Variables

1. Search_Type
    a. Flood
    b. Normalized Flood
    c. Random_walk
2. Graph_Type
    a. k - Regular with j nodes each
    b. Tree with j nodes
    c. k Clusters j nodes each
    d. Megagraph with k clusters

5.3 Results

As a result of our simulations, we ended up having 1,200 total simulations run, and, after the aggregation of data into a single file resulted in 94,921 rows of data massing to 1,044,131 individual data points.

For the visualization of our results we chose to have two sources of information, the first would be pure calculations from the data collected from each of the simulations. The second is the aggregated and categorized data which would be used as input into Microsoft's Power BI to enhance our visuals and understanding of the results.



Graph Type	Sum of Sent Requests	Sum of Query Hits	Sum of Received Requests
⊟ **3_regular**	**12552**	**718**	**17311**
flood	3960	320	6080
normalized_flood	3618	210	4837
random_walk	4974	188	6394
⊟ **4_regular**	**10710**	**176**	**16641**
flood	7620	120	12940
normalized_flood	1396	24	1590
random_walk	1694	32	2111
⊟ **5_regular**	**49813**	**1006**	**99236**
flood	44820	960	93420
normalized_flood	2480	24	2802
random_walk	2513	22	3014
Total	**73075**	**1900**	**133188**

Figure 23. Summation and categorization of Sent Requests, Query Hits, and Received Requests for each Regular graph type ran in simulations

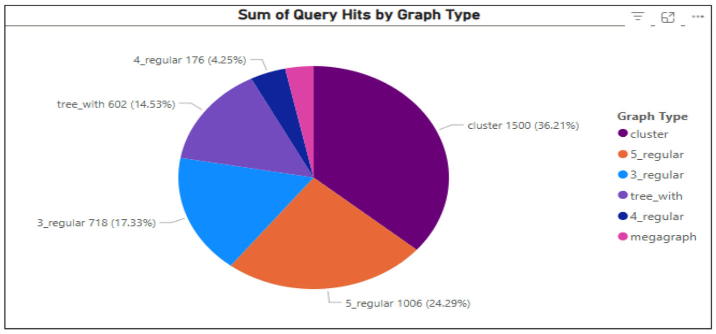


Figure 24. X Sum of Query Hits by Graph type

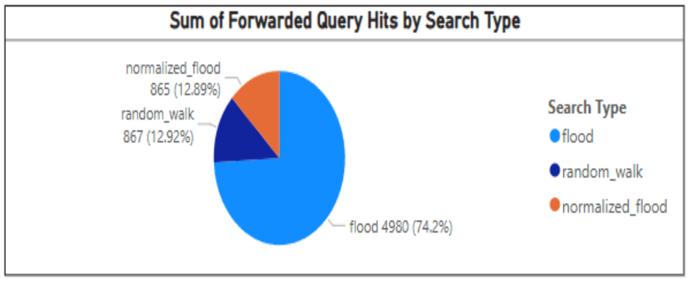


Figure 25. Sum of Forwarded Query Hits by Search Type

6. DISCUSSION

6.1 Topology Level Insights

After running the proposed simulations, we gathered several key insights that lead to the following insights. For clarity, we refer to Flood as F, Normalized Flood as NF, and Random Walk as RW throughout this section. A complete set of raw results is presented in the Appendix for reference (CITE?). As mentioned before, among the most important performance we analyzed are:

- **Redundant Query Hits**: the percentage of extra query hits received from the same neighbours, indicated inefficiency.
- **Total Work Done**: the cumulative effort performed by intermediate nodes, including sending and receiving requests, and forwarding query hits.
- **Wasted Requests**: the proportion of requests processed and forwarded by intermediate nodes that did not result in an successful query hit, highlighting unnecessary communication overhead
- **Efficiency**: the proportion of requests processed that resulted in query hits.

