A case for ML-Learning Algorithms in solving TSP or the problem of routing in wireless networks

Kyle Rumpza

rumpz020 - *December 18th, 2025*

1 Introduction

Traditional network routing has been very standardized since the creation of the internet to allow for the universality of any end-systems to connect to it despite the differences in protocols, hardware, and software. Typically interior gateway protocols (IGP) like OSPF, or RIP are used to exchange messages and compute forwarding tables for routers. While border gateway protocols link together various ASes together. It has capabilities to perform route selection using local policies, path-based routing, and node-based routing. While BGP allows some customization it can still only determine which route advertisements a gateway router will or won't accept. As the internet of things (IoT) continues to grow rapidly our networks are becoming more expansive, mobile, and complex. There is a growing need for more intelligent network routing that can handle more dynamically changing link distances, fluctuations in traffic congestion, and higher data demands. This paper seeks to look at the utilization of ML-algorithms for routing data in more complex networks.

2 Problem Description

The traveling salesman problem (TSP) is a widely studied problem in both mathematics and computer science with the objective of finding the shortest possible route to a point and back. The issue is there is no known polynomial-time algorithm which can solve TSP optimally in all cases. And algorithms which try solve the problem are NP-hard meaning verifying optimality is easy but finding the solution is too computationally intensive. TSP is also applicable to the world of computer networking when looking at the case of routing data packets through a network. In this case not just the distance is considered but the delay, computational overhead, and traffic intensity play a role in how packets may experience longer transmission times. As the internet continues to grow new technologies are being developed to meet future network demands. The fastest growing being software defined networks (SDNs) which allow network administrators more control over their network design to build more intelligent and efficient networks. We will be testing a variety of machine learning (ML) algorithms on different simulated networks to see how they perform with TSP and in minimizing packet transmission time. The ML algorithms which uti-

lize more informed searches and which can explore state spaces should do better at dynamically finding paths through a network.

New networking technologies such as 6G, a greater prevalence of low-earth-orbiting satellites, and the fact that the number of wireless end-systems are surpassing the number of wired end-systems show the need to advance current infrastructure. "6G and beyond will fulfill the requirements of a fully connected world and provide ubiquitous wireless connectivity for all" Akyildiz et al. [1]. In modern wireless networks users connect wirelessly to various cell towers in their area, but the cell towers themselves are connected physically via underground cables. This structure still requires physical infrastructure (cell tower, cables, base stations) in a local vicinity to users. Not only will the IoT become more mobile it will also see an increase in the number of users and therefore increase the amount of traffic on the network at different times. Because of this we need to test routing (TSP) on different types of networks. Three separate types of networks will be tested, one that represents a static physical network, one that represents a dynamic network with changing traffic congestion, and one that represents a wireless network with changing topology. This is again draws back to the need to find algorithms that perform well at routing or in other words performs well in TSP.

Software defined networks (SDN) have been around for a while and completely changed the traditional understanding of network architecture by separating the control and data planes. This allows SDNs to be logically centralized and can dynamically implement network-specific algorithms, and obtain global information on network topology. SDNs really allow for greater traffic engineering (TE) in which routing is guided by self-defined heuristics. Recently a renewed interest in the topic has come up with the popularity of machine learning growing as more research and development is going into implementing these algorithms into SDNs. Such examples include google's b4 WAN and CISCO+'s NAAS. My algorithms will attempt to solve TSP within a SDN network under the assumption that network state information can be accessed if requested by a central controller.

3 Related Work

I have included 8 separate papers in my literature review on the topics of using ML-techniques in routing in wireless networks and in SDNs. They were all published on either IEEE Access, research gate, or Springer journal. There are some papers not included in this literature review that should be mentioned. The first is Zhao et al. [13] research on ant colony optimization in communication networks. Two articles that contributed were from Medium which is the "Traveling salesman problem using simulated annealing" by Francis Allanah [7]. And "How to solve a routing problem with a genetic algorithm: A practical guide" by Christa Fernandes [6]. Lastly the textbook *Artificial Intelligence: A modern approach* by Stuart Russel and Peter Norvig [11] was a massive help in learning the various ML-algorithms implemented in the code.

Amin et. al [2] discus ways to optimize routing in SDNs with ML-algorithms, they divided ML techniques into three categories: supervised, unsupervised, and reinforcement learning. Research published between 2005 and 2021 is considered in the survey.

Boutaba et. al [3] provided a brief introduction to ML techniques, traffic engineering techniques, and methods for data

gathering in a network. Shows how ML can be used for routing, traffic classification, QoS, anomaly detection, and intrusion detection. This survey covers 500 studies.

Akin et. al [4] Analyzes the use of existing routing algorithms in three different categories, ones that calculate static link costs, one which calculates dynamic link costs, and one that tries to evenly distribute load balance and minimize interference. It also talks about the issue of extracting network state information in terms of overhead computational cost, and inconsistencies created due to delay.

Faezi et. al [5] analyzes a wide range of surveys on the subject of ML in traffic engineering and extracts the performance evaluation, experiment environment, and solution category from these surveys. Offers very useful quantifiable metrics of how ML-techniques can improve SDNs.

Mammeri et. al [8] Comprehensively analyze reinforcement learning approaches for routing with a focus on wireless networks. Provides a very good overview of the of the evolution of these specific ML-techniques and its application in communication networks.

