Minimum Interference Routing Algorithm (MIRA)

Report submitted to the SASTRA Deemed to be University

As the requirement for the course

CSE302: COMPUTER NETWORKS

Submitted by

HARIHARAN SUBRAMANIAN

(Reg.No:224003032, B.TECH.CSE)

December 2022

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

KUMBAKONAM, TAMIL NADU, INDIA – 612001

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

KUMBAKONAM, TAMIL NADU, INDIA – 612001

Bonafide Certificate

This is to certify that the report titled "**Minimum Interference Routing Algorithm**" submitted as a requirement for the course ,

CSE302 : COMPUTER NETWORKS for B.Tech. is a bonafide record of the work done by **Shri. HARIHARAN SUBRAMANIAN (Reg.No: 224003032, CSE)** during the academic year 2021-22, in the School of Computing

Project Based Work Viva Voce held on _____

Examiner 1 **Examiner 2**

ABSTRACT

Minimum Interference Routing Algorithm (MIRA) was formulated by M Kodialam and TV Lakshman(2000) for Multipurpose Label Switching Networks (MPLS). MPLS in brief , allows packets of any protocol to traverse, and they follow a label which establishes an explicit route (Label Switched Paths (LSP)) before start of transmission of packet. This enables MPLS to provide bandwidth-guaranteed tunnels and other QoS metrics. MIRA is an online dynamic routing algorithm. It is “online” since it does not have any pre-defined traffic profile. It is “dynamic” since it uses the current network state to determine the routing decisions. The main idea behind MIRA, is to choose paths for an Ingress-Egress Pair(IEP) such that it shall not interfere with future demands. Mathematically, chosen path should result in minimum decrease of maxflow values of all possible IEP. In this project, we will implement MIRA, and analyze the common metrics of Remaining available Maxflow values and no. of rejected requests with other classic routing algorithms like WSP, SWP, MHA and so on.

KEYWORDS : Interference, Routing, QoS, MPLS, LSP, MIRA

Table of contents

List of Figures

List of Tables

CHAPTER 1

1.1 INTRODUCTION

1.1.1 What Happens In A Normal IP Network?

When an internet router receives an IP packet, that packet carries no information beyond a destination IP address. There is no instruction on how that packet should get to its destination or how it should be treated along the way. Each router has to make an independent forwarding decision for each packet based solely on the packet's network-layer header. Thus, every time a packet arrives at a router, the router has to "think through" where to send the packet next. The router does this by referring to complex routing tables.

1.1.2 What is an MPLS network?

Multiprotocol Label Switching, or MPLS, is a networking technology that routes traffic using the shortest path based on "labels," rather than network addresses, to handle forwarding over private wide area networks. As a scalable and protocol-independent solution, MPLS assigns labels to each data packet, controlling the path the packet follows. MPLS greatly improves the speed of traffic, so users don't experience downtime when connected to the network

An MPLS network is Layer 2.5, meaning it falls between Layer 2 (Data Link) and Layer 3 (Network) of the OSI seven-layer hierarchy. Layer 2, or the Data Link Layer, carries IP packets over simple LANs or point-to-point WANs. Layer 3, or the Network Layer, uses internet-wide addressing and routing using IP protocols. MPLS sits in between these two layers, with additional features for data transport across the network

With MPLS, the first time a packet enters the network, it's assigned to a specific forwarding class of service (CoS)—also known as a forwarding equivalence class (FEC)--indicated by appending a short bit sequence (the label) to the packet. These classes are often indicative of the type of traffic they carry. For example, a business might label the classes real time (voice and video), mission critical (CRM, vertical app), and best effort (Internet, email). Each application would be placed in one of these classes.

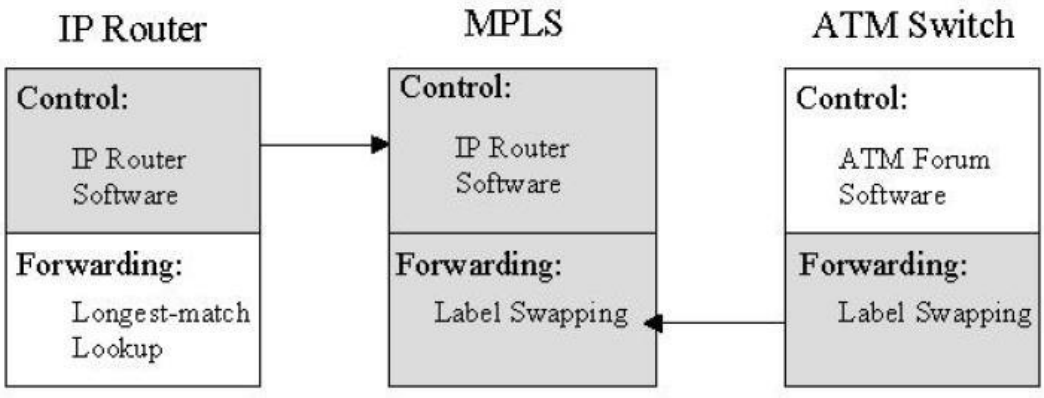

Fig 1.1 MPLS interface and usage in Computer Networks

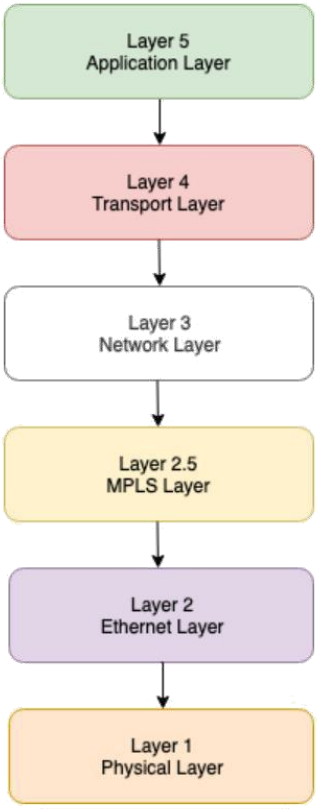

Fig 1.2 MPLS Layer

1.1.3 MPLS Label Structure

1. Label: The label field is of 20 bits, hence the label could take values from 0 to 2^20–1, or 1,048,575. However, the first 16 label values ie from 0 to 15 are exempted from normal use as they have a special meaning.

2. Experimental(Exp): The three bits are reserved as experimental bits. They are used for Quality of Service(QoS).

3. Bottom of Stack(BoS): A network packet can have more than one MPLS labels which are stacked one over another. To ensure which MPLS label is at the bottom of stack we have a BoS field which is of 1 bit. The bit is high (ie value 1) only when that particular label is at the bottom of the stack otherwise its value remains 0.

4. Time to Live(TTL): The last 8 bits are used for Time to Live(TTL). This TTL has the same function as the TTL present in the IP header. Its value is simply decreased by 1 at each hop. The job of TTL is to avoid the packet being stuck in the network by discarding the packet if its value becomes zero.

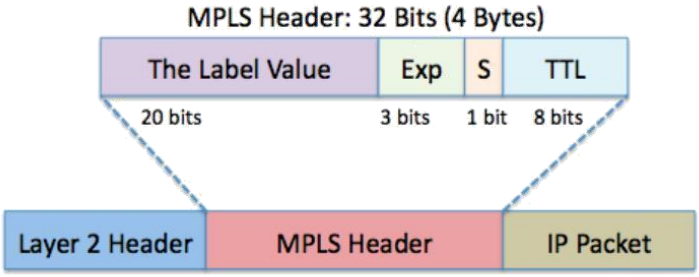

Fig 1.3 MPLS Label Structure

1.1.4 How does MPLS network look like?

MPLS network contains Label Switch Routers(LSR). These routers are capable of understanding MPLS labels and of receiving and transmitting the labeled packet.

There are three kinds of Label Switch Routers present in the MPLS Network:-

1. **Ingress LSR:** These routers are present at the beginning of the MPLS network. Their job is to receive unlabelled IP packet and push the label on top of it.

2. **Egress LSR:** These routers are present at the end of the MPLS network. Their job is to pop the label from the incoming packet and forward the packet as an IP packet.

3. **Intermediate LSR:** These routers are present in between the above two routers. Their job is to receive the labelled packet, swap the label of the packet and forward it to the next hop. They are responsible for the MPLS forwarding of the packet

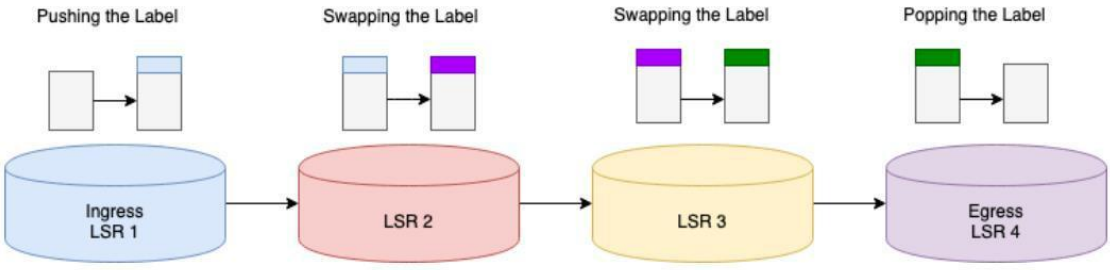

Fig 1.4 MPLS Operations

Thus at any particular router, either of the three steps PUSH, POP or SWAP of the label takes place

1.1.5 Label Switched Paths(LSP)

A label Switch Path(LSP) can be defined as the sequence of label switch routers(LSR) that transmit the packet within an MPLS network. Basically, LSP is a predefined path that the packet takes during the transmission.

The first LSR in an LSP is an Ingress LSR, similarly the last LSR in an LSP is an Egress router followed by intermediate LSR's in between.

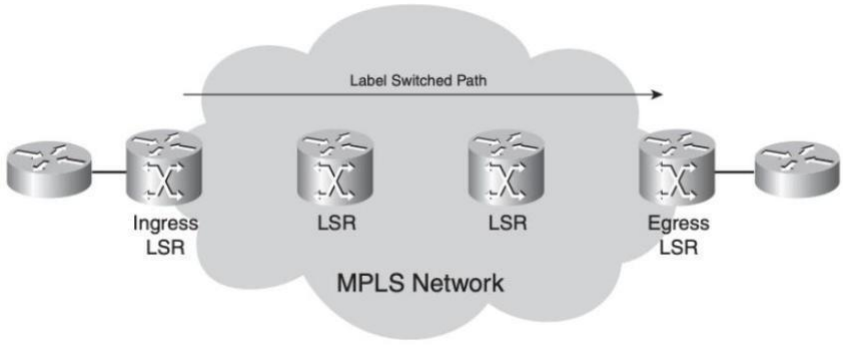

Fig 1.5 MPLS LSP Setup

The main takeaway from here is that in IP forwarding there is no fixed path that a packet must follow whereas in MPLS forwarding we predefine the path ie the LSP's which must be followed during the packet transmission

1.1.6 MPLS Forwarding

1. So now we can easily see how an MPLS packet is forwarded in the network combining the above concepts that we have learned.
2. When a network packet enters the MPLS network, the ingress router places a label on top of it. This label corresponds to a particular path that the packet needs to follow ie the LSP. Different LSP's corresponds to different label stack. The label is distributed using different protocols such as Tag Distribution Protocol (TDP), Label Distribution Protocol(LDP) and Resource Reservation Protocol (RSVP).
3. As in IP forwarding, each router contains a routing table similarly in an MPLS network each LSR contains Label Forwarding Information Base(LFIB). This information base guides the LSR to swap the label with its corresponding outgoing label thus allowing the packet to transmit through the network. The major takeaway here is that the router needs only to see the label on top of the incoming packet, and it does not care about the IP address (source and destination) present inside the packet, thus allowing faster routing through the network.

1.2 ROUTING IN MPLS : A Brief Overview

Before mapping packets onto an LSP, the LSP is set-up using a signalling protocol such as RSVP or LDP (Label Distribution Protocol). A key aspect of LSPs relevant to this paper is that LSPs can be explicitly routed along specific paths. This means that when an LSP is being set-up it is possible to specify all intermediate points between the ingress and egress pairs.

Requirements for MPLS Routing Algorithms:
1. *Necessity to use on-line algorithms:* For traffic engineering purposes, it it usually assumed that all point-to-point demands are known. While this is a valid assumption for network design, for MPLS application this implies that all LSPs that traverse the network are known at the time of initial routing. This is unlikely to be the case in practice. Furthermore, in this offline (all LSPs are known) model the objective usually is to make the most efficient use of the network, i.e., to minimize the resource usage for the LSPs that are being routed. Note that with this objective, it may happen that after the routing has been done there may no available capacity between certain ingress-egress routers (even though a different routingmay have resulted in some available capacity between those routers). In the offline model, this lack of residual capacity between certain ingress-egress pairs is not relevant since all LSPs that need to be routed are known and no future routing requests

are expected. In practice, since the possibility of having to route future LSP demands cannot be excluded, the routing algorithm must be an on-line algorithm capable of routing requests in an "optimal" manner when the requests are not all presented at once and re-routing of existing LSPs is not allowed.

2. Use knowledge of ingress-egress points of LSPs: Even though future demands may be completely unknown, the routers where LSPs can potentially originate and terminate are known since these are the network's edge routers. The algorithm must be able to use any available knowledge regarding ingress-egress pairs and must not always assume that every router can potentially be an ingress and egress point (though this may be the case sometimes). To our knowledge, the algorithm we present is the first algorithm to take ingress-egress information explicitly into account. Note however that the algorithm can be used even if the ingress-egress nodes are not known because we can then assume that all nodes are ingress-egress nodes.