6.1.1 Binary Tree

Flood performs exceptionally well in small networks, achieving 100% success with a total message count comparable to that of other algorithms. In contrast, Normalized Flood and Random Walk) proved unreliable even in the smallest tested network of 10 nodes, failing to reach full success despite generating similar amounts of work at intermediate nodes and requiring multiple retries. As the number of nodes increased, RW produced inconsistent results, with success rates fluctuating (e.g., dropping from 65% to 35% and back), highlighting the unpredictability of its random path selection. Contrary to expectations, RW did not offer significant overhead savings: its total work grew almost proportionally to Flood, yet without the reliability. In larger trees (e.g., 70–80 nodes), RW's success rate fell to around 27%, accompanied by a high percentage of wasted requests. NF followed a different pattern - initially performing well in smaller topologies, but its success rate declined steeply as the network scaled, reaching as low as 10%. While NF had about half the internal message load (overhead in intermediate nodes) compared to Flood and RW, the dramatic drop in success rate made it completely useless. A key insight from both RW and NF is that, on average, more than 50% of intermediate nodes were left idle—the search algorithm simply never reached them. Although this reduced overhead, it also meant that even with multiple retries, these algorithms frequently failed to discover any results. We will see this exact same problem in the other topologies. Disabling 20 % of intermediate nodes did not change much. RW and NF performed slightly better, but still failed with a high number of nodes. In conclusion, Flood emerges as the clear winner for reliable search, while Normalized Flood was useless, and Random Walk offers inconsistent success with overhead nearly as high as Flood, making it an unreliable alternative.

6.1.2 Regular Graph

We evaluated performances of three search algorithms on 3-regular graphs ranging from 10 to 50 nodes to observe how they scale in moderately connected networks. Additionally, we used larger, highly connected graphs with 200 nodes, using 4-regular and 5-regular topologies.

6.1.2.1 3-Regular

In 3-regular graphs, Flood consistently achieved 100% success across all network sizes, making it the most reliable algorithm. It scaled predictably in terms of latency and total work, but at the cost of increasing redundant communication and wasted requests in larger networks. At 50 nodes, Flood generated 33% redundant query hits, indicating an increase in unnecessary duplicate responses as the network grew.

Normalized Flood performed well in small graphs, but as the number of nodes increased, it suffered from rising latency, declining success rates (down to 80%), and poor node utilization, with many intermediate nodes remaining idle. Random Walk showed inconsistent performance, with success rates fluctuating and total work approaching that of Flood despite being designed as a lightweight alternative. While RW had slightly better latency than NF at scale, its overall efficiency and reliability remained low. Across all algorithms, the average percentage of wasted requests hovered around 50%, yet the query hit efficiency declined steadily. Although NF was the most resource-efficient, its latency and average number of hops became disproportionately high, suggesting it often failed to explore the right paths to reach target nodes efficiently.

When 20% of intermediate nodes were disabled, Flood showed strong resilience and full network utilization. NF and RW average success rate increased in small graphs but with 40 and 50 nodes they experienced steep drop in success rate - as low as 25 % and 30 % respectively despite similar or lower total work done, and still a high number of idle nodes. These results suggest that while Flood is resource-intensive, it remains the only option that scales reliably in 3-regular topologies, whereas NF and RW offer little benefit and degrade significantly as network size increases.

6.1.2.2 4-Regular and 5-Regular with 200 nodes

In highly connected networks with 200 nodes, the performance gap between Flood, Normalized Flood and Random Walk becomes very noticeable. In the 4 regular graphs, Flood had 33 % redundant query hits and 58 % wasted requests, with a total of 1000 work done in intermediate nodes. In contrast, NF and RW achieved 55% and 75% success rates respectively, while their overhead was 5 times less than Flood. The success rate can be explained with about 138 nodes idle, meaning that 70% of intermediate nodes did not work. However, the success rate remains quite high despite lower overhead and the high number of idle nodes. In the 5-regular graph, the disparity grows further. Nodes' total work exploded to 7000 messages in Flood, accompanied by 79% redundant query hits and 45% wasted requests. Meanwhile, NF and RW drop to 50% and 40% success rates, with over 100 nodes idle in each (50%). Even though NF and RW generate far fewer messages, they fail to capitalize on the connectivity, leading to significant loss in reachability and effectiveness. Overall, Flood is the only algorithm that consistently achieves reliable discovery, but its high overhead makes it poorly suited for scalability in large, dense networks. Normalized Flood struggles significantly, with low success rates and poor utilization of the network. Random Walk performs moderately well—succeeding roughly half the time—and while it lacks consistency, it is sometimes preferred due to its lower communication overhead compared to Flood.

When 20% of intermediate nodes were disabled, Flood remained fully reliable with almost triple less total overhead in intermediate nodes. However, in the 5-regular graph, there was still a 76% redundant query hits and high overhead (1800 total messages). Normalized Flood collapsed, with success rates dropping to just 5% - 15% and over 60% of nodes idle. Random Walk performed slightly better with 50% success in 4-regular graphs, but had only 10% in 5-regular ones. This shows that both NF and RW struggle severely in the presence of node failures.