Mendiola et. al [9] survey approaches for traffic engineering, discusses applications for routing in SDNs.

Nunes et. al [10] produced a highly cited survey evaluating diverse alternatives for routing in SDNs as well as analyzing future SDN application trends.

Xie et. al [12] Presents a comprehensive detail of the ML techniques, architecture, and working of SDNs. Various ML algorithms are explored and described in terms of QoS, security, resource management, and traffic classification.

3.1 SDN Background

In conventional networks there was a tight bond between the control and data plane but in software defined networks these planes are separated. The reason for this is that SDNs can flexibility update routing functions by exchanging control logic. SDNs are logically centralized usually with a remote controller that obtains a global view of the network by using topology discovery protocols to get accurate link-state information from switches. However the memory and computational overhead from the centralized controller are a major factor to consider when implementing these networks. Amin et. al places a lot of emphasis on "deciding how much control should be delegated to the controller to avoid bottlenecks"Amin. SDNs abstract the network and allow applications to interact with it. This centralized architecture provides faster overview of the network with smoother programmability and updates but requires much more carefully managed overhead control. It also means that SDNs make routing decisions per flow rather than per-hop. The advantages of SDNs make it very useful for future network programming as our network topologies become more dynamic. SDNs are seeing deployment for various services including enterprise networks, data centers, WANs, optical networks, vehicular networks, and mobile ad-hoc networks.

3.2 ML Techniques and Classifications

To be put simply machine learning is the branch of AI that enables the system to

learn on its own. This means that ML systems can make decisions and identify different patterns on their own. Boutaba et. al [3] presents a comprehensive survey which looks all the way back to the 80s to analyze the history of ML algorithm development. In the 1980s Bayesian networks arose as a directed acyclic graph representation was used to show a link between conditions and probability. Then the end of the 80s saw the creation of the Q-learning algorithm, which is a model free learning technique that converges optimum action-values with probability. The 1990s was the last time so much attention was given to neural networks and many bio-inspired optimization algorithms like genetic algorithms and particle swarm optimization received increased attention. These surveys divided their routing algorithms into several types. Both Amin et al. [2], Mameri et al. [8], and Xie et al. [12] surveyed ML techniques for route optimization based on three categories; supervised learning, unsupervised learning, and reinforcement learning. For supervised learning data is fed into the model and cross validated to ensure correctness. Relies on external knowledge like informed search. Unsupervised learning seeks patterns among an unlabelled dataset. Explores unknown states without guidance like uninformed search. The last category reinforcement learning teaches an agent to make local decisions and take actions based on memory. The most common algorithm for this is Q-learning, and deep Q-learning. These surveys focus on how these different ML algorithms work with varying levels of information about the network for the route optimization problem.

On the other hand Akin and Korkmaz [4] in their survey of routing algorithms separated algorithms based on whether they are able to adapt to changing network conditions, in this case static vs. dynamic link-costs. Their survey was focused much more on how the SDN controller obtains accurate network state information to get optimal paths from these algorithms. There were 3 distinct categories RA-SLCL (routing with static link costs), RA-DLC (routing with dynamic link costs), RA-DLCMI (routing with dynamic link costs and minimum interference). These surveys differed in their approaches to analyzing routing algorithms. The former classifications seek to analyze routing algorithms based on how they perform with different levels of information provided. This is important as the more topology information we need to discover from the controller, or the amount of data that needs to be in the memory ("knowledge base") of the RL-algorithm can drastically increase the overhead computational cost on the network. And also poses challenges to how we disseminate information to the other nodes. The latter approach used by Akin and Korkmaz et al. [4] focuses on these issues and makes a case for obtaining accurate NSI to allow these algorithms to perform correctly. "We plan to investigate how to effectively collect the NSI while minimizing the load on the controller and the message overhead throughout the network". It is important to also note that while cloud computing has allowed for seemingly infinite storage and computational data for ML-algorithms it isn't set up for many traditional networks.

3.3 Applications of ML in Networking

Traditional algorithms are not suitable for SDN because their convergence and responses are slow to changing network conditions. We need new solutions to meet more elastic demands. "The Internet has become extremely difficult to evolve both in terms of its physical infrastructure as well as its pro-

tocols and performance" Nunes. [10]. In traditional networks trying to configure these new high-level policies combined with integrated "black-box" devices makes network management error-prone. Most of these surveys along with my own project focuses on how machine learning can improve traffic engineering in networks. Generally traffic engineering is defined as the practice of adjusting and managing data flow across the network. This includes routing optimization, load balancing, congestion control, and quality of service (QoS). Some of these surveys such as Boutaba et. al [3] gives a much more comprehensive view of the service improvements made by ML-based algorithms. Some other services these algorithms can provide are predicting faults, resource management, security, and traffic classification. Mendiola et. al [9] specifically talks about Bandwidth on Demand (BoD) a service provided by ISPs to large corporations, which will require much more dynamic solutions as we require larger and larger datasets for training AI models, and for more distributed networks.

3.4 Results

Faezi et al.[5] had organized results spanning a large variety of surveys. In the study experiments run using deep RL and traditional RL algorithms showed a 14% increase in forwarding table hits, which means higher throughput. And out of the evaluation metrics used performance optimization was the highest improvement with 29%, and attack detection was the second with a 23% increase.