3. Good re-routing performance upon link failure: This is clearly an important performance metric. When a link fails, it must be possible to find alternate routes for as many LSPs as possible. If before failure, certain ingress-egress pairs have no residual capacity available between them then re-routing LSPs between these pairs after link failure is not possible. One of the algorithms that we present tries to maximize a surrogate measure of the residual capacity between all ingress-egress pairs and this makes it perform well upon link failure.

4.Routing without traffic splitting: Though splitting is used for load balancing purposes (by routing demands over multiple LSPs at the ingress point), it is not permissible for the routing algorithm to always split traffic in an arbitrary manner since the traffic being routed may be inherently unsplittable. Hence, the algorithm must be able to route a desired amount of bandwidth between a given ingress-egress pair without being able to split traffic onto multiple paths in an arbitrary way at every potential router in the path even though such splitting could permit better network usage

5. Computational requirements: We show later that the optimal routing in our formulation is NP-hard. Any heuristic or approximation algorithm must be implementable on routers and route servers and must execute within a reasonable time-budget for networks with a few thousand ingress-egress pairs

1.3 NEED FOR A NEW ALGORITHM?

The most commonly used algorithm for routing LSPs is the min-hop algorithm. In this algorithm, the path fro m the ingress to the egress with the least number of feasible links is chosen. This algorithm though simple use s the same information that the proposed new algorithm uses. Its performance in terms of efficient network us age can be easily improved upon with a little more computation. With the rapid rise in processor speeds and w ith traffic trunks not expected to be set-up and torn-down at high rates, it is justifiable to trade-off increased co mputation for more efficient network usage. The min-hop algorithm does not take information on ingress-egre
ss pairs into account nor does it adapt routing to increase chances of successful re-routing upon link failure. T his widest-shortest path algorithm(WSP) finds a feasible min-hop path between ingress and egress such that th e chosen min-hop path has the maximum residual path bottleneck link capacity. This algorithm too does not ta ke ingress-egress information into account. With every node being assumed to be a potential ingress and egres s point, a widest path algorithm without a min-hop restriction does not work well since long paths which incre ase network usage get chosen. Hence, a widest-shortest path hop heuristic performs better. However, if the ing ress-egress pairs are known then an algorithm which picks paths longer than min-hop works well provided it a voids unnecessarily loading certain "critical" links as we shall see later. More sophisticated algorithms in addi tion use the residual bandwidth on the link to influence the weight of the link and the shortest path is chosen w ith respect to these dynamically changing weights. Since the weights are chosen to increase with link load, the idea is to not use up link capacity completely if alternate lower loaded paths are available. This has the tenden cy to make capacity available for future demands. Nevertheless, this algorithm is also oblivious to information regarding ingress-egress pairs and therefore can pick long paths to defer loading on links which may not be im portant to satisfy future demands.

1.4 ILLUSTRATIVE EXAMPLE

These routers serve as sources and destination of future traffic. If routing is done oblivious to the location of t hese sources and destinations of traffic then we may **"interfere"** with the routing of some future demands. We illustrate this with a simple example. Consider the network shown in Figure 1. There are three potential source destination pairs, (S1;D1), (S2;D2),(S3;D3). Assume that all links have a residual bandwidth of 1 unit. We no w have a request for an LSP between S3 and D3 with a bandwidth request of 1 unit. If min-hop routing is used , the route will be 1-7-8-5. Note that this route blocks the paths between S2 and D2 as well as S1 andD1. In thi s example it is better to pick route 1-2-3-4-5 even though the path is longer.

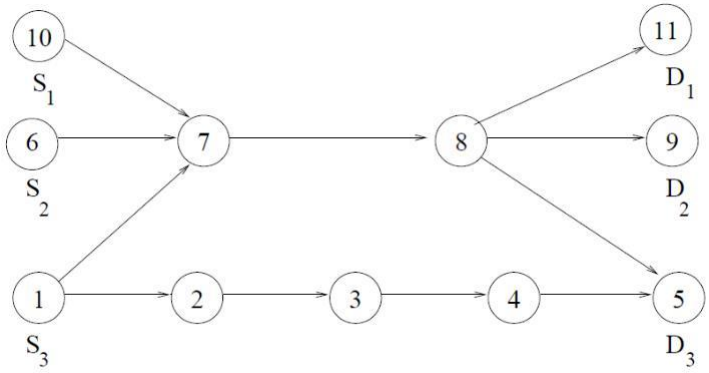

Fig 1.6 Test Network

1.5 KEY IDEAS IN MINIMUM INTERFERENCE ROUTING

1. Interference: The key idea is to pick paths that do not interfere too much with potential future LSP set-up requests (demands) between other source destination pairs. We first have to make this concept of interference more concrete. Consider the maximum flow (maxflow) value v_1 between a given ingress-egress pair (S1, D1). This maxflow value is an upper bound on the total amount of bandwidth that can be routed between that ingress- egress pair (S_1, D_1). Note that maxflow value 01 decreases by D units whenever a bandwidth demand of D units is routed between S_1 and D_1. Note that the value of v_1 will also decrease when an LSP is routed between some other ingress-egress pair. We define the amount of interference on a particular ingress- egress pair say, (S1, D1) due to routing an LSP between some other ingress-egress pair as the decrease in the value of v1.
2. Minimum Interference Paths: With interference defined as above, we can think of a minimum interference path for an LSP between say, (S1, D1) as that explicit route which maximizes the minimum maxflow between all other ingress-egress pairs. Intuitively, this can be thought of as a choice of path between (S_1, D_1) that maximizes the minimum open capacity between every other ingress-egress pair. Though this formulation has intuitive appeal it has the draw- back that it is the minimal maxflow that impacts the routing irrespective of values of the other maxflows. Hence, another objective might be to pick a path that maximizes a weighted sum of the maxflows between every other ingress-egress pair. We formalize these notions in the next section.
3. Critical Links: To progress from the notion of minimum interference paths to a viable routing algorithm that uses familiar max-flow and shortest path algorithms we need the notion of "critical links". These are links with the property that whenever an LSP is routed over those links the maxflow values of one or more ingress-egress pairs decreases. The next section gives an algorithm to determine critical links with a reasonable amount of computation (i.e., remains within a small target time budget for networks with hundreds of routers and thousands of ingress- egress pairs).
4. Path Selection by Shortest Path Computation: Once the critical links are identified, we would like to avoid routing LSPs on critical links to the extent possible. Also, we would like to utilize the well-used

Dijkstra or Bellman-Ford algorithms to compute the actual explicit route. We do this by generating a weighted graph where the critical links have weights that are an increasing function of their "criticality" (see next section for the actual link weight functions). The increasing weight function is picked to defer loading of critical links whenever possible. The actual explicit route is calculated using a shortest path computation as in other routing schemes.

1.6 TEST NETWORK FOR ALL ALGORITHMS

The ingress-egress pairs are shown in the figure. For the illustrative example below the size of the light links is 12 units and the dark link is 48 units (taken to model the capacity ratio of OC-12 and OC-48 links) and each link is bidirectional (i.e., acts like two unidirectional links of that capacity). For subsequent performance studies, we use a different network obtained as follows. We scale all capacities by 100. This larger capacity network is used for the performance studies because this permits us to experiment with thousands of LSP set-ups. The LSP bandwidth demands are taken to be uniformly distributed between 1 and 4 units.

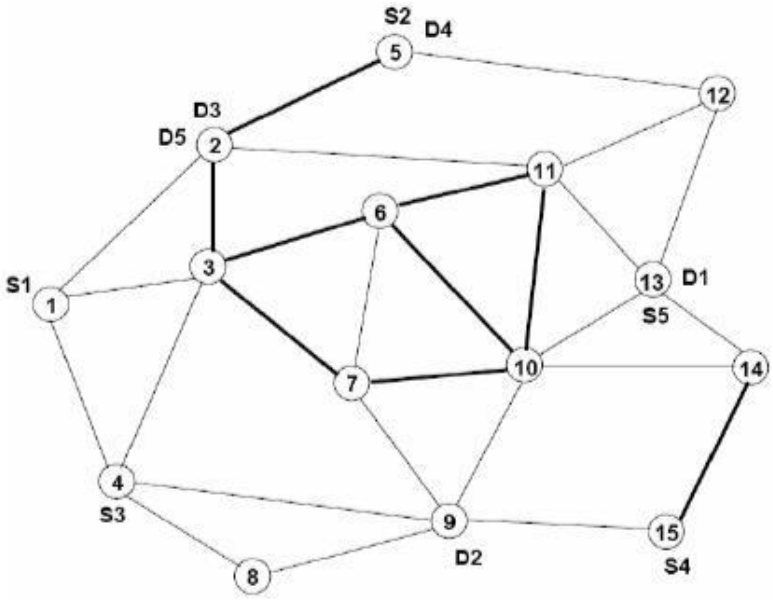

Fig 1.7 Network Under Consideration

A single demand instance which contains a set of demands between source-destination pairs, is used, again for all upcoming algorithms, to compare the performance against a defined test instance. The demand instance is given in the appendix.

1.7 ROUTING ALGORITHMS TABLE

1	Minimum Interference Routing Algorithm (MIRA)
2	Minimum Hop Algorithm(MHA)
3	Improved Minimum Interference Routing Algorithm(using Link Weights)(IMIRA)
4	Minimum Interference Hybrid Algorithm(MIH)
5	Least Interference Optimization Algorithm(LIOA)
6	Improved Least Interference Routing Algorithm(ILIOA)
7	Bandwidth Constrained Routing Algorithm(BCRA)
8	Residual Network and Link Capacity Routing Algorithm(RNLC)
9	Max RC Min F(RCF)
10	New Hybrid Minimum Interference(NHMI)

11	Minimum Interference Routing based on Lexicographic Max Flow(LEX-MAX)
12	Shortest Widest Path(SWP)
13	Widest Shortest Path(WSP)

Table 1.1 Routing Algorithm Table

1.8 COMMON METRICS FOR COMPARISON

The below set of common metrics/factors shall be used for all routing algorithms to compare performance and efficiency.

1. Flow Decrease on Link: We shall see the flow decrease that is happening on each link with the usage of a particular algorithm.
2. Rejection Count: We shall see which Ingress-Egress Router Pairs, are facing the most rejections
3. Link Usage: We shall see the frequency of calling/usage of each link. This shall not include the demand bits. It will try to only depict the frequency of usage. We will also see, which Ingress-Egress pair is frequently using which link.
4. Path Length: For every Ingress-Egress pair, we shall see the average path length.
5. Max Flow Decrease: For every Ingress-Egress pair, we shall track the maxflow decrease

CHAPTER 2
ALGORITHMS

2.1 MINIMUM INTERFERENCE ROUTING ALGORITHM(MIRA)

Salient Features and Notes on MIRA:

The value of $_{sd}$ can be chosen to reflect the importance of the ingress-egress pair (s, d).
If the value of $_{sd}$ = 1 for all (s, d) then the w(l) represents the number of ingress-egress pairs for which link is critical.
The weights can be made inversely proportional to the maxflow values, i.e, $_{sd}$ = 1/ $_{sd}$ where sd is the maximum flow value for the ingress-egress pair (s, d). **This weighting implies that the critical arcs for the ingress- egress pairs with lower maximum flow values will be weighted heavier than the ones for which the maxflow value is higher.**

MINIMUM INTERFERENCE ROUTING ALGORITHM ($\mathcal{MIRA}$)

INPUT:
A graph $G(N, L)$ and a set B of all residual link capacities
An ingress node a and an egress node b between which a flow of D units have to routed.

OUTPUT:
A route between a and b having a capacity of D units of bandwidth.

ALGORITHM:
1. Compute the maximum flow values for all $(s, d) \in \mathcal{P} \backslash (a, b)$.
2. Compute the set of critical links C_{sd} for all $(s, d) \in \mathcal{P} \backslash (a, b)$.
3. Compute the weights $w(l) = \sum_{(s,d):l \in C_{sd}} \alpha_{sd} \quad \forall l \in L$.
4. Eliminate all links which have residual bandwidth less than D and form a reduced network.
5. Using Dijkstra's algorithm compute shortest path in reduced network using $w(l)$ as the weight on link l.
6. Route the demand of D units from a to b along this shortest path and update the residual capacities.