6.1.3 Cluster

In clustered topologies with increasing cluster size, Flood consistently achieved 100% success rate, but its communication cost scaled aggressively, with total work growing from 70 to 2800 total messages, and redundant query hits increasing from 50% to 93%. Normalized

Flood performed well in small clusters, maintaining 10% success at 4 nodes per cluster, but its success rate dropped to just 40% as cluster size grew. Random Walk showed similar behaviour, with success failing from 95% to 50%, increasing average latency and number of hops, and over 60% idle nodes. Once again, the problem with idle nodes hit NF and RW which resulted in lower success rate. Overall, while Flood remains most reliable, it fails to scale well with high overhead. NF and RW performed similarly with good results than other graphs.

With 20% of intermediate nodes disabled, Flood performs well but still has similar overhead and high redundant query hits. In contrast, RW and NF collapsed under failure with success rates dropping to 20% and 10%. While NF and RW maintained low message counts, their inability to reach across clusters under failure made them ineffective for reliable search.

6.1.4 Megagraph

Megagraph is a topology consisting of multiple loosely connected clusters. Flood effectively discovered targets across clusters without generating any redundant query hits or idle nodes. However, its total work increased 4 times, from 69 messages in 3 clusters to 315 messages in 5 clusters. NF and RW had a success rate of 55% and 65% respectively with 3 clusters and similar overhead compared to Flood. However, with 5 clusters NF and RW failed to scale and their success rate was about 15%. Despite their lower communication overhead, NF and RW failed to explore across clusters, making them ineffective for reliable query resolution in such topologies.

6.2 Strengths

One of the strengths this project provides is the custom built simulation developed in ns-3. This allowed for detailed control over node behaviour, packet forwarding logic and topology generation which gave a flexible and extensible platform to test P2P network behaviours. Further, the project evaluated multiple search algorithms including flooding, normalized flooding and random walks (across various topologies). Because of this we were able to gain valuable performance metrics including query success rate, latency, hop count and redundant work. This enabled a well rounded comparison and gave us insight into each method's trade off.

6.3 Weaknesses

While this project hit many of its objectives, it had challenges. One major limitation was the broad scope of the project, which was narrowed during the time frame we were building. As we implemented and researched, we learned that our initial goals were not feasible within the given time frame and thus were moved to future work (see section below).

Our goal of investigating search algorithms seemed concise, however to do this we needed to cover many more topics and were unfamiliar with ns-3 initially. There is a certain documentation, however we found a lack of specific external resources to help guide us on this project. This increased the learning curve and development time which made us shorten certain goals. Further, our group had many implementation details to discuss, which while educational introduced uncertainty and inconsistency at times. The project also produced a large number of output files and results, which made the organization and analysis more difficult. We had to create automated tools to parse and separate data, as well as make decisions on the importance of certain files over others. Despite these hurdles, our group was able to produce a functional simulation which gave results and insights.

6.4 Moving Forward

Moving forward we hope to implement functionality to better reflect real world unstructured decentralized P2P systems. This would include incorporating different topologies that are less symmetrical or small-world networks to align more closely with real networks. It would be interesting as future work to introduce real file indexing and sharing mechanism to simulate the full lifecycle of a P2P request, which could potentially add more insight into bandwidth consumption and content availability. Finally we could introduce smarter algorithms for query routing such as GIA or heuristic based walks to further investigate efficiency and redundancy. There are many different ways this project could be furthered, and we hope to continue this work to provide more insights on the comparisons in unstructured decentralized P2P networks.

7. CONCLUSION

7.1 Summary

We designed and simulated a decentralized P2P network inspired by the Gnutella protocol, focusing on search queries. Using the ns-3, we implemented and compared three search algorithms: flooding, normalized flooding, and random k-walks. Each of these techniques was evaluated in terms of how efficiently they locate a sink node while tracking important metrics like query hits, hop counts, and packet overhead.

We developed a custom graph generation tool in Python to produce varied network topologies including regular graphs, clustered structures, and mega-graphs with multiple subclusters connected via a central node. These were imported into ns-3 using GraphML or plain text, then visualized with NetAnim using a custom layout function. Our packet and application code simulated query propagation, including TTL control and path tracking.