Mendiola et al. [9] found improvements with various ML-based routing and forwarding protocols with different communication APIs operating on different networking layers. Concluded the best improvements and most complete solutions will involve D-CPI, A-CPI, and MI protocols which communicate information across all 3 layers.

Boutaba et al. [3] found improvements for traffic prediction using time series forecasting (TSF), which constructs a regression model drawing correlation between future traffic demand and previously observed values. Also found applications for congestion control. He found that ML-based packet loss classifiers out performed TCP-vegas by using strategies such as active queue management (AQM), and more dynamic congestion window updates. ML AQM schemes mitigate the limitations of the "drop-tail" used to manage switch buffers and increase queue stability. And dynamic approaches to congestion window control is important for use on different networks such as satellite communication where congestion windows are updated differently.

Akin et al.[4] showed the difference of algorithm performance with accurate NSI vs inaccurate NSI. And tested periodically getting NSI information for 3, 5, 10 seconds. The results showed that periodically getting NSI information produces inconsistencies due to delays between nodes and the controller. But found overall improvements of accurate NSI in performance. They also look into algorithms that route paths dynamically based on future network demands (RA-DLC) but this problem was found to be NP-Complete.

Mammeri et al. [8] collected data from various routing protocols and analyzes their performances. Found a correlation between algorithms which perform with higher throughputs also had a much higher network overhead

3.5 Conclusion

The studies observed conclude that ML-based techniques improve route optimization and have proven and tested applications for traffic engineering. And have potential to help with other networking problems such as security, fault management, and traffic prediction. The problem of trying to update traditional network infrastructure to handle newer challenges is new and recently has seen renewed interest from the wider networking community. The application of these new systems can improve our modern network and build upon it. Many studies proposes future research in the fields of hardware with new controller and switch designs. But also for use in cloud services, wireless networks, and in advanced network security.

4 Approach to Solving

Because networks are really just a collection of routers and hosts connected by various links (either physical or wireless) we can represent networks in the form of an undirected graph. The nodes in the graph will represent the hosts and routers, while the edges are the links which connect them. The edge-cost or "link-costs" will represent the distance between two nodes. Various ML-based routing algorithms will be used to compute a path from a source node to a destination node.

The series of ML-based algorithms were learned in my CSCI 4511W course to attempt solving TSP. These algorithms can be divided into 3 broad categories: Uninformed Search, Informed Search, and Simulation-based Search. Uninformed search strategies explore the state space without additional information other than the current state and problem structure. Informed searches have a heuristic function that can help guide them towards hopefully more promising paths. Simulation based search uses random sampling, and simulations to explore the state space, which is especially useful for complex state spaces.

Name	Run Time	Space Time	Short Description
Uninformed Search:			
UCS	$O(b^{C^*})$	$O(b^{C^*})$	Similar to OSPF (Open Shortest Path First) in network routing; always chooses the lowest path cost
Informed Search:			
GBFS	$O(b^m)$	$O(b^m)$	Disregards edge costs and routes based on some heuristic function
A-star	$O(b^d)$	$O(b^d)$	Combines UCS with a heuristic function to determine the path
CSPF Backtracking	$O(V+E)$	$O(V+E)$	Uses specific state information with DFS to guide its search. Similar to the MPLS routing protocol, which utilizes label-switched paths as a constraint in routing. Can be configured to constrain based on values such as the number of hops or reserved bandwidth.
Simulation-Based Search:			
Genetic Algorithm	$O(g \cdot p \cdot f)$	$O(p)$	Uses an analogy of evolution to constantly evolve a generated initial state space based on heuristic evaluation and randomness to determine paths. Similar to OSGA (Open Shortest Path First and Genetic Algorithm) in networking.
Monte Carlo Tree Search	$O(n \cdot t)$	$O(n)$	Combines tree search with random play-out simulations evaluated by a heuristic function to generate a path. A type of reinforcement learning as it uses back-propagation to update previous nodes' heuristic evaluation.

For the informed search algorithms some variants are used in testing using different heuristics to generate paths. For example there is a A star search variant using a distance heuristic which performs closer to greedy best first search. Heuristics used in informed search will generally attempt to minimize the delay, traffic, distance, and number of hops in a path. This is because even if the distance a path takes is physically longer than another path found, the transmission speed of that original path could be faster.

Once an algorithm has been run on the network graph and a path is found we will have a simulated self-defined packet class that will move along each node listed along the path. The packet will then track the delay it incurs traveling from source to destination node. To simulate packet bursts and have a more consistent way to ensure packet delay correctness while testing, packets are fragmented similar to how IP datagrams are fragmented. Packet size is regulated by a maximum transmission unit (MTU) and all fragments of a packet contain the same group id which allows us to track the delay for the whole packet. Because we need to simulate different types of network conditions to add random/dynamic elements we will have three network types. One network will be static where edge-costs/distances don't change. Another will be dynamic where packets increase traffic intensity and delay on each node in that path as they travel which should encourage algorithms to select new paths to receive lower delays. And the last network will be an ad-hoc wireless network. In networking an ad-hoc network is a decentralized type of wireless network which doesn't rely on physical infrastructure but instead each node participates in routing between each other. These networks have two key features being mobility and flexibility. Meaning these networks may have nodes removed or distances and delays changed randomly.