1. Flow Decrease on Link

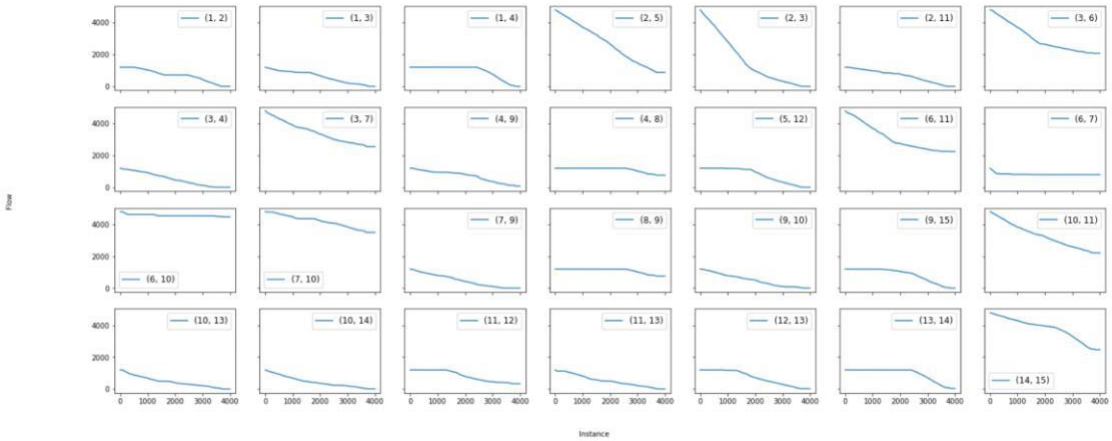

Fig 2.1 Flow decrease on Links for MIRA

We see that flow decreases in a varied format in MIRA with flow values of certain links quickly decreasing (for eg(2,3), (2,5), (7,9)), some link that don't decrease after a certain point((8,11),(10,11) and some links that are rarely used((3,7),(4,8))

2. Rejection Count

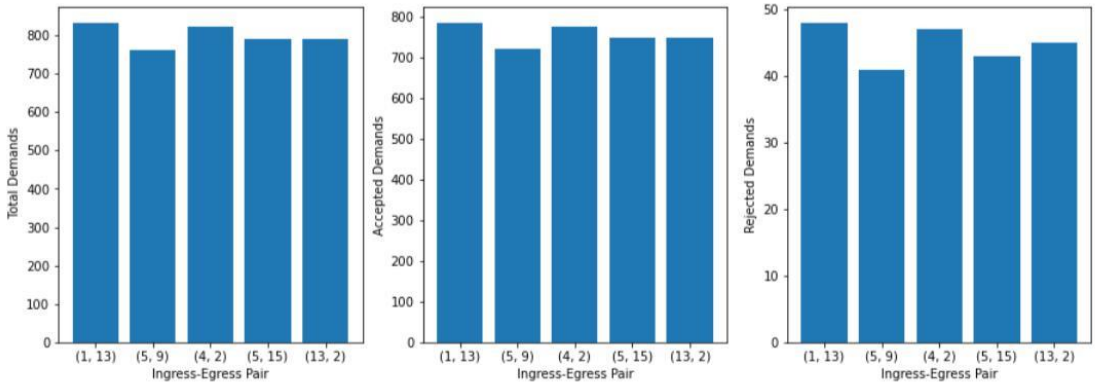

Fig 2.2 Rejection Count for MIRA

All IEPs have similar amount of rejections.

3.Link Usage

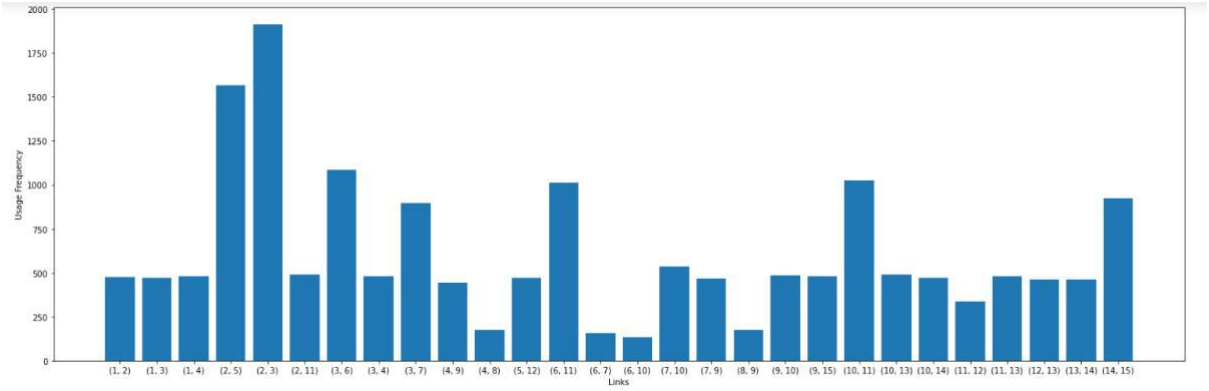

Fig 2.3 Link Usage for MIRA

We see certain links like (2,5) and (2,3) being heavily used when compared with other links. At the same time, links like (6,7) and (6,10) are negligibly used.

Link Usage by IEP

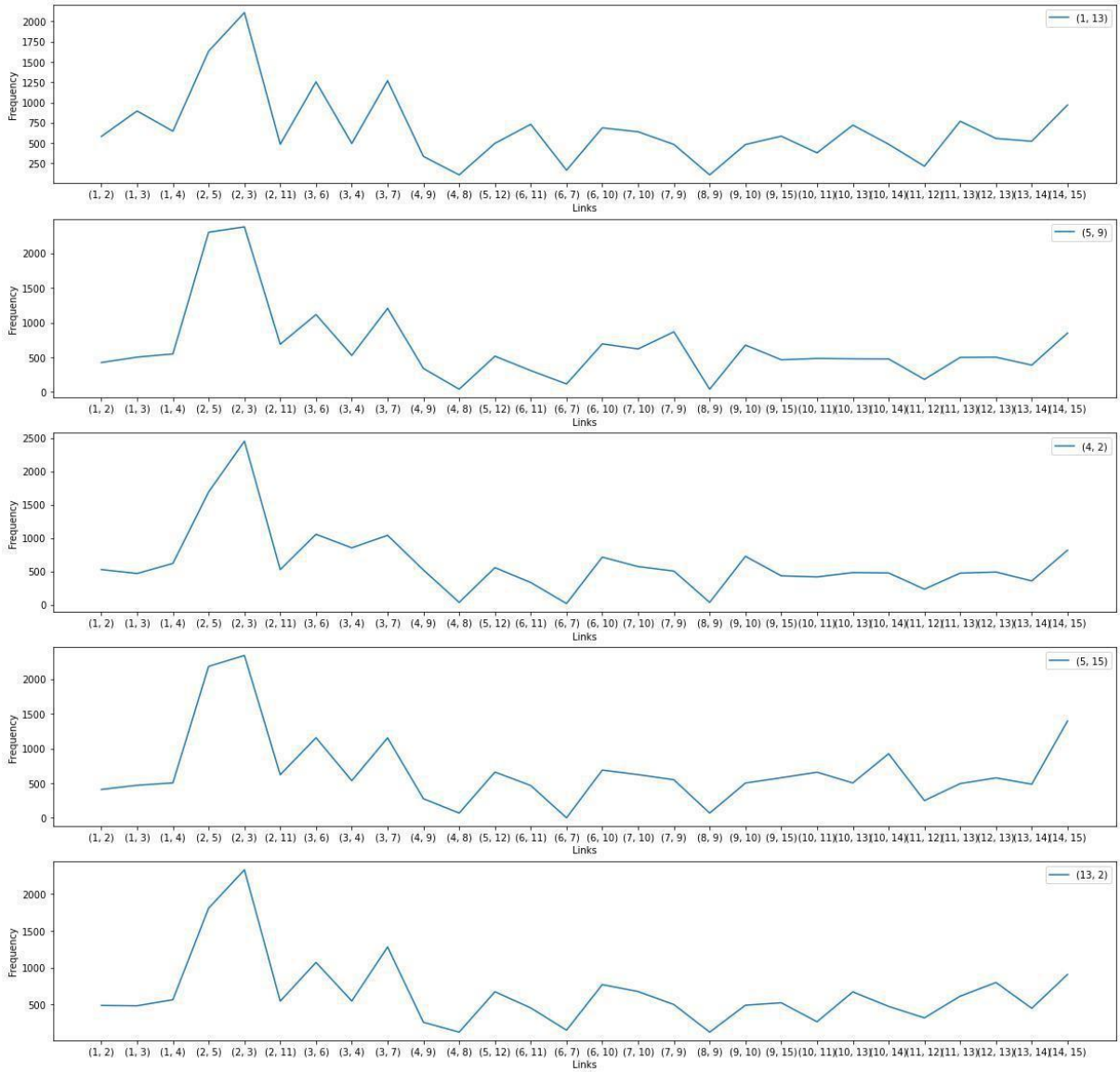

Fig 2.4 Link Usage by IEP for MIRA

We see a usage of certain links irrespective of the IEP, for eg: (2,5). Certain links like (7,9) are used only by a specific IEP i.e (5,9).

4.Path Length

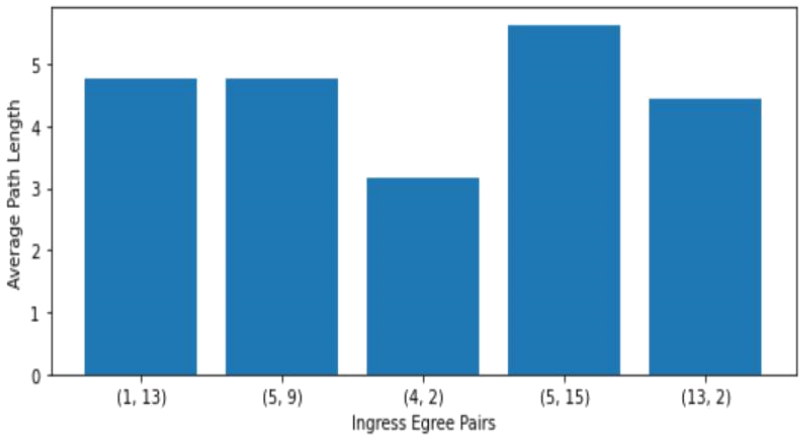

Fig 2.5 Average Path Length for MIRA

Almost all IEP have the same path avg. path length around 4.5. However, IEP (4,2) has a much lesser path length =3.

5.Max Flow Decrease

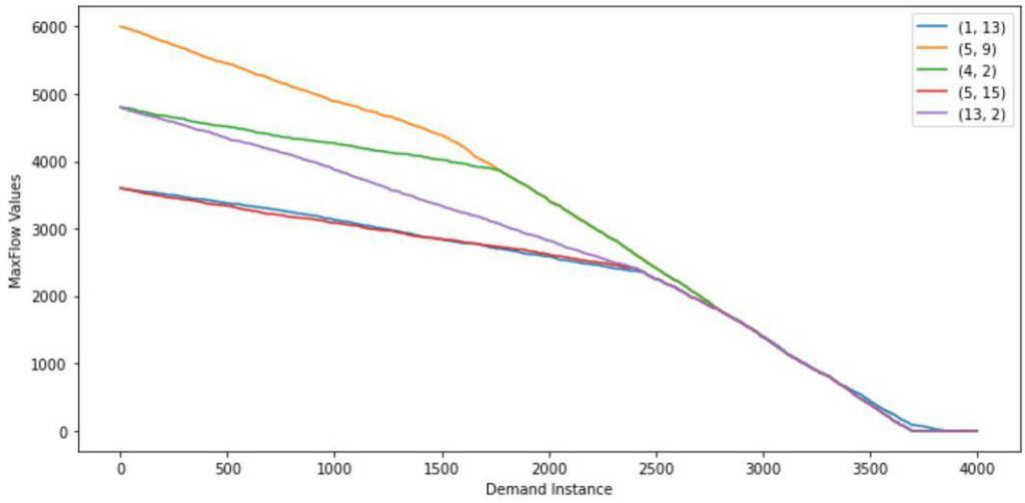

Fig 2.6 Max Flow Decrease for MIRA

We see that certain IEPs like (5,9) gradually lose their maxflow from the start, whereas others take a turn at around 2500 where their max flow values steeply reduce.

2.2 MINIMUM HOP ALGORITHM(MHA)

Salient features and Notes on MHA:

MHA is a traditional algorithm that has been used for a very long time. It does not aim to perform any specific objective(eg, load balancing) etc but aims to always wishes to find the shortest path(that has sufficient residual bandwidth to route the demand) so that, the lesser interactions with routers(i.e hops), the lesser router/node processing time.

INPUT:

A graph G(N, L) and a set B of all residual link capacities An ingress node a and an egress node b between which a flow of D units have to routed.

OUTPUT:

A route between a and b having a capacity of D units of bandwidth.

ALGORITHM:

1. Set the weights of all links =1.

2. Eliminate all links which have residual bandwidth less than D and form a reduced network.

3. Using Dijkstra's algorithm compute shortest path in reduced network.

4. Route the demand of D units from a to b along this shortest path and update the residual capacities.

1.Flow Decrease on Link

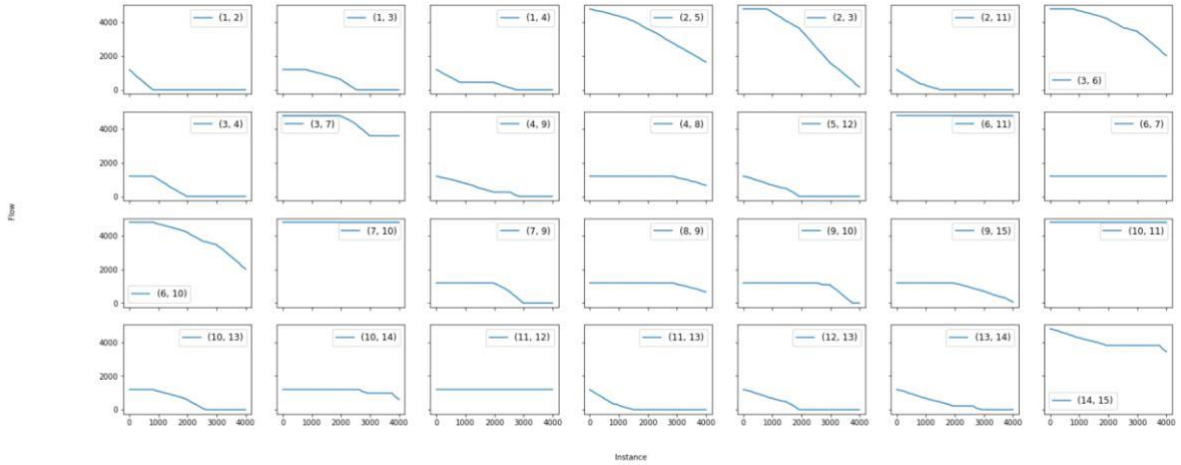

Fig 2.7 Flow Decrease on Link for MHA

We observe that certain links are negligibly or never used(for eg: (7,10), (6,7) etc). On the other hand we al see some rapid decline in residual bandwidth reaching 0 in some cases(for eg (1,2),(11,13)).
Links like (9,15) and (10,14) start to be used only around 2000 or 3000 when other links have exhausted their residual capacities.