We learned how P2P concepts like query flooding can be optimized with strategies that reduce message duplication and bandwidth usage. We also gained experience building real-time network applications in ns-3, managing simulation complexity, and measuring performance through statistical logging. While our system is simplified and doesn't include file transfers, it serves to explore the trade-offs in decentralized search which is a foundational component of real-world systems

7.2 Contributions

CONTRIBUTOR NAME	CONTRIBUTIONS
Holly Gummerson V00986098 Computer Science University of Victoria Victoria BC Canada hgummerson@uvic.ca	● Proposal, Biweekly update 1, Biweekly update 2, Midterm Update, Final Presentation, Final project ● Rewriting and editing documents ● Final Assessment of documents ● Updating Website with new documents ● Search algorithm research ● Search algorithm implementation ● Receive packet ● Queryhit forward ● Paper (1-10), (search algos) ● Presentation (speaking slides) ● Cluster graph generator ● Megagraph generator ● Positioner for those two
Ali Gaineshev V000979349 Computer Science University of Victoria Victoria BC Canada ggaineshev@gmail.com	● Proposal, Biweekly update 1, Biweekly update 2, Midterm Update, Final Presentation, Final project ● Updating Website with new documents ● Rewriting and editing documents ● Final Assessment of documents ● Updating Website with new documents ● Topology Implementation ● Netanim visualizations for Trees ● Code Refactoring ● Caching ● Code structure + commenting ● Statistics Gathering with Code ● Running Simulations

Mathew Terhune V00943466 Computer Science University of Victoria Victoria BC Canada mterhune@uvic.ca	● Proposal, Biweekly update 1, Biweekly update 2, Midterm Update, Final Presentation, Final project ● Rewriting and editing documents ● Final Assessment of documents ● Topic Research ● Search algorithm research ● Final Presentation Speaking ● Final Presentation Content and slide designing ● Graph generation Code ● NetworkX research ● Query Retry initialization and setup ● Node leaving / joining research and attempted implementation ● Topology Research ● Metric aggregation and attribute creation ● Metric displaying through Power BI

REFERENCES

[1] J. Wu and X. Li, “Searching Techniques in Peer-to-Peer Networks,” in Handbook on Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless, and Peer-to-Peer Networks, 2005. doi: https://doi.org/10.1201/9780203323687.ch37.

[2] G. Manes et al., “Search Algorithms for Unstructured Peer-to-Peer Networks,” CiteSeer X (The Pennsylvania State University), pp. 343–352, Oct. 2007, doi: https://doi.org/10.1109/lcn.2007.65.

[3] nsnam, “ns-3,” ns-3, 2019. https://www.nsnam.org/

[4] “ENSC 427 Communication Networks Implementation of the Gnutella Protocol Spring 2010 FINAL PROJECT.” Accessed: Apr. 16, 2025. [Online]. Available: https://www.sfu.ca/~ljilja/ENSC427/Spring10/Projects/team7/final_report_Gnutella.pdf

[5] “Peer-to-Peer Networks: Gnutella” Accessed: Apr. 16, 2025. [Online]. Available: https://www.sfu.ca/~ljilja/ENSC427/Spring10/Projects/team6/Ensc427_final_presentation.pdf

[6] “Gnutella - Stable - 0.4,” Sourceforge.net, 2025. https://rfc-gnutella.sourceforge.net/developer/stable/

[7] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker, “Making Gnutella-like P2P Systems Scalable.” Accessed: Apr. 16, 2025. [Online]. Available: https://courses.cs.washington.edu/courses/cse522/05au/GIA.pdf

[8] V. Kanade, “Peer-To-Peer Networks: Features, Pros, and Cons,” Spiceworks, Nov. 07, 2023. https://www.spiceworks.com/tech/networking/articles/what-is-peer-to-peer/.

[9] “Gnutella Protocol - an overview | ScienceDirect Topics,” www.sciencedirect.com. https://www.sciencedirect.com/topics/computer-science/gnutella-protocol

[10]J. Bo, “Flooding-Based Resource Locating in Peer-to-Peer Networks,” Lecture Notes in Electrical Engineering, pp. 671–678, 2011, doi: https://doi.org/10.1007/978-3-642-21697-8_85.

[11] J. Wu and X. Li, “Searching Techniques in Peer-to-Peer Networks,” in Handbook on Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless, and Peer-to-Peer Networks, 2005. doi: https://doi.org/10.1201/9780203323687.ch37.