Each algorithms effectiveness will be tested by a variety of factors. The path generated by the algorithm will tell us if an optimal path is found, in other words if it's complete. Optimality in this case is the route with the lowest total delay. The packets sent through the network will give us the run time and their total delay. In dynamic networks we can test the algorithms for load balancing. And in ad-hoc networks we can test the ability of these algorithms to update paths dynamically. By contrasting results between the static and dynamic networks we can see which algorithms are better in operating in complex networks.

Attributes to test for: Optimality, path distance, total delay, run time, dynamic path routing

5 Experimental Design

The first image is the class representing a node in the network graph and its related functions. A node is defined by a name string, a dictionary containing its adjacent neighbors and distances to those nodes, integer for processing delay, integer for propagation delay, and an integer for current traffic congestion. The most important functions here are process packet and dynamic process packet. The process packet function takes a packet and adds the current node's processing delay to that packet. If the packet is the first packet entering the node then it also receives propagation delay. The reason other packets don't receive this delay is because packets in traditional networks are pipelines through so after the first packet arrives the other packet fragments are directly behind. If the packet length is less than the MTU

than its transmission time is properly adjusted so the packet receives less delay. In dynamic networks we use the dynamic process packet function. This function acts similarly to the other process packet function with one addition that the packet increases the current node's traffic by 1 and adds it to the packet's delay. This is because in dynamic networks we want packets passing through to increase traffic congestion to see if different ML-algorithms can adapt and find alternative paths.

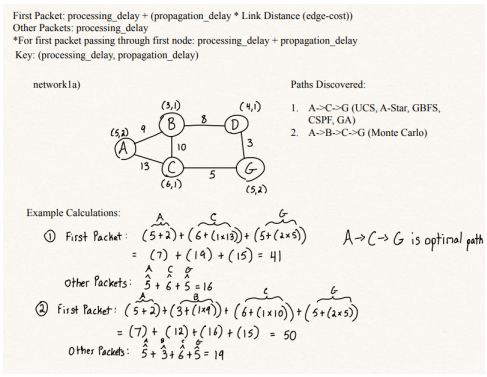
```python
class Node:
    def __init__(self, name):
        self.name = name
        self.neighbors = {} # format is Node: {'cost': cost}
        self.delay = 0
        self.prop_delay = 0
        self.traffic = 0

    def __lt__(self, other):
        return self.delay < other.delay

    # Creates adjacency list showing adjacent nodes and related cost
    def add_neighbor(self, neighbor, cost):
        if neighbor not in self.neighbors.keys():
            self.neighbors[neighbor] = cost
```

This is another key networking object class representing a packet. Packets are defined by a group id attribute, packet size attribute, a boolean to tell if the packet is the first packet after fragmenting, and an integer tracking the total delay incurred by the packet. One important function is the create children function. When the packet size is over the predefined MTU of 1024 the packet is fragmented (divided) into several smaller "child" packets with the same group id, however each of these packets do not have the first attribute. Another useful function is reset delay which sets the total packet delay back to zero for testing purposes.

```python
class Packet:
    # gid = group id
    # size in bits (int)
    # first should be boolean indicating first packet
    def __init__(self, gid, size, first):
        self.gid = gid
        self.size = size
        self.first = first
        self.delay = 0

    # Splits up the packet to send through the network
    def create_children(self):
        children = []
        num_children = self.size // MAX_THROUGHPUT
        n_remaining = self.size % MAX_THROUGHPUT
        self.size = MAX_THROUGHPUT
        for _ in range(num_children - 1):
            children.append(Packet(self.gid, MAX_THROUGHPUT, False))
        if n_remaining > 0:
            children.append(Packet(self.gid, n_remaining, False))
        return children

    # Sets the accumulated delay of the packet back to zero
    def reset_delay(self):
        self.delay = 0

    # print string for the packet
    def __str__(self):
        return f"({self.gid})({self.size}){self.first}"
```

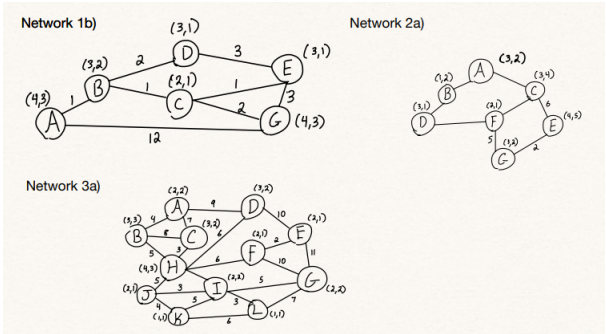
These functions are used to simulate routing a packet through a network along the specified path given. Packet group transmission is a function which processes packets of the same id at each node along the given path and then sums the delays of each packet. The dynamic packet group transmission function operates the same way except dynamically processes the packet at each node along the given path. Again the reset packets function is used to reset the delays between simulations. Below is also an example representation of how a packet is fragmented by the create children function.