2.Rejection Count

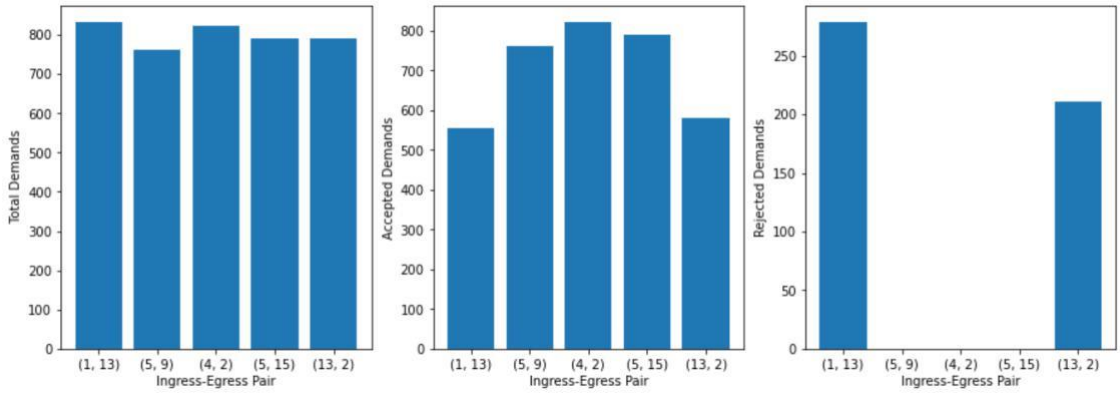

Fig 2.8 Rejection Count for MHA

We observe that although IEPs (5,9),(4,2),(5,15) have no rejections, it is at the cost of IEPs (1,13) and (13,2) with close to 200 rejections.

3.Link Usage

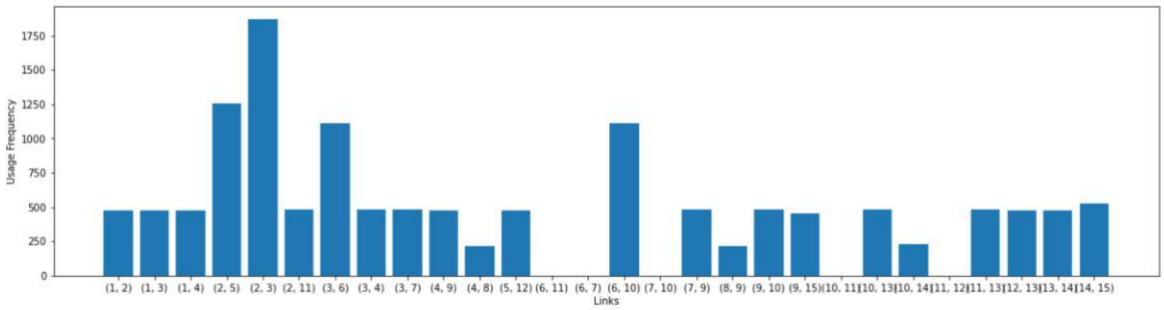

Fig 2.9 Link Usage for MHA

We again observe that certain links like (6,11) and (6,7) are not being used. Also, links like(4,8), (10,14), are being used in just 200 out of 4000 instances, whereas links like (2,5),(2,3),(3,6) are used in almost 1000 out 4000 instances

Link Usage by IEP

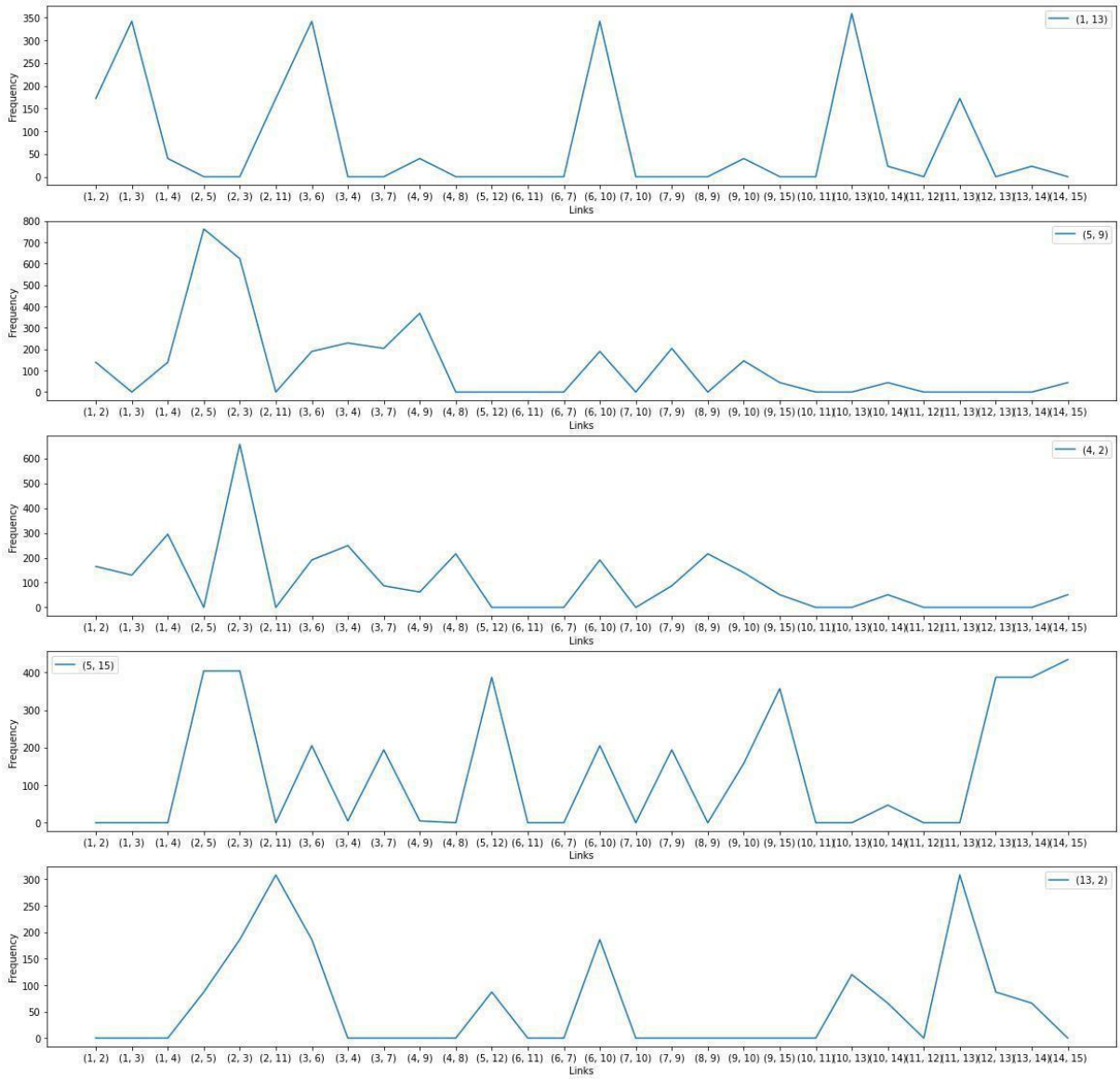

Fig 2.10 Link Usage by IEP for MHA

A lot of bias can be seen in choosing of links by IEPs (5,9) and (4,2). This could be, since MHA always chooses the shortest route, and these links might lie on those routes.

4.Average Path Length

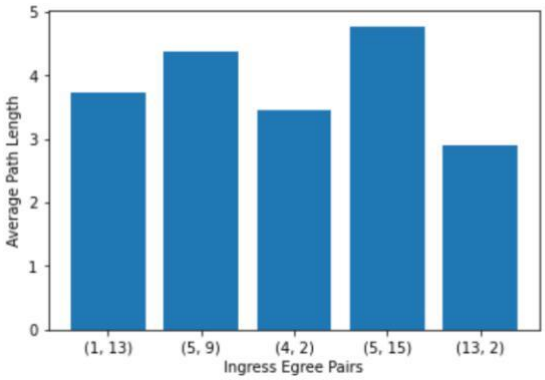

Fig 2.11 Average Path Length for MHA

All have a similar average path length averaging around 3.5

5.Max flow Decrease

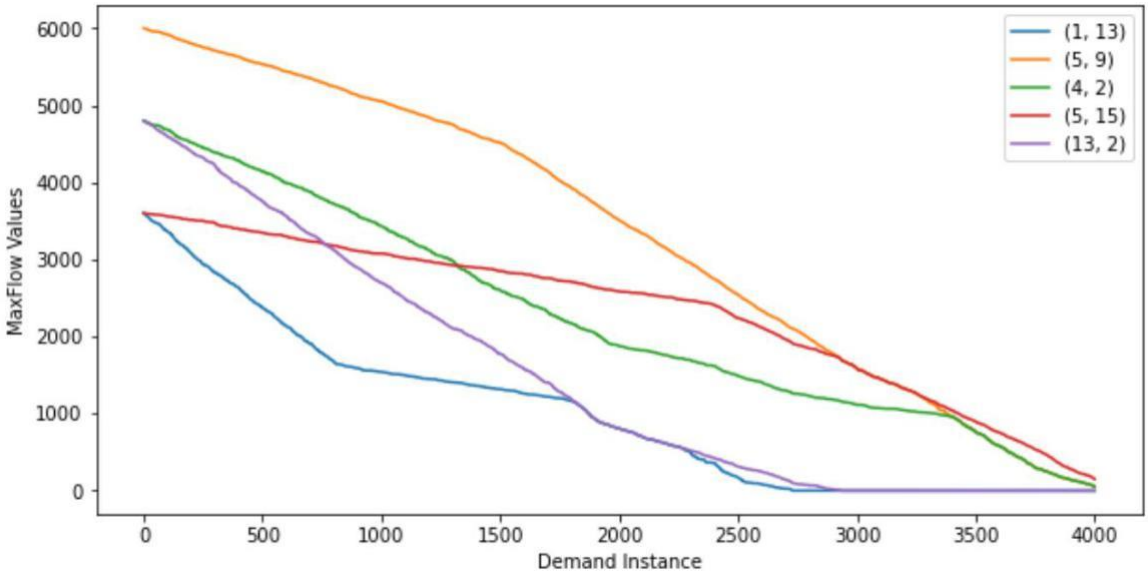

Fig 2.12 Max Flow Decrease for MHA

IEPs like (1,13),(13,2),(4,2) have a decrease from the start, with a sharp drop at 2000. Only IEPs like (5,9) and (5,15) seem to not decrease at the same rate as other IEPs.

2.3 IMPROVED MINIMUM INTERFERENCE ROUTING ALGORITHM

Salient features and Notes on IMIRA:

The minimum interference routing algorithm (MIRA) was designed to reduce the "interference" of routing current LSP request to potential unknown future requests. The basic observation is that routing an LSP along a path can reduce the maximum available bandwidth between some other ingress-egress router pairs. This phenomenon is termed as "interference." If paths that reduce a large amount of available bandwidth between other ingress-egress pairs are avoided, creation of bottlenecks can also be avoided. This routing problem is translated to finding the shortest path in a weighted graph, in which link weights are assigned according to their criticality

Shortcomings of MIRA:

Consider another example network topology with a bottleneck link shown in below figure. There are three potential source destination pairs: (S1, D1), (S2, D_2) and (S3, D3). Assume that all links have a residual bandwidth of 1 unit. Suppose that an online sequence of 3 requests arrives in the following order (S3, D3, 1), (S2, D2, 1), and (S1, D1, 1). According to the MIRA algorithm, link {7,8} is a critical link with respect to ingress-egress pairs (S1, D1) and (S2, D_2). When routing request (S3, D3, 1), the MIRA algorithm will try to avoid using critical links. Therefore, it will choose the four-hop route {1, 2, 3, 4, 5}. If there were more links between node 3 and 4, the longer route would be {1, 2, 3, · · ·, 4, 5}. The MIRA algorithm would still choose the longer route instead of the shorter one {1, 7, 8, 5}. If link {7, 8} has a residual bandwidth of 2 units instead of 1 unit, according to MIRA algorithm, link {7, 8} is not a critical link for any individual ingress-egress pair. So the MIRA algorithm will choose the path {1, 7, 8, 5} to route the request (S3, D3, 1), thus blocking request (S1, D1). Ideally, the first request should be routed along the longer route between S3 and D3, allowing the two subsequent requests to be routed through link {7, 8}.