[12] D. R. Kumari, H. Guclu, and M. Yuksel, “Ad-hoc limited scale-free models for unstructured peer-to-peer networks,” Peer-to-Peer Networking and Applications, vol. 4, no. 2, pp. 92–105, May 2010, doi: https://doi.org/10.1007/s12083-010-0067-1.

[13] S. M. THAMPI, “SURVEY OF SEARCH AND REPLICATION SCHEMES IN UNSTRUCTURED P2P NETWORKS,” Network Protocols and Algorithms, vol. 2, no. 1, May 2010, doi: https://doi.org/10.5296/npa.v2i1.263.

[14] C. Gkantsidis, M. Mihail, and A. Saberi, “Random walks in peer-to-peer networks,” IEEE INFOCOM 2004, vol. 1, pp. 120–130. doi:10.1109/infcom.2004.1354487

[15] S. M. THAMPI, “SURVEY OF SEARCH AND REPLICATION SCHEMES IN UNSTRUCTURED P2P NETWORKS,” Network Protocols and Algorithms, vol. 2, no. 1, May 2010, doi: https://doi.org/10.5296/npa.v2i1.263.

[16] Bordignon, Fernando & Tolosa, Gabriel. (2001). Gnutella: Distributed System for Information Storage and Searching Model Description.

[17] Barjini, H., Othman, M., Ibrahim, H. et al. Shortcoming, problems and analytical comparison for flooding-based search techniques in unstructured P2P networks. Peer-to-Peer Netw. Appl. 5, 1–13 (2012). https://doi.org/10.1007/s12083-011-0101-y

[18] IBM, “Network topology,” Ibm.com, Oct. 24, 2024. https://www.ibm.com/think/topics/network-topology

[19] “Roles of NetAnim in Ns3| Process of Permitting NetAnim,” Ns3 Projects, Jan. 19, 2023. https://ns3simulation.com/what-is-the-role-of-netanim-in-ns3-simulator-how-do-i-enable-netanim/ (accessed Apr. 16, 2025).

[20] F. M. Cuenca-Acuna, C. Peery, R. P. Martin and T. D. Nguyen, "PlanetP: using gossiping to build content addressable peer-to-peer information sharing communities," High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on, Seattle, WA, USA, 2003, pp. 236-246, doi: 10.1109/HPDC.2003.1210033. keywords: {Peer to peer computing;Data structures;IEEE Membership Directory;Internet;Environmental management;Distributed computing;Computer science;Computational modeling;Virtual prototyping;Collaboration},

[21] NetworkX, “NetworkX — NetworkX documentation,” networkx.org. https://networkx.org/

[22] E. W. Weisstein, “Regular Graph,” mathworld.wolfram.com. https://mathworld.wolfram.com/RegularGraph.html

APPENDIX

D-Regular Graph with X nodes total : R_D N = X

Cluster with X clusters and Y nodes in each one : Cl_X N = Y

Tree with X nodes: T_X

Megagraph with X clusters : M_X

Topology	Alg	Average Tried Requests	Average Unique Query Hits	Average Success Rate	Redundant Query Hits	Average Hops	Average Latency	Average Total Work	Average Wasted Requests	Average Query Hit Efficiency	Count of Zero Work Done Nodes
R_3 N = 10	F	2	1	100%	0%	1	8.28×10^{-3}	16	12%	12%	0/8
	NF	2.9	1.1	100%	0%	1.4	9.94×10^{-3}	18.6	22%	16%	2.7/8
	RW	3.2	1.1	100%	5%	1.1	8.89×10^{-3}	18.1	21%	11%	2.8/8
R_3 N = 10 Disabled	F	2	1	100%	0%	1	8.38×10^{-3}	14	12%	12%	0/8
	NF	2.8	1	100%	0%	1.1	8.84×10^{-3}	13.4	17%	11%	1.9/8
	RW	3.4	0.8	80%	5%	1.2	9.08×10^{-3}	16.6	18%	9%	1.9/8
R_3 N = 20	F	1	2	100%	0%	4	2.08×10^{-2}	60	47%	44%	0/18
	NF	2.9	1	90%	0%	5.8	2.90×10^{-2}	73.7	53%	21%	3/18
	RW	3.5	0.9	85%	0%	5.2	2.66×10^{-2}	68.9	49%	17%	4.1/18
R_3 N = 20 Disabled	F	1	1	100%	0%	4	2.08×10^{-2}	35	50%	22%	0/18
	NF	3.5	0.6	60%	0%	4.9	2.57×10^{-2}	47.4	39%	13%	4.2/18
	RW	3.8	0.7	70%	0%	4.8	2.48×10^{-2}	52.2	39%	11%	3.9/18
R_3 N = 30	F	1	1	100%	0%	4	2.09×10^{-2}	53	48%	14%	4/28
	NF	3.2	1.2	95%	0%	9.1	4.19×10^{-2}	114	45%	24%	6.2/28
	RW	4.2	0.8	75%	2%	7.2	3.39×10^{-2}	104.5	52%	13%	6.7/28
R_3 N = 30 Disabled	F	1	1	100%	0%	4	2.09×10^{-2}	46	41%	14%	3/28
	NF	4.2	0.6	55%	0%	5.3	2.62×10^{-2}	76.5	42%	6%	7.6/28
	RW	4.5	0.3	30%	0%	5.7	2.77×10^{-2}	76	40%	3%	8.3/28