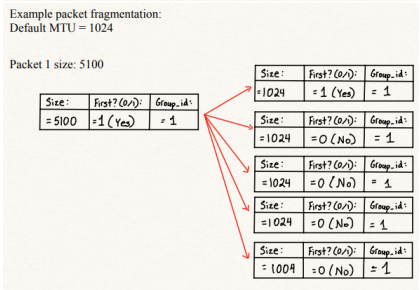
```python
# Simulates transmission and adds delays of all packets together of a certain group id
def packet_group_transmission(path, packets, gid):
    total_delay = 0
    for p in packets:
        if p.gid == gid:
            for i in range(len(path) - 1):
                path[i].process_packet(p, path[i + 1])
            path[len(path) - 1].process_packet(p, None)
            total_delay += p.delay
    return total_delay

# Simulates a dynamic transmission and adds delays of all packets of a certain group id
def dynamic_packet_group_transmission(path, packets, gid):
    total_delay = 0
    for p in packets:
        if p.gid == gid:
            for i in range(len(path) - 1):
                path[i].process_packet_dyn(p, path[i + 1])
            path[len(path) - 1].process_packet_dyn(p, None)
            total_delay += p.delay
    return total_delay

# resets all packets in a packet cluster
def reset_packets(packets):
    for p in packets:
        p.reset_delay()
```



Here is the representation of a network graph within my project. For each network the source node is always 'A' and the destination node is always 'G'. This is so we know the start and end point for each network despite the networks having different topologies. Below I included visual representations of the network graphs. And a sample calculation of the packet delays expected in the results for paths generated by algorithms for routing in network 1a.

```
# Set up for creating small-sized networks
def network1a():
    edges = {('A', 'B'): 9, ('A', 'C'): 13, ('B', 'C'): 10, ('B', 'D'): 8, ('C', 'G'): 5, ('D', 'G'): 3}
    nodes = {name: Node(name) for name in ['A', 'B', 'C', 'D', 'G']}
    config_graph(nodes, edges)
    nodes['A'].set_delay(5, 2)
    nodes['B'].set_delay(3, 1)
    nodes['C'].set_delay(6, 1)
    nodes['D'].set_delay(4, 1)
    nodes['G'].set_delay(5, 2)
    return nodes, edges
```





This is the main loop called on to run simulations in a static network. The function takes in as inputs a machine learning algorithm to perform routing to generate a path and the packet cluster being used to transmit through the network. Each ML-algorithm used returns the path and the total distance of the path found. The simulation works by starting a timer, using the given algorithm to perform routing, and then processing each packet by group id. After each packet is processed the timer stops and the algorithms name, path, distance, total delay, and runtime are recorded in a global static results library. The packets delays are then reset to zero as clean up for the next simulation iteration.

```
# Function for running a packet simulation through a network
def run_algorithm(algorithm_name, start, goal, algorithm_function, group_ids, packets, *args):
    delays = []
    start_time = time.time() # start timer

    path, distance = algorithm_function(start, goal, *args) # run algorithm
    for id in group_ids:
        delays.append(packet_group_transmission(path, packets, id))

    end_time = time.time() # end timer
    total_time = end_time - start_time

    # Recording results in table
    static_network_results[algorithm_name] = {'path': path, 'distance': distance, 'total_delay': sum(
    reset_packets(packets)
```

This is the main loop called to run simulations in a dynamic network. The function takes in an algorithm to use for routing and packet cluster as inputs. The simulation runs by first setting up local variables to track of current best paths and distances seen. Then a timer is started and the algorithm is run first to generate an initial path (one that is equal to the static network paths since there is no traffic congestion). Then each packet is sent through the network and processed dynamically. After this the ML-algorithm is run after each iteration to re-determine the path. Here we are curious to see if the algorithm being used will adapt and change routes due to higher delays along previously used routes. So if the path generated is different than the path previously generated by the algorithm we store this path as an old path with the old distance and the next packet is routing along the new path. After all packets have been transmitted the timer stops and the results are recorded in a

global dynamic results library similar to the static network simulation loop. However one difference is that the dynamic networks also record information about which paths the algorithm had previously used to see if it can adapt to changing network delays.

```python
# Function for testing a specfic algorithms routing
def run_dynamic_algorithm(algorithm_name, start, goal, algorithm_function, nodes, group_ids, packets,
    # tracking libraries
    delays = []
    best_path = []
    old_paths = {}
    best_distance = 0

    start_time = time.time() # start timer
    best_path, best_distance = algorithm_function(start, goal, *args) # Get initial paths for first pa
    for id in group_ids:
        delays.append(dynamic_packet_group_transmission(best_path, packets, id))
        temp_path, temp_distance = algorithm_function(start, goal, *args) # Run algorithm again to see
        if temp_path != best_path:
            #print_path_change(temp_path, best_path, temp_distance, best_distance)
            old_paths[print_path(temp_path)] = temp_distance
            best_path = temp_path
            best_distance = temp_distance

    end_time = time.time() # end timer
```

```python
# A function to simulate the randomness that can occur in ad-hoc networks
def scramble_network(nodes, edges):
    if random.random() <= 0.33: # Flexibility (triggers 1/3rd of the time)
        new_nodes, new_edges = reconfigure_graph(nodes, edges)
        for node in new_nodes:
            nodes[node].reset_neighbors()
        config_graph(new_nodes, new_edges)
        return new_nodes, new_edges
    new_nodes = delay_scrambler(nodes)
    new_edges = distance_scrambler(edges) # Mobility
    for node in new_nodes:
        nodes[node].reset_neighbors()
    config_graph(new_nodes, new_edges)
    return new_nodes, new_edges

# A function for assigning random distances to edges between the ranges of the highest and lowest di
# Simulates mobility in ad-hoc networks
def distance_scrambler(edges):
    max_distance = max(edges.values())
    min_distance = min(edges.values())
    for edge in edges:
        rand_int = random.randrange(min_distance, (max_distance + 1))
        edges[edge] = rand_int
    return edges

# A function for assigning random delays to the nodes
# Simulates changing delays in ad-hoc networks that could be caused by "line-of-sight", interference,
def delay_scrambler(nodes):
    delays = [node.delay for node in nodes.values()]
    max_delay = max(delays)
    min_delay = min(delays)
    for node in nodes:
        rand_int = random.randrange(min_delay, (max_delay + 1))
        prop_delay = nodes[node].prop_delay
        nodes[node].set_delay(rand_int, prop_delay)
    return nodes
```