Notice that these examples appear to be special cases of network topologies, they are nonetheless not uncommon in real networks where dynamic network traffic constantly creates bottlenecks. **As a result, a good online routing algorithm needs to be able to adapt to both topology and traffic condition changes.**

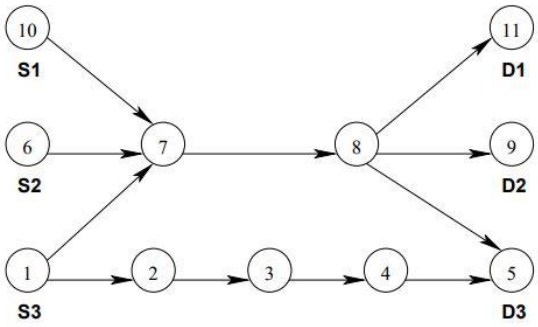

Fig 2.13 Sample Network

Improvements:

Similar to the MIRA algorithm, our algorithm takes into account the importance of critical links. In addition, we also consider links' residual bandwidth and path hop counts which were largely ignored in some previous work. The algorithm pseudo code is listed in Figure 3. Specifically, given an LSP request (, ,) to be routed, we consider the impact of routing this LSP request on future LSP set-up requests originating from the set of ingress and egress router pairs . The impact is characterized by assigning weights on links that may be used by future LSP requests between a pair of ingress and egress routers. When considering an ingress and egress pair (′, ′) ∈ , we first compute the maximum network flow, ′′ , between ′ and ′. Routing of the current LSP request between and will affect or reduce the maximum network flow between ′ and ′. For each link ∈ , we then calculate the link's bandwidth contribution to the maximum network flow between ′ and ′, which is represented as ′′ , where ′′ is the amount of flow passing through link . The

rationale behind is to characterize the relative importance of a link to future LSP requests between ′ and ′. Furthermore, we also need to consider the residual bandwidth on link so as to incorporate its capability of routing future LSP requests. We do this by calculating the normalized bandwidth contribution of link as follows,

′′ . ( ) − (1)

After obtaining the impact of routing the current request to future LSP requests in terms of weight on individual link as given by Eq. (1), we assign the overall weight of a link , ( ), to be the sum of individual weight contributed by all ingressegress pairs (′, ′) ∈ that can potentially originate LSP requests in the future. The individual weight of a link is calculated by Eq. (1) and the weight of link is obtained as follows,

′

0=Σ (′, ′)∈ ¨·0 .∈

Clearly the new algorithm leads to improved performance and hence provides better overall network resource utilization. The algorithm considers not only the importance of critical links, but also their relative importance to routing possible future LSP set-up requests. Moreover, link residual bandwidth information is also incorporated.

INPUT: G(V, E, B), L, an LSP request (s, d, b).

OUTPUT: A route between s and d having a capacity of b units of bandwidth.

ALGORITHM

1. Compute the maximum network flow values for all (s', d') = L

2. Compute the weight w(1) for all I Є E according to Eq. (2); 3. Eliminate all links that have residual bandwidth less than b and form a reduced network topology with remaining links and nodes

4. Using Dijkstra's algorithm to compute the shortest path in the reduced network using w(l) as the weight on link l

5. Route the bandwidth requirement of b units from s to d along this shortest path and update the residual link capacities

1.Flow Decrease on Link

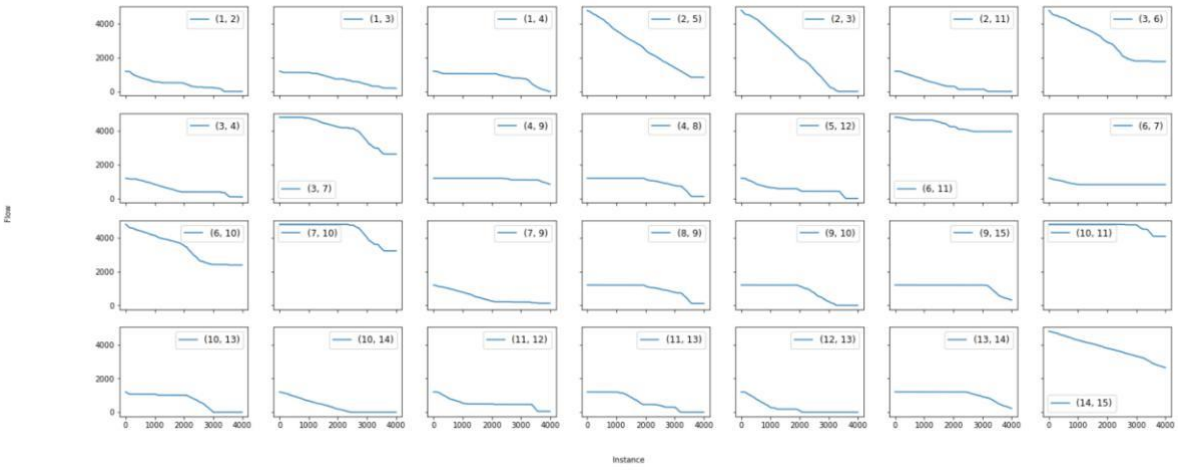

Fig 2.14 Flow Decrease on Links for IMIRA

Links like (6,7),(10,11) are negligibly used, whereas links like (2,5) and (2,3) are completely used.

2.Rejection Count

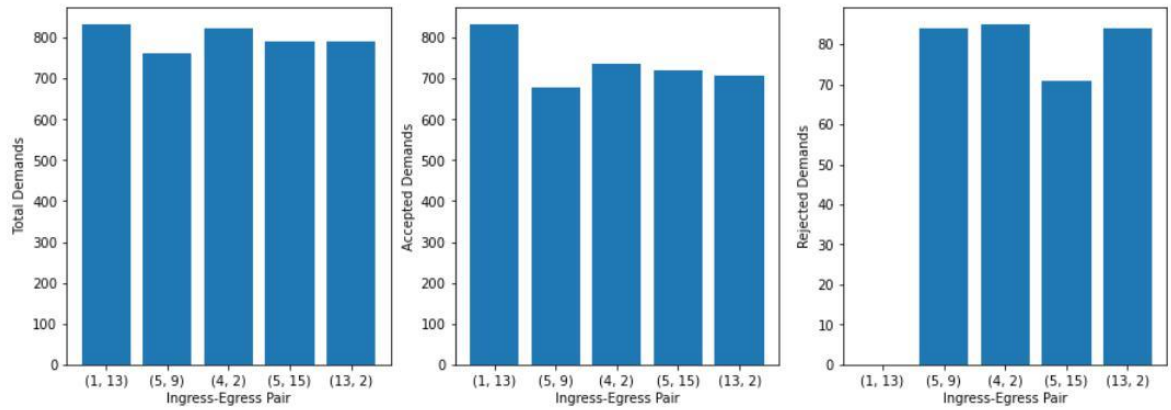

Fig 2.15 Rejection Count for IMIRA

(1,13) has 0 rejections. Rest have an avg of 70 rejections

3.Link Usage

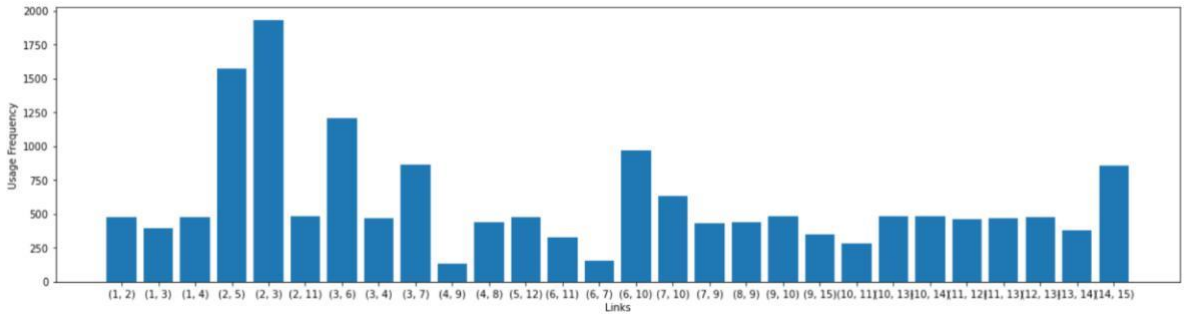

Fig 2.16 Link Usage for IMIRA

Links like (6,7) and (4,9) are not used much, whereas again, (2,3),(2,5) are used heavily.

Link Usage by IEP

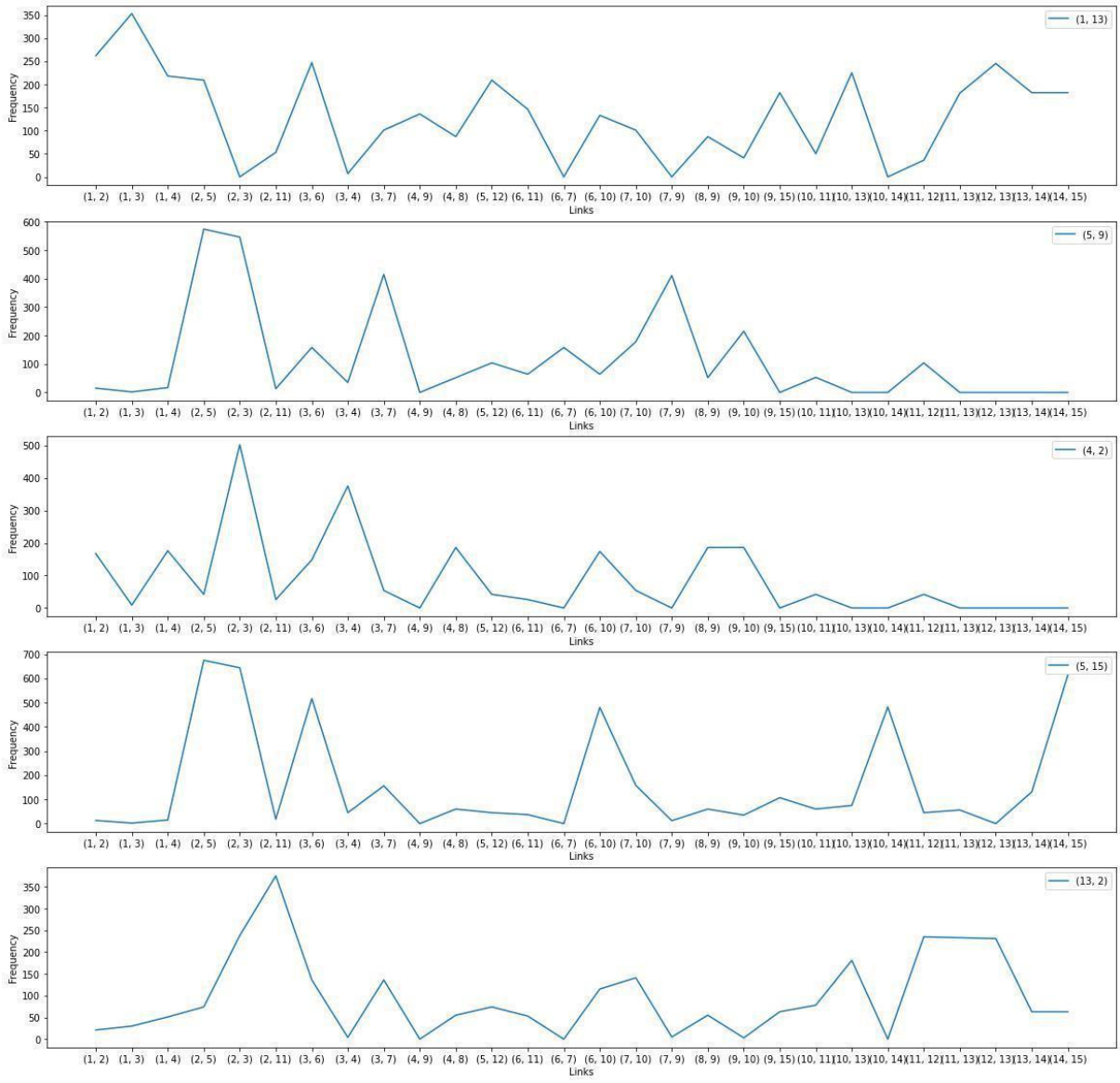

Fig 2.17 Link Usage by IEP for IMIRA

Links like (6,7) are used by specific IEPs (5,9). Otherwise these links are negligibly used. 4.Path Length

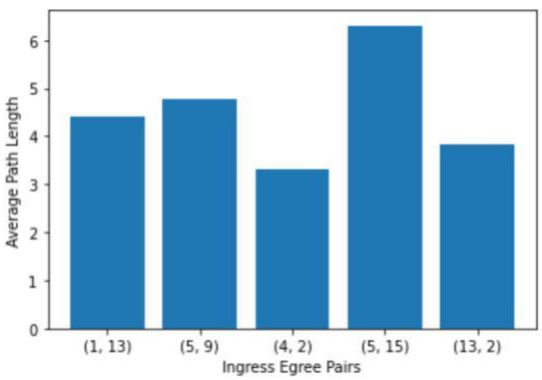

Fig 2.18 Average Path Length for IMIRA

IEP (5,15) has an avg path length of 6, while others have around 4.