R_3 N = 40	F	1	1	100%	0%	4	2.10×10^{-2}	72	53%	10%	6/38
	NF	3	1	95%	0%	7.3	3.52×10^{-2}	98.5	42%	12%	15.2/38
	RW	3.8	0.8	80%	0%	6.2	3.04×10^{-2}	91.5	40%	9%	16.2/38
R_3 N = 40 Disabled	F	1	1	100%	0%	4	2.10×10^{-2}	66	51%	10%	6/38
	NF	3.6	0.8	70%	0%	7.6	3.61×10^{-2}	92.4	46%	12%	13.2/38
	RW	4.3	0.7	65%	0%	6.2	2.98×10^{-2}	84	44%	7%	13.9/38
R_3 N = 50	F	2	2	100%	33%	7	3.40×10^{-2}	323	63%	20%	0
	NF	3.5	0.9	80%	0%	9.6	4.48×10^{-2}	141.8	50%	11%	14.9
	RW	4.3	1	80%	0%	8.5	3.98×10^{-2}	301.6	66%	8%	5.7
R_3 N = 50 Disabled	F	2	1	100%	0%	6	2.93×10^{-2}	166	59%	7%	0
	NF	4.8	0.2	25%	0%	9.4	4.41×10^{-2}	94.8	42%	2%	15.4
	RW	4.5	0.5	45%	0%	7.4	3.45×10^{-2}	173.3	46%	4%	11
R_4 N = 200	F	2	2	100%	33%	5	2.52×10^{-2}	1037	58%	5%	0
	NF	4.2	0.6	55%	0%	2.8	1.56×10^{-2}	149.8	26%	0%	139.2
	RW	3.8	0.8	75%	2%	2.9	1.63×10^{-2}	190.9	23%	0%	138.5
R_4 N = 200 Disabled	F	2	1	100%	0%	4	2.08×10^{-2}	376	53%	1%	7
	NF	5	0.1	5%	0%	2	1.21×10^{-2}	48	8%	0%	140.3
	RW	4.3	0.5	50%	0%	2	1.27×10^{-2}	84.8	9%	0%	136.2
R_5 N = 200	F	2	5	100%	79%	6	2.96×10^{-2}	7008	45%	7%	0
	NF	4.3	0.6	50%	0%	8.2	3.80×10^{-2}	267.4	41%	1%	104.8
	RW	4.5	0.5	40%	1%	5.2	2.59×10^{-2}	278.4	36%	1%	108.6
R_5 N = 200 Disabled	F	2	3	100%	76%	5.9	2.90×10^{-2}	1896	43%	10%	0
	NF	4.7	0.1	15%	0%	5	2.56×10^{-2}	87.8	14%	0%	126.7