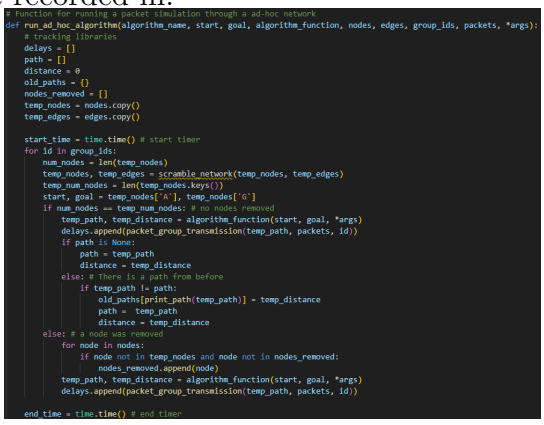
Below are the functions required for simulating some of the key features present in ad-hoc wireless networks. Ad-hoc wireless networks are different from wired networks because they have to be both mobile and flexible. In these networks hosts can be mobile which means distances to base stations or mobile hosts can change randomly. So distances and delays (due to longer latency) should randomly change. Ad-hoc networks use routing between hosts if they are within range of each other but these mobile hosts can disconnect at any time meaning our simulated nodes can be removed from the network graph randomly. To replicate both mobility and flexibility the class scramble network is used. This function 2/3rds of the time will call both the distance scrambler and delay scrambler functions. The delay scrambler function assigns each node in the network new processing delays randomly in the range of the nodes minimum and maximum delay. The distance scrambler function works by assigning each edge in the network a random distance in the range of the network's longest and shortest distance. This simulates mobility in the ad-hoc network.

1/3rd of the time the scramble network function will call the reconfigure graph function. The reconfigure graph function chooses a node randomly to remove as long as it isn't the start or goal node. We then track all of the neighbors of the node being removed and connect it to the next nearest node setting the distance to a random number generated in the range between the maxim distance and minimum distance in the network graph. The network graph is configured again to ensure each node's neighbors dictionary is correct and doesn't contain duplicates.

```python
# Removes a node randomly and reconfigures the graph after
def reconfigure_graph(nodes, edges):
    if not nodes:
        return nodes, edges
    max_distance = max(edges.values())
    min_distance = min(edges.values())
    removed_node = random.choice(list(nodes.keys())) # randomly select node to remove
    if removed_node == 'A' or removed_node == 'G':
        # do nothing
        #print("Can't remove start or goal node")
        return nodes, edges
    #print("removed node: " + str(removed_node))
    affected_nodes = nodes[removed_node].neighbors # the removed nodes neighbors
    nodes_copy = nodes.copy()

    del nodes_copy[removed_node]
    edges_copy = {edge: cost for edge, cost in edges.items() if removed_node not in edge} # new edge list ex

    # Reattach affected neighbors
    for neighbor in affected_nodes.keys():
        if neighbor not in nodes_copy.values():
            print("skipping")
            continue
        # Find nearest neighbor
        remaining_nodes = [node for node in nodes_copy.keys() if node != neighbor.name] # remaining nodes th
        if not remaining_nodes:
            continue
        nearest_node = min(remaining_nodes, key=lambda node: edges_copy.get((neighbor, node), float('inf')))

        # Reconnect to nearest node
        cost = random.randrange(min_distance, (max_distance) + 2) # sets new cost
        # Need to alter cost of the edge if it already exists in edge list (instead of adding)
        if (neighbor.name, nearest_node) in edges_copy:
            edges_copy[(neighbor.name, nearest_node)] = cost
        elif (nearest_node, neighbor.name) in edges_copy:
            edges_copy[(nearest_node, neighbor.name)] = cost
        else: # has to be a new node
            edges_copy[(neighbor.name, nearest_node)] = cost
    return nodes_copy, edges_copy
```