5.Max Flow Decrease

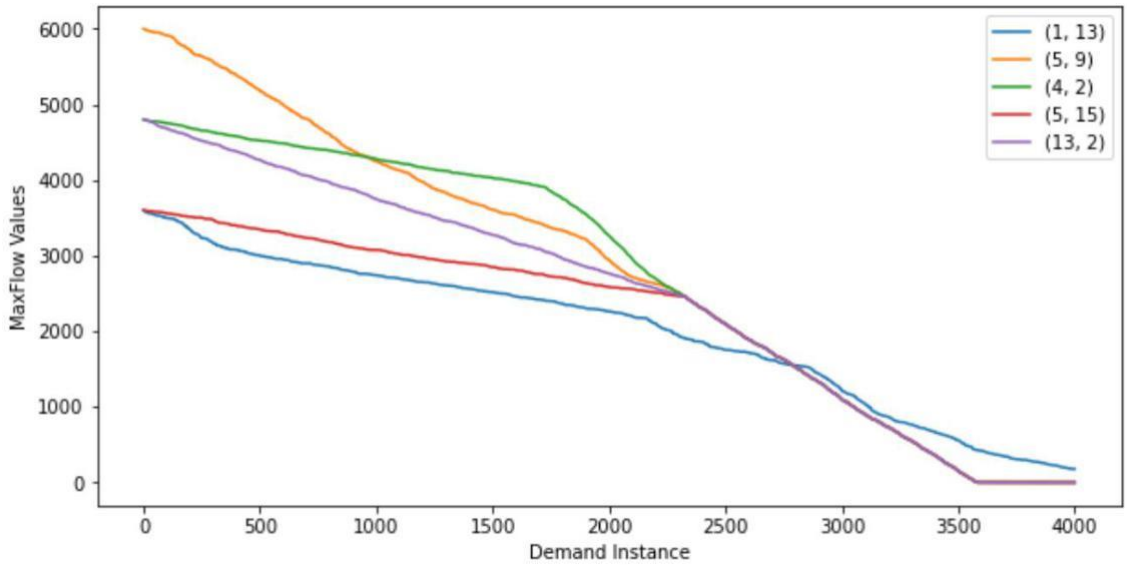

Fig 2.19 Max Flow Decrease for IMIRA We see that, unlike MHA, all IEPs don't lose their flow until instance 2500.

2.4 MINIMUM INTERFERENCE HYBRID ALGORITHM(MIH)

Salient features and notes on MIH:

The main goal of MIRA is to reduce interference. In order to achieve other equally important objectives, the MIRA algorithm was modified to a hybrid nature.

1. Minimizing Path Length: The simplest routing algorithm is the Minhop algorithm which finds paths with the minimum number of hops from the source to the destination. **Though this algorithm is capable of finding the shortest path that can support the requested bandwidth, it has some drawbacks. Shortest paths of multiple traffic streams may converge causing overloading of specific paths whereas slightly longer paths might go underutilized**. It may block future requests. The Widest Shortest Path algorithm uses the hop count as the metric and chooses the path with the minimum hop count that can satisfy the request. If there are more than one such paths, the algorithm chooses the one with the maximum bandwidth. **This algorithm optimizes the hop count and considers the bandwidth, but ignores the interference factor.**
2. Minimizing Load: The Shortest Widest Path algorithm uses bandwidth as the metric and chooses the path with the maximum bottleneck bandwidth. The bottleneck bandwidth of a path is the minimum residual bandwidth among all the links in the path. If more than one path have the same maximum bottleneck bandwidth, it chooses the one with the minimum number of hops. **The disadvantage of this algorithm is that it might choose lengthy paths in favor of less important shorter paths. So this algorithm optimizes only the link load metric and ignores other factors.**
3. Minimize Interference: The goal of the Minimum Interference Routing Algorithm (MIRA) is to accommodate as many requests as possible. The idea is to use those paths which do not interfere too much with future LSP demands between other source destination pairs. This algorithm assumes that there is some knowledge about the potential ingress-egress pairs. Knowledge about potential ingress-egress pairs helps in routing new traffic along paths which are not critical to future requests, thus reducing the number of request rejections.

Link Weight Definition:

For each link, let be its current load, be its total capacity and be the current maximum link utilization. Let be its criticality and be the maximum criticality in the network. We initially set , , and to zero and update them every time a demand is routed. Let be the latest demand. Then the link cost metric is defined as follows:

$$= \frac{}{}^{+} + \times \mathrm{m}[0, \frac{}{}^{+} -] + \times \frac{}{}$$

Here load 2 are tunable parameters.

The algorithm selects the routes as least cost routes based on this link cost metric. In the metric the first term is the link utilization after the new demand is added, the second term is the increase in the maximum of link utilization due to selecting this link and the third term is the normalized criticality of the link. The second term produces optimal routes in terms of minimizing the maximum utilization. The third term produces optimal routes in terms of minimizing the interference. The first term refines the link metric by making sure that unnecessarily long paths are not chosen to optimize utilization and interference. The parameters T1 and T2 can be tuned according to how much weight needs to be given to the three different factors. For example if T_1 is 10 and T2 is 5, then each unit increase in the maximum utilization is equivalent to ten times the increase in the utilization of the link and twice the increase in the criticality of the link.

INPUT: A graph G (N, L) and a set B of all residual link capacities. An ingress node a and an egress node b between which a flow of D units have to routed.

OUTPUT: A route between a and b having a capacity of D units of bandwidth.

ALGORITHM:

1. Find maximum flow values between all (s, d) Є P \ | (a, b).
2. Find the set of critical links for each pair.
3. Compute criticality g_{ij} of each link i-j.
4. Find maximum criticality ß in the network.
5. Compute utilization f_{ij} for each link i-j
6. Find maximum utilization a in the network.
7. Compute the weights as θ_{ij}
8. Eliminate links with residual bandwidth less that D.
9. Using Dijkstra's algorithm find shortest path between a and b based on the above link weights.
10. Route the demand of D units from a to b and update the residual capacities in C.

1.Flow Decrease On Links

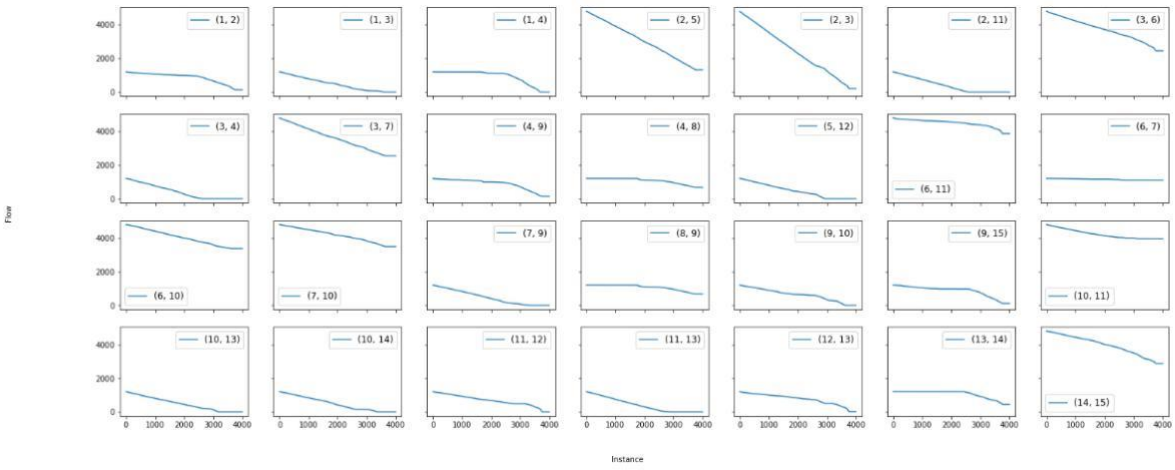

Fig 2.20 Flow Decrease on Links for MIH

We can see that almost all links are put to use except (6,7) that seems to be used negligibly. This could be because of the link utilization objective that aims to minimize link utilization or criticality and instead focus more on load balancing.

2.Rejection Count

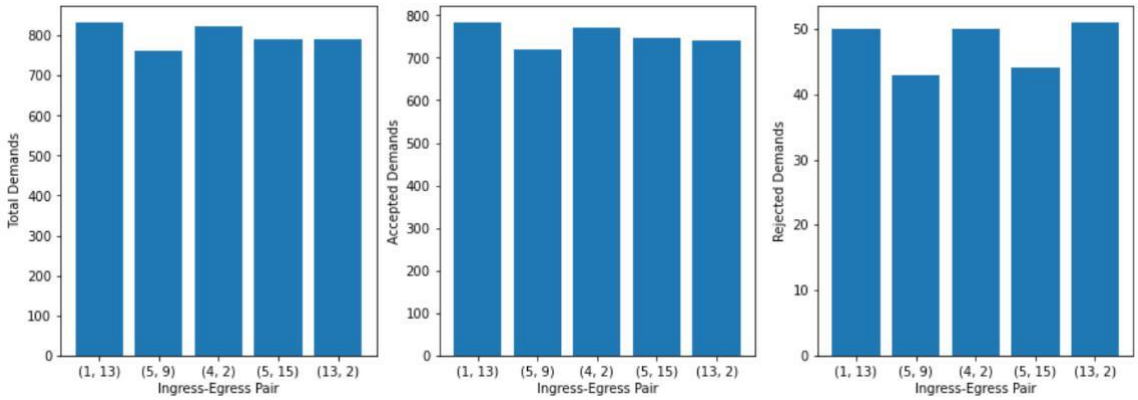

Fig 2.21 Rejection Count for MIH

3.Link Usage

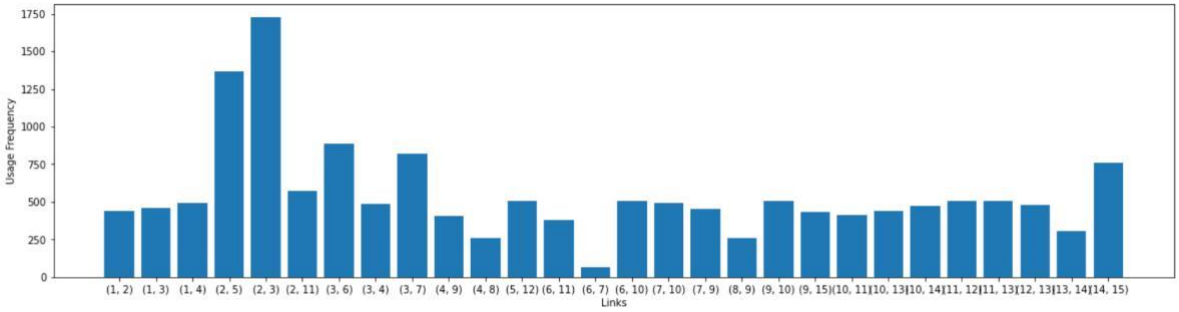

Fig 2.22 Link Usage for MIH

We see almost all links are used equally except certain links((2,5),(2,3))

Link Usage by IEP

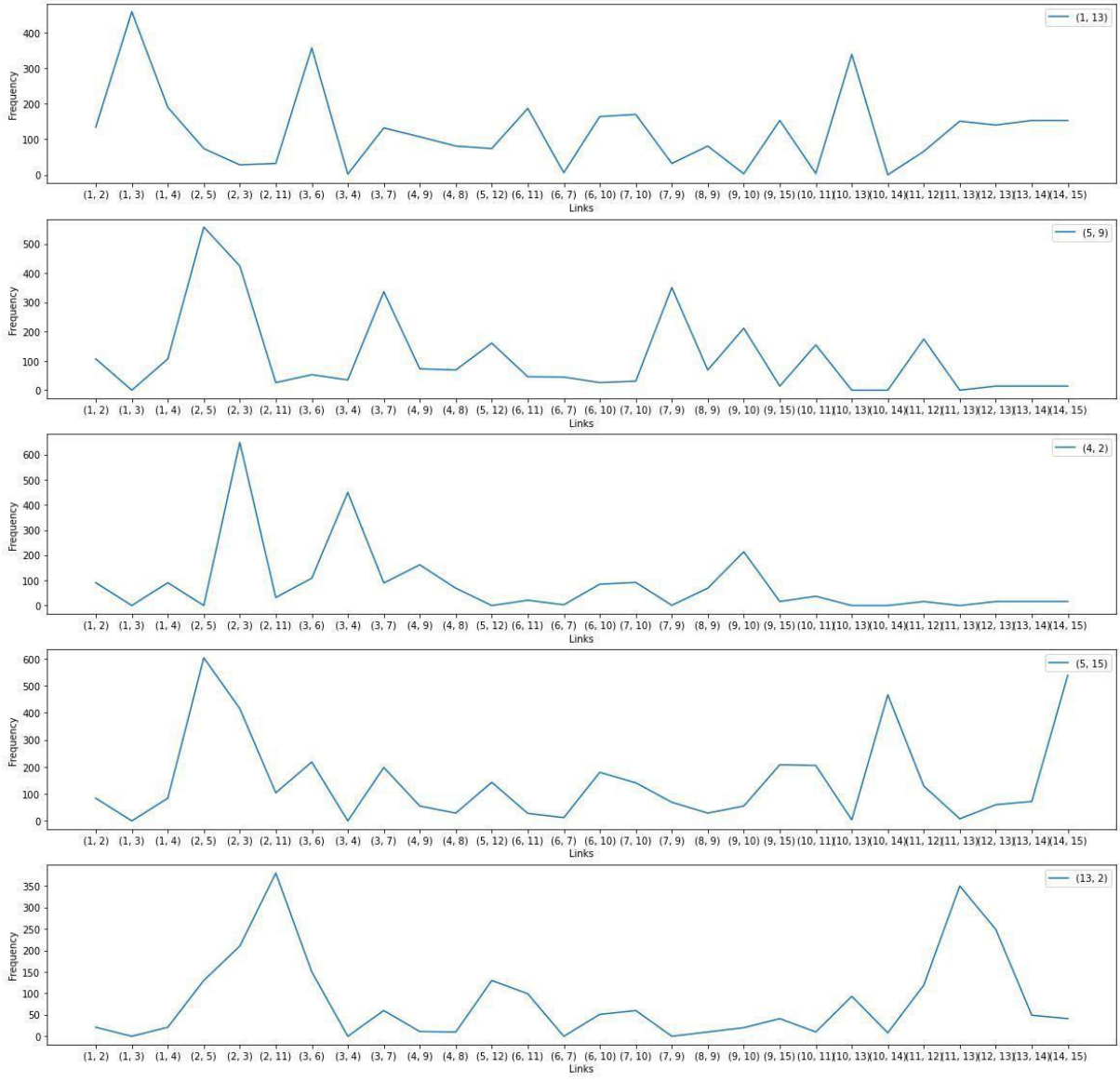

Fig 2.23 Link Usage by IEP for MIH

Other than the (1,3) peak, for IEP (1,13), we don't see any particular affinity of an IEP towards a link. Although, we see the peaks at link (2,3) and (2,5), this is observed for all IEPs. IEP (4,2) seems to not use links pertaining to nodes 11, 13, 14, 15, because these nodes don't lie on a reasonably short path.