	RW	5	0.1	10%	0%	8.5	4.13×10^{-2}	133.3	17%	0%	117.6
Cl_3 N = 10	F	1	1	100%	80%	1.8	1.17×10^{-2}	340	25%	15%	0
	NF	3.5	0.8	80%	0%	2.6	1.49×10^{-2}	93.5	44%	2%	28.4
	RW	3.8	0.7	70%	0%	2.9	1.63×10^{-2}	88.4	41%	2%	29.3
Cl_3 N = 10 Disabled	F	1	1	100%	66%	1.7	1.11×10^{-2}	182	21%	8%	8
	NF	3.8	0.5	50%	0%	1.3	9.55×10^{-3}	53.9	27%	0%	29.1
	RW	4.4	0.2	25%	0%	2	1.22×10^{-2}	66.5	31%	0%	26.4
Cl_3 N = 20	F	1	1	100%	90%	1.9	1.27×10^{-2}	1275	13%	15%	0
	NF	4	0.6	60%	0%	3.7	1.94×10^{-2}	140.8	38%	1%	67.7
	RW	4	0.5	45%	2%	2.9	1.65×10^{-2}	110.4	28%	1%	78.1
Cl_3 N = 20 Disabled	F	1	1	100%	85%	1.9	1.22×10^{-2}	597	8%	10%	29
	NF	4.5	0.2	25%	0%	2.4	1.47×10^{-2}	73.5	20%	0%	68.3
	RW	4.7	0.2	20%	0%	2	1.29×10^{-2}	64.8	17%	0%	71.8
Cl_3 N = 30	F	1	1	100%	93%	1.9	1.30×10^{-2}	2810	8%	16%	0
	NF	4.1	0.4	40%	0%	2.8	1.56×10^{-2}	159.9	32%	0%	115.7
	RW	4.3	0.5	50%	0%	4.2	2.19×10^{-2}	136.4	25%	0%	124.8
Cl_3 N = 30 Disabled	F	1	1	100%	90%	1.9	1.24×10^{-2}	1799	8%	10%	0
	NF	4.7	0.2	20%	0%	1.8	1.19×10^{-2}	91.2	17%	0%	109
	RW	4.8	0.1	10%	0%	2	1.24×10^{-2}	92.2	17%	0%	107.2
Cl_3 N = 4	F	1	1	100%	50%	1.5	1.04×10^{-2}	67	58%	13%	0
	NF	1.9	1	100%	0%	1.8	1.18×10^{-2}	34.5	40%	6%	9.9
	RW	2.7	0.9	95%	2%	1.7	1.14×10^{-2}	44.9	42%	6%	9.9
Cl_3 N = 4 Disabled	F	1	1	100%	0%	1	8.32×10^{-3}	40	52%	4%	2

	NF	3.1	0.7	65%	0%	1.1	8.74×10^{-3}	37.4	37%	2%	8.2
	RW	3.4	0.7	70%	0%	1.3	9.37×10^{-3}	44.2	37%	2%	7.2
M_3	F	1	1	100%	0%	2	1.24×10^{-2}	69	26%	4%	0
	NF	3.8	0.6	55%	0%	2.8	1.57×10^{-2}	62	37%	2%	28.2
	RW	3.6	0.7	65%	0%	2.7	1.54×10^{-2}	81.4	40%	2%	25.1
M_3 Disabled	F	1	1	100%	0%	2	1.25×10^{-2}	52	22%	4%	5
	NF	4.2	0.6	55%	0%	2.6	1.50×10^{-2}	49	25%	2%	25.1
	RW	3.8	0.6	60%	7%	2.8	1.53×10^{-2}	62.3	25%	2%	23.9
M_5	F	1	1	100%	0%	2	1.26×10^{-2}	315	34%	0%	0
	NF	4.9	0.1	10%	0%	2	1.34×10^{-2}	108.6	17%	0%	186.5
	RW	4.7	0.2	20%	0%	2.8	1.61×10^{-2}	163.6	23%	0%	170.7
M_5 Disabled	F	1	1	100%	0%	2	1.27×10^{-2}	185	22%	0%	57
	NF	5	0.1	5%	0%	2	1.15×10^{-2}	69.9	10%	0%	157.8
	RW	4.8	0.1	15%	0%	2	1.35×10^{-2}	105.7	13%	0%	148.6
T_{10}	F	1	1	100%	0%	2	9.99×10^{-3}	16	25%	25%	0
	NF	2.9	0.9	95%	0%	2	1.01×10^{-2}	15.2	17%	20%	3.6
	RW	3.6	0.8	75%	0%	2	1.00×10^{-2}	15.7	22%	16%	2.7
T_{10} Disabled	F	1	1	100%	0%	2	9.99×10^{-3}	16	25%	25%	0
	NF	2.9	0.8	85%	0%	2	1.01×10^{-2}	14.3	16%	18%	3.6
	RW	3.7	0.7	70%	0%	2	1.00×10^{-2}	16.4	23%	13%	2.9
T_{20}	F	2	1	100%	0%	3	1.35×10^{-2}	53	31%	11%	0
	NF	4.7	0.3	30%	0%	3	1.37×10^{-2}	27.1	29%	3%	8.1
	RW	3.4	0.7	65%	0%	3	1.36×10^{-2}	36.2	23%	7%	8.2