This is the main loop for running ad-

hoc network simulations. As inputs the function takes in the algorithm to use for routing, packet cluster to transmit, as well as the nodes and edges of the network graph to make copies to to perform scramble network function on without altering the original graph. The simulation runs in the following manner. First a timer is started then for each packet the network is scrambled first. If no nodes were removed then we know that the distances and delays were scrambled emulating the mobility of an ad-hoc network. The algorithm is run again to find a new route which it most likely will since the delays and distances have been altered dramatically. Any path changes made are recorded in a old paths library. If a node was removed this is emulating flexibility within our ad-hoc network simulation. We need to update our node list for our network graph and run our algorithm again to determine a new route. After all packet have been transmitted the timer is stopped and the results are recorded in a ad-hoc simulation results library in the same format that the dynamic simulations were recorded in.



```
# function for running a packet simulation through a ad-hoc network
def run_ad_hoc_algorithm(algorithm_name, start, goal, algorithm_function, nodes, edges, group_ids, packets, *args):
    # tracking libraries
    delays = []
    path = []
    distance = 0
    old_paths = {}
    nodes_removed = []
    temp_nodes = nodes.copy()
    temp_edges = edges.copy()

    start_time = time.time() # start timer
    for id in group_ids:
        num_nodes = len(temp_nodes)
        temp_nodes, temp_edges = scramble_network(temp_nodes, temp_edges)
        temp_num_nodes = len(temp_nodes.keys())
        start, goal = temp_nodes['A'], temp_nodes['G']
        if num_nodes == temp_num_nodes: # no nodes removed
            temp_path, temp_distance = algorithm_function(start, goal, *args)
            delays.append(packet_group_transmission(temp_path, packets, id))
            if path is None:
                path = temp_path
                distance = temp_distance
            else: # There is a path from before
                if temp_path != path:
                    old_paths[print_path(temp_path)] = temp_distance
                    path = temp_path
                    distance = temp_distance
        else: # a node was removed
            for node in nodes:
                if node not in temp_nodes and node not in nodes_removed:
                    nodes_removed.append(node)
            temp_path, temp_distance = algorithm_function(start, goal, *args)
            delays.append(packet_group_transmission(temp_path, packets, id))

    end_time = time.time() # end timer
```

6 Results Analysis

6.1 Static Network Results:

Uniform cost search, A star search, Constrained shortest path first algorithms found the same paths in all four networks. For the small and medium networking graphs they consistently found paths with the lowest delay. Genetic algorithms performed generally well finding the optimal paths in small to medium-sized network graphs and routes found in larger networks despite being sub-optimal were not the worst delays seen. Monte carlo tree search (MCTS) performed poorly in both network1b and network 2a finding routes with the highest delays. However in the large scale network 3a MCTS was the only algorithm to find the optimal path receiving the highest delay. In fact the large scale network was the only one in which UCS, A*, and CSPFdidn't perform better. The worst performing algorithm in the static network was greedy best first search which had the highest delays on network 3a and on network 2a. Because distances, delays, and traffic doesn't change in a static network the optimal paths also correlated with the shortest distances. For run time the only algorithm that had a recorded time was the genetic algorithm with a total time between 0.47-1.04 seconds. MCTS had the second highest run time around 0.001 second. And CSPF had the third highest runtime around 1 microsecond.

6.2 Dynamic Network Results:

In network 1a our small scale network all algorithms found the same route and incurred the same delay so it won't be discussed in the remainder of the dynamic results. UCS, A* with the distance heuristic, and CSPF performed the same in all dynamic network simulations choosing the same routing paths. The paths they found were all sub-optimal but generally their delays were not far off from the optimal path. A star search performed well in the dynamic network. In network 2a it demonstrated adaptability by

changing paths to a path with a higher distance but received less delay having a total delay only three off from the optimal solution. In the large-scale network 3a it found a suboptimal path despite changing paths during the simulation having the same delay as UCS and CSPF. MCTS generally performed even better than A star search finding the optimal paths in the small and large-scale networks. And in network 1b it was the only algorithm to find the optimal path with the lowest delay. It performed extremely well considering it never switched paths in any of the networks. However it had a hiccup performed the among the worst in network 2a. The genetic algorithm performed the best finding optimal paths in all networks (except network 1b). GA also switched paths multiple times in medium and large networks demonstrating the ability to adapt to dynamic changes. GBFS performed the worst routes in all of the networks. For runtime genetic algorithms had a fairly high overhead with around 2.0 - 5.7 seconds total time. MCTS and CSPF also had measurable run times but they were around the same times recorded in static networks (0.001 seconds, 1 microsec).

6.3 Ad-hoc Network Results:

Because the distances, delays, and even the network topology is subject to change the results gained are evaluated over several iterations of simulations and the algorithms trend overall. UCS, A* with distance heuristic, and GBFS perform the worst on ad-hoc networks finding suboptimal or the worst paths. This is despite both GBFS and A* switching paths often to avoid high delays. A* search performs sub-optimally in all networks. It is worth noting that A* performs better than GBFS and A* with the distance heuristic. MCTS performs optimally on larger networks but performs either suboptimally or even poorly on smaller network graphs. GA performed very well in most network graphs finding delays close to optimal routing. The algorithm that did the best was CSPF as it regularly found the lowest delays performing optimally most of the time. Genetic algorithms had lower run times in this network simulation roughly around 1.5 seconds for all networks. CSPF and MCTS had run times similar to the previous network results. And A star with the distance heuristic actually had a measured run time of 0.001 seconds.