4.Path Length

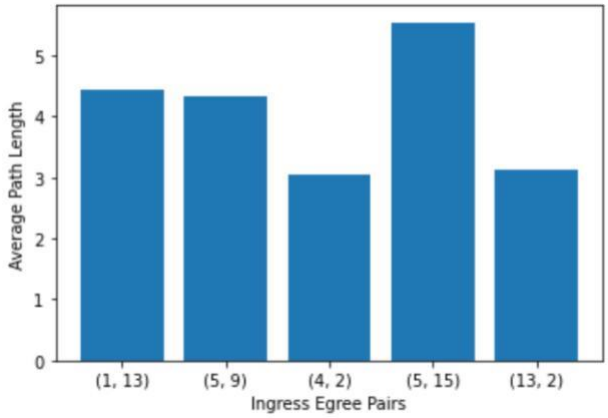

Fig 2.24 Average Path Length for MIH

(4,2) has lowest avg path length around 3.

5.Max Flow Decrease

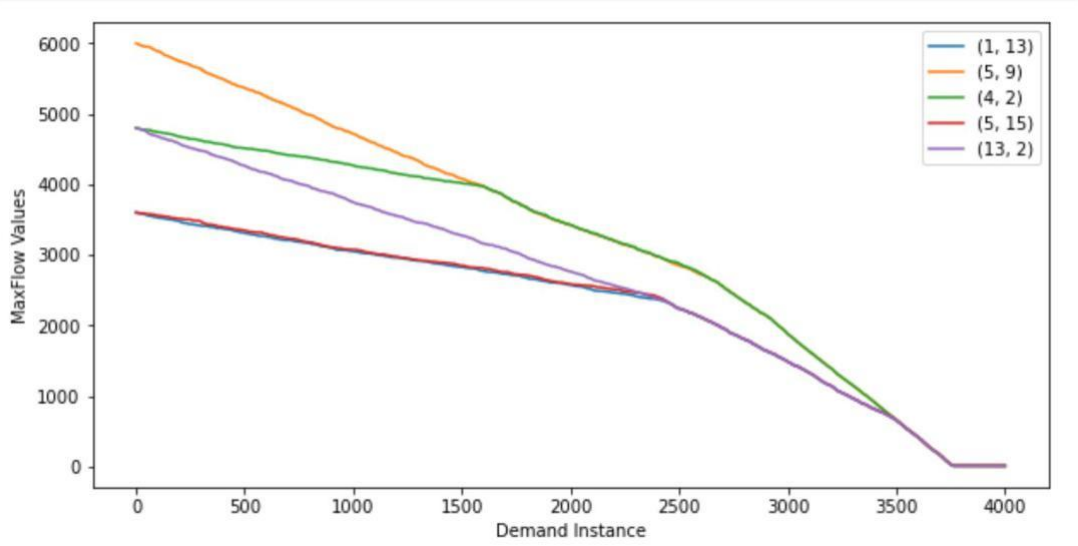

Fig 2.25 Max Flow Decrease for MIH

We are able to see, max flow values start decreasing rapidly at around instance 2500

Bonus: Experimenting with various values of T1 and T2

T1T2	0	1	2	3	4	5	6	7	8	9
0	167	246	270	260	238	238	238	245	245	245
1	166	244	254	244	251	250	250	250	234	234
2	165	237	254	263	261	262	262	262	262	262
3	167	249	256	262	256	252	252	252	252	252
4	168	241	255	251	242	236	236	236	236	236
5	168	246	270	248	269	260	265	257	257	257
6	166	245	255	261	262	259	259	254	254	254
7	169	246	254	250	271	265	265	262	262	262
8	167	249	272	249	238	244	244	244	249	249
9	172	243	262	263	256	252	242	242	240	250

Table 2.1 Rejection Count for Various Values of T1 and T2 in MIH Algorithm

2.5 LEAST INTERFERENCE OPTIMIZATION ALGORITHM(LIOA)

Salient features and Notes on LIOA:

Constraint Shortest Path First (CSPF) was proposed as a protocol which solves the problem of load balancing in OSPF by using link costs which reflect the current resource availability. These include costs which are **inversely proportional to the residual link capacities.**

Consider a link cost function

$$\ell(\ell, \ell) = \ell / \ \ell^{1-}$$

where the link interference ℓ is the number of flows carried on the link, the link slack $\ell = \ell - \ell$ where ℓ is the maximum reservable bandwidth of link ℓ, ℓ is the total bandwidth reserved by the LSPs traversing the link and $0 \leq \leq 1$ is a parameter representing a tradeoff between the number and the magnitude of the LSPs traversing a link.

The link cost function is minimized by minimizing the the link interference ℓ (IOPT routing) and maximizing the link slack ℓ (CSPF routing). By dispersing traffic flows over the network through interference minimization, this link cost will minimize the number of LSPs blocked under congestion and minimize the number of LSPs which must be re-routed under a single link failure. Through link slack maximization, the link load is kept far from a congestion region where the link load approaches the link capacity.

$$\ell(\ell, \ell) = \begin{cases} 1/(\ell - \ell) & = 0 \\ & = \ell = 0 \\ & = 1 \end{cases}$$

CSPF routing

OSPF routing

IOPT routing.

When $0 <$ < 1 the link cost function yields a mix of IOPT and CSPF routing.

Therefore:

In LIOA the interference is defined by the number of flows (LSPs) carried by a link. However, in the MIRA interference is defined as a reduction of the maximum flow between a pair of nodes after set up the LSP between another pair of nodes. In LIOA weights of links are proportional to the number of flows (the number of LSPs) realized on these links, and inversely proportional to their residual capacity. Dispersing traffic flows

(LSPs) in the network and minimizing the rejected LSP requests is obtained by minimizing interference and maximizing the residual capacity of the links

Consider a demand for , bandwidth units between two nodes and . LIOA executes four steps in routing this demand

1 Prune the network. Set (,) = ∞ for each link whose slack − ≤ , .
2 Find the new least cost path. Apply Dijkstra's algorithm to find the least cost path * ∈ , .
3 Route the traffic demand. Assign the traffic demand to path *.
4 Update the link reserved bandwidth, interference and costs. For each link ∈ *: := + ; := +1; (,)= /(−)$^{1-}$.

An I-E pair is said to have a path multiplicity of n if the pair is connected by n paths. It is probably an advantage for an I-E pair to have two paths rather than one path when traffic splitting is used to load balance the network or support network recovery. Having four paths rather than three is probably a disadvantage when multipath routing tends to find longer paths which may tie up more resources. LIOA finds relatively more single and dual paths than CSPF and MIRA. Hence, it can achieve more load balancing than MIRA and CSPF.

1.Flow Decrease on Links

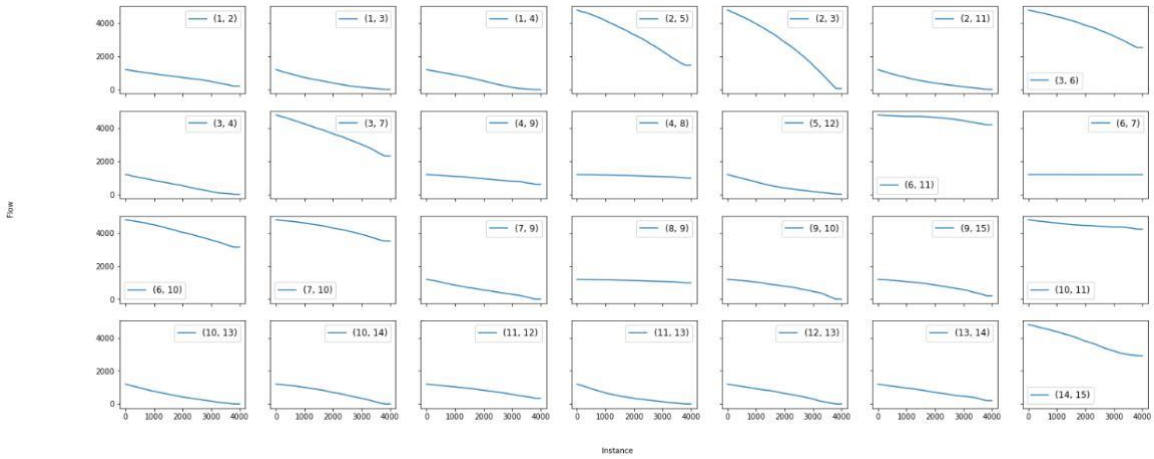

Fig 2.26 Flow Decrease on Links for LIOA

2.Rejection Count

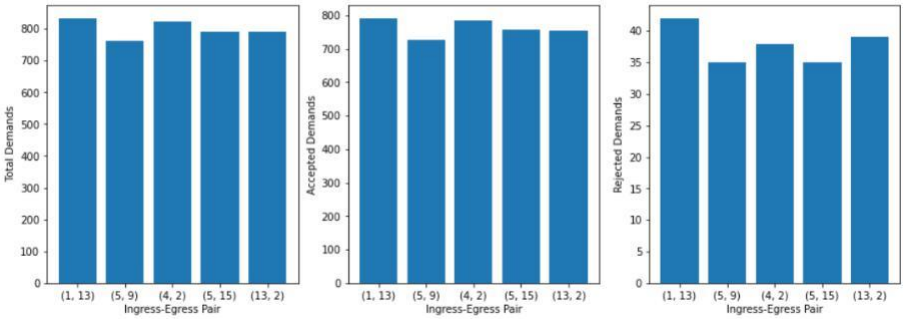

Fig 2.27 Rejection Count for LIOA

3.Link Usage

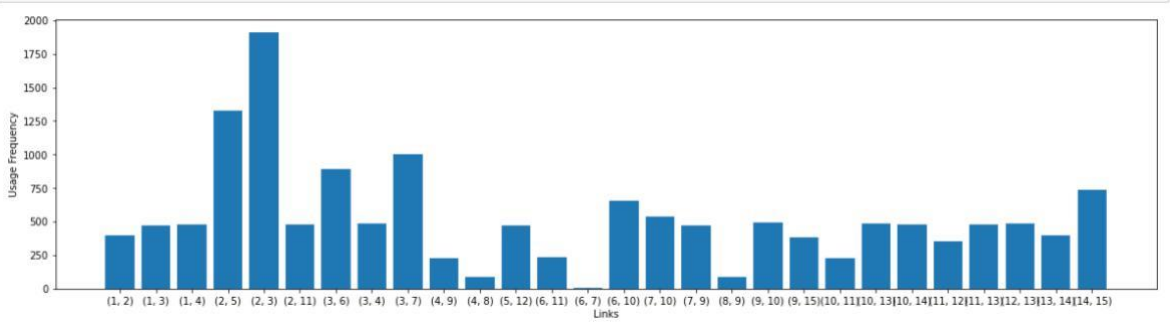

Fig 2.28 Link Usage for LIOA

LIOA works mainly between optimizing path lengths and interference objectives. It does not take into account, load balancing. Hence, we see a “slight” disparity in link utilization levels of certain links.

Link Usage by IEP

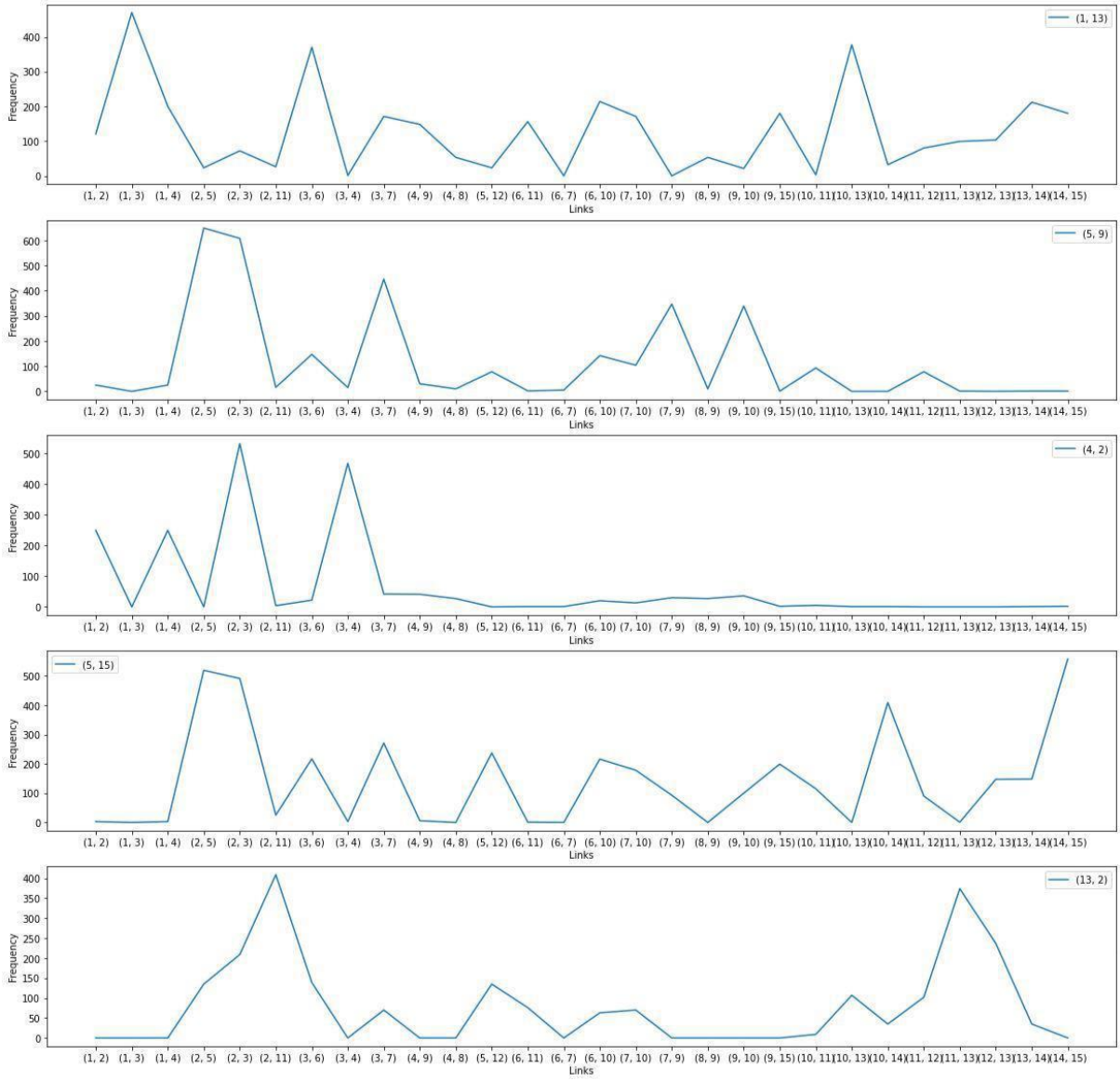

Fig 2.29 Link Usage by IEP for LIOA

We are seeing particular affinities of an IEP towards links. Some observations are : IEP(1,13) surprisingly does not use links connected to node 2. IEP(5,15) does not use links connected to node 4 and its main entry of

arrival into node 15 is via link(14,15) instead of link (9,15). IEP(4,2) uses links pertaining to nodes 1,2,3. It uses other links negligibly.

4.Path Length

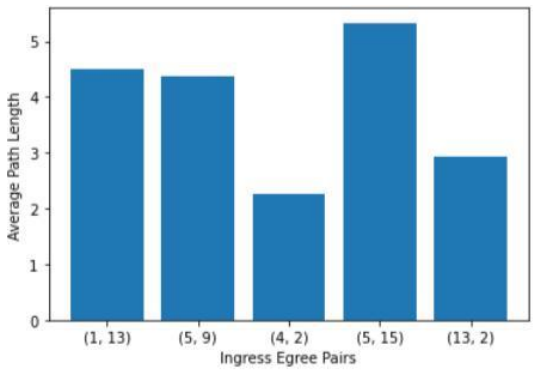

Fig 2.39 Average Path Length for LIOA

In MIH, the avg path length was 3 for IEP(4,2), but here its averaging around 2, primarily because LIOA tries to minimize path length.

5.Max Flow Decrease

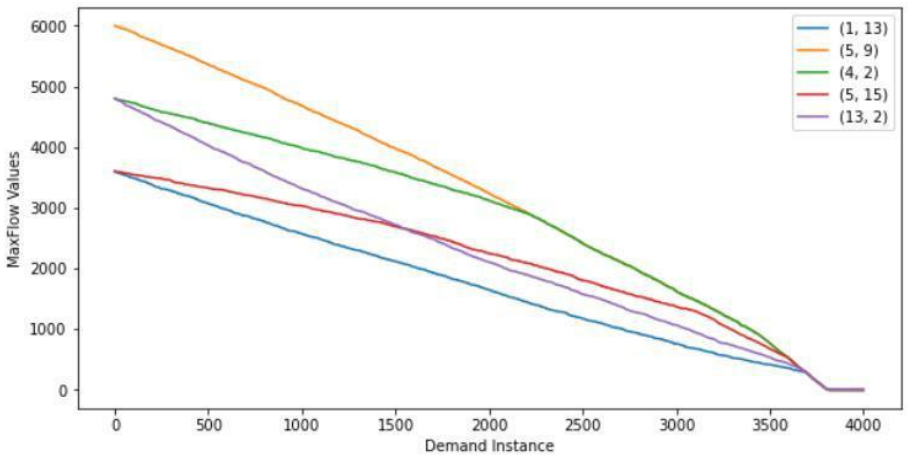

Fig 2.31 Max Flow Decrease for LIOA

Very uniquely from the previously observed algorithms, we see a linear decrease in maxflow upto instance 3900. There are no sharp drops around instance 2500.

Bonus: Trials with various values of

	Rejections
0.1	151
0.2	152
0.3	162
0.4	175
0.5	189
0.6	201

0.7	251
0.8	316
0.9	323

Table 2.2 LIOA Rejection Count for various values of

We see that, as approaches 1.0, the no. of rejections increases, since it starts following the IOPT methodology.

When = 0.1

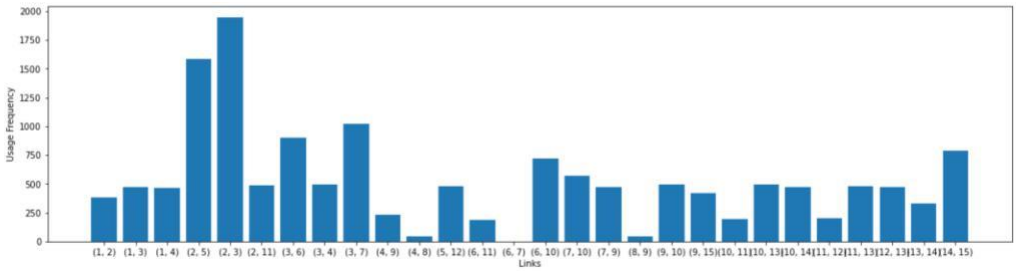

Fig 2.32 Link Usage for LIOA(= 0.1)

When = 0.5

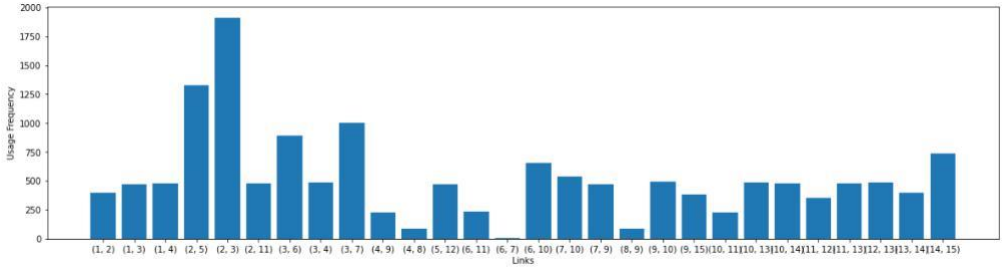

Fig 2.33 Link Usage for LIOA(= 0.5)

When = 0.9

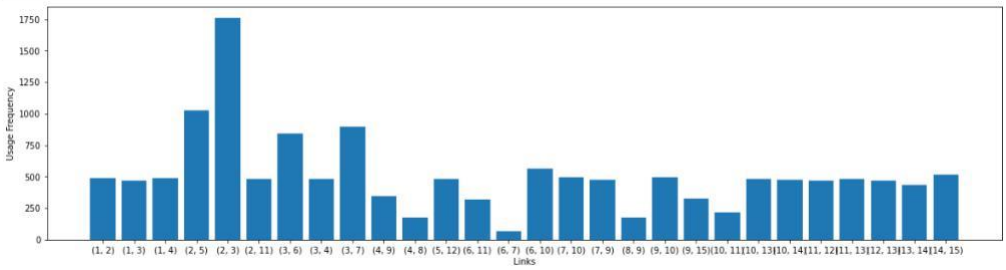

Fig 2.34 Link Usage for LIOA(= 0.9)

We are seeing more link usage at = 0.9, since it follows IOPT, it aims to minimize routing flows over links that have already carried many flows.

2.6 IMPROVED LEAST INTERFERENCE OPTIMIZATION ALGORITHM(ILIOA)

Salient Features and Notes on ILIOA:

Let (, ,) be the network, where is the set of nodes (routers) and is the set of unidirectional links (arcs). is -vector of bandwidth of the links. Let denote the number of the nodes and the number of links in the network. Let be a -element vector of the residual capacity of links. Moreover, let , be the set of all possible paths between a pair of nodes (,). In addition, let (, ,) denoted the weight of link . is the number of flows carried on the -th link (the size of interference). Suppose that between a pair of nodes (,) the request of LSP set up with bandwidth , appears. Informally, the problem of routing optimization can be formulated as follows: Find a path *, * ∈ , , with a minimum weight * = ∑ ∈ * (, ,) with a guaranteed bandwidth , , which minimizes the number of rejected requests of LSPs set up in the network.

This algorithm is similar to LIOA, but the weights of links are defined in a more sophisticated way.
Let consider the following link weight function:

(, ,)=(1−) ―― + ――
1− 1−

where: = 1 − / is the usage of link; in turn and are real coefficients from range [0,1].

It should be noticed that in the case of low usage of -th link, a decisive factor of a weight of link (, ,) will be: (1 −) / 1− . Weight of link, with fixed bandwidth, is minimized by minimizing interference, i.e. the number of carried flows. In order to realise the LSP in the network the links with low numbers of carried flows and large capacities will be chosen. However, in the case of high usage of link the decisive factor of the weight of a link (, ,) will be / 1− (where / 1− is weight of link in LIOA). Weight of link will be minimized by the number of flows and maximizing the residual capacity of the link. This fosters the dispersion of traffic in the network. In order to realise the LSP in the network the links with relatively low numbers of carried flows and large residual capacities will be chosen. The links load will be kept far from congestion area, so that the number of rejected requests of LSPs set up will be minimized.

INPUT:

Network $G(N, E, B)$, vector R of residual capacities of links; vector I of flows and pair of nodes (i, j), between which $d_{i,j}$ bandwidth units should be realized.

OUTPUT:

The path between pair of nodes (i, j) with $d_{i,j}$ bandwidth units.

ALGORITHM:

1 Compute the weights:

$$w_l(C_l, R_l, I_l) = (1 - u_l)\frac{I_l^{\beta}}{C_l^{1-\beta}} + u_l \frac{I_l^{\alpha}}{R_l^{1-\alpha}} \; \forall l \in E;$$

2 Eliminate all links $(w_l(C_l, R_l, I_l) = \infty)$, which have residual bandwidth $R_l \prec d_{i,j}$;

3 Using Dijkstra's algorithm compute shortest path p^* between pair of nodes (i, j).

4 Route $d_{i,j}$ bandwidth units from node i to node j along path p^*;

5 Update vector R and vector I; i.e. for each link $l \in p^*: R_l := R_l - d_{i,j}; I_l := I_l + 1$.

Results were quite similar to that obtained in

LIOA **Bonus: Trials with various values of and**

ALPHA/BETA	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0.1	151	160	330	261	370	464	451	550	565	596
0.2	151	151	188	297	352	445	457	584	569	596
0.3	151	152	174	263	258	364	445	624	561	583
0.4	151	151	152	189	275	339	441	558	644	620
0.5	152	152	151	163	197	340	378	488	559	595
0.6	152	152	151	155	173	212	353	399	462	583
0.7	152	153	153	153	160	181	265	279	447	577
0.8	157	160	153	157	160	168	199	329	346	524
0.9	168	167	165	169	167	172	179	219	313	386
1	198	206	204	201	202	193	201	222	280	320

Table 2.3 ILIOA Rejection Count with Various Values of and

2.7 BANDWIDTH CONSTRAINED ROUTING ALGORITHM(BCRA)

Salient features and Notes on BCRA:

In this paper the following objectives are considered: Distributing network load, Minimizing path length and Reducing path cost.

Distributing Network Load:

For distributing network load, a link load parameter called l_load ( ) is defined. The idea behind this parameter is that if there exists highly loaded links in a network then these links should be avoided during selection of paths for routing requests. –load( ) is defined as follows:

$$\text{–load}(\ ) = \frac{\text{reserved bandwidth on}}{\text{total reservable bandwidth on}}$$

The mean link load throughout the network is defined as:

$$\text{load}_{\ } = \frac{1}{|\ |} \sum_{\ } \text{load}_{\ }$$

A critical link is defined as a link whose load is running above the threshold –load ( ). A critical path is one which has critical component links. The higher the number of critical component links on a path is, the more critical it is.

To distribute path loading, we consider that each link belonging to the graph has a TE metric that we called TEML(i) as follows:

$$\text{TEML}(\ ) = \quad (\ \text{–load}(\ ))$$

Where is a positive objective function that will be formulated