T_{20} Disabled	F	2	1	100%	0%	3	1.36×10^{-2}	53	31%	11%	0
	NF	4.2	0.5	50%	0%	3	1.37×10^{-2}	26.4	23%	6%	9.6
	RW	4.2	0.6	60%	2%	3	1.36×10^{-2}	45.2	26%	6%	7
T_{30}	F	2	1	100%	0%	3	1.38×10^{-2}	68	29%	7%	0
	NF	4.5	0.3	30%	0%	3	1.39×10^{-2}	31.4	24%	2%	16.4
	RW	3.8	0.8	85%	7%	3	1.39×10^{-2}	71.8	29%	5%	11.3
T_{30} Disabled	F	2	1	100%	0%	3	1.38×10^{-2}	68	29%	7%	0
	NF	4.3	0.5	50%	0%	3	1.39×10^{-2}	30.9	20%	4%	18
	RW	3.4	0.8	85%	0%	3	1.39×10^{-2}	62.6	27%	5%	12.6
T_{40}	F	2	1	100%	0%	4	1.75×10^{-2}	85	32%	7%	0
	NF	4.8	0.2	25%	0%	4	1.77×10^{-2}	35.1	21%	1%	24.8
	RW	4.5	0.3	35%	0%	4	1.76×10^{-2}	90.2	31%	2%	16.2
T_{40} Disabled	F	2	1	100%	0%	4	1.75×10^{-2}	85	32%	7%	0
	NF	4.6	0.3	30%	0%	4	1.77×10^{-2}	34.5	19%	2%	26
	RW	4	0.6	60%	11%	4	1.77×10^{-2}	83	27%	4%	18.2
T_{50}	F	2	1	100%	0%	4	1.78×10^{-2}	100	35%	6%	0
	NF	4.5	0.3	30%	0%	4	1.80×10^{-2}	37.2	19%	2%	33.5
	RW	3.9	0.6	55%	0%	4	1.80×10^{-2}	84.8	25%	3%	25.9
T_{50} Disabled	F	2	1	100%	0%	4	1.78×10^{-2}	100	35%	6%	0
	NF	4.6	0.2	25%	0%	4	1.80×10^{-2}	37	19%	1%	33.7
	RW	4.2	0.6	60%	0%	4	1.79×10^{-2}	92	28%	2%	24.6
T_{60}	F	2	1	100%	0%	4	1.81×10^{-2}	115	38%	5%	0
	NF	4.5	0.2	25%	0%	4	1.83×10^{-2}	38.2	17%	1%	43.2

	RW	3.5	0.6	60%	2%	4	1.82×10^{-2}	81.6	22%	3%	35.8
T_{60} Disabled	F	2	1	100%	0%	4	1.81×10^{-2}	115	38%	5%	0
	NF	4.2	0.3	35%	0%	4	1.83×10^{-2}	36.4	15%	1%	44.2
	RW	3.9	0.7	65%	0%	4	1.82×10^{-2}	90.8	25%	2%	33.2
T_{70}	F	2	1	100%	0%	5	2.27×10^{-2}	132	38%	5%	0
	NF	4.8	0.1	10%	0%	5	2.29×10^{-2}	41.1	16%	0%	52
	RW	4.7	0.2	25%	0%	5	2.28×10^{-2}	111.9	28%	1%	38
T_{70} Disabled	F	2	1	100%	0%	5	2.27×10^{-2}	132	38%	5%	0
	NF	5	0.1	10%	0%	5	2.29×10^{-2}	42.9	16%	0%	51.6
	RW	4.8	0.1	15%	0%	5	2.28×10^{-2}	116.1	29%	0%	36.8
T_{80}	F	2	1	100%	0%	5	2.30×10^{-2}	147	39%	5%	0
	NF	4.8	0.1	10%	0%	5	2.31×10^{-2}	42	15%	0%	60.9
	RW	4.8	0.3	30%	0%	5	2.31×10^{-2}	118.2	26%	1%	46.1
T_{80} Disabled	F	2	1	100%	0%	5	2.30×10^{-2}	147	39%	5%	0
	NF	4.8	0.1	10%	0%	5	2.31×10^{-2}	41.5	15%	0%	61.4
	RW	4.8	0.2	20%	0%	5	2.31×10^{-2}	118.5	26%	0%	45.9