7 Conclusion

Based on the results we can conclude a few key things about the effectiveness of these algorithms in different network architectures. In static networks where the distances and delays never change uninformed algorithms or uninformed exploration algorithms perform the best. UCS and A* search which builds on UCS by implementing a heuristic function found optimal paths with no overhead.Genetic algorithms and MCTS performed sub-optimally and of course have a higher computational overhead. In dynamic networks simulation based search algorithms performed well finding optimal routes. And demonstrated that they were able to adapt to dynamic network changes by switching paths. Informed searches performed suboptimally in most cases. For ad-hoc networks again we see that the heuristic algorithms outperform the uninformed and informed search algorithms finding lower delay routes. Since the ad-hoc network has constantly changing conditions for algorithms to perform well they must adapt to changing network topology. Informed searches performed sub-optimally but have shorter run

times.

Algorithms that are simulation based on repeated sampling to explore state spaces performed better in dynamic networks. They show a greater efficiency in load balancing, finding lower delay paths, and in adaptability. However they also have the highest runtimes, and much higher space complexity since they need more storage. As pointed out earlier having a large computational overhead can worsen processing delays in an SDN, "deciding how much control should be delegated to the controller to avoid bottlenecks" Amin et al. [2]. Informed searches performed generally well but sub-optimally in all scenarios except for greedy best first search. This is because GBFS picked locally made decisions for shorter path without performing state space exploration. UCS performed better than GBFS despite being uninformed because it at least explored the state space when constructing routes. Algorithms that performed the best specifically did two things right. They all performed playouts or simulations of various routes to find the best. And secondly they all used a heuristic evaluation that considered multiple constraints or metrics which used information about delays, distances, and number of hops in the path to limit to. This correlates closely to the results found in earlier surveys and studies on ML-techniques in SDNs and wireless networks. In the surveys it was found that algorithms which obtained network state information had better performance similar to how heuristics and obtaining global information showed better route aggregation. And that dynamic ML-algorithms that had higher network overhead and had a much higher throughput. This is an indication that the need for adapting network infrastructure is now. Traditional networks do not support centralized controllers and instead employ a distributed algorithm OSPF which is uninformed similar to UCS. This does well when network conditions are relatively stable but hit snares and latency during broadcast storms (short bursts of high volume traffic). And as we become mobile and network traffic can dynamically change, for instance as more computers are introduced into vehicles. This means that networks need to adequately reroute traffic in more intelligent ways to avoid congestion without attempting to drastically increase the number of base stations in a region. SDNs are of great interest as they can really improve network performance but require oversight of an administrator. If there was a way to combine intelligent and informed simulation based algorithms but distribute it and use a messaging system like OpenFlow to propagate network state information this would be the ideal solution for the problem of routing in a dynamic network.

References

[1] Ian F. Akyildiz, Ahan Kak, and Shuai Nie. "6G and Beyond: The Future of Wireless Communications Systems". In: *IEEE Access* 8 (2020), pp. 133995–134030. DOI: 10.1109/ACCESS.2020.3010896.

[2] R. Amin et al. "A survey on machine learning techniques for routing optimization in SDN". In: *IEEE Access* 9.104582–104611 (2021).

[3] R. Boutaba et al. "A comprehensive survey on machine learning for networking: Evolution, applications and research opportunities". In: *Springer* (2018).

[4] Akin E. "Comparison of routing algorithms with static and dynamic link cost in software defined net-

working (SDN)". In: *IEEE Access* 7.148629–148644 (2019).

[5] A. Faezi S. Shirmarz. "A comprehensive survey on machine learning using in Software Defined Networks (SDN) - human-centric Intelligent Systems". In: *Springer* (2023).

[6] Christa Fernandes. *How to solve a routing problem with a genetic algorithm: A practical guide*. https://medium.com/data-and-beyond/how-to-solve-a-routing-problem-with-a-genetic-algorithm-a-practical-guide-a0f0f8aa36db. Accessed: 2024-10-21. Nov. 2023.

[7] Allanah Francis. "Traveling salesman problem using simulated annealing". In: *Medium* (Jan. 2022).

[8] Z. Mammeri. "Reinforcement learning based routing in networks: Review and classification of approaches". In: *IEEE Access* 7.55916–55950 (2019).

[9] A. Mendiola et al. "A survey on the contributions of software-defined networking to traffic engineering". In: *IEEE Communications Surveys* 1617–1634 (2017).

[10] B. Nunes et al. "A survey of software-defined networking: Past, present, and future of Programmable Networks". In: *IEEE Communications Surveys* 1617–1634 (2014).

[11] Peter Norvig Stuart J. Russel. *Artificial Intelligence: A modern approach*. Upper Saddle River: Pearson, 2010.

[12] J. Xie et al. "A survey of machine learning techniques applied to software defined networking (SDN): Research issues and challenges". In: *IEEE Communications Surveys* 393–430 (2019).

[13] Dongming Zhao, Liang Luo, and Kai Zhang. "An improved ant colony optimization for the communication network routing problem". In: *Mathematical and Computer Modelling* 52.11 (2010), pp. 1976–1981. DOI: https://doi.org/10.1016/j.mcm.2010.04.021. URL: %5Curl%7Bhttps://www.sciencedirect.com/science/article/pii/S0895717710002116%7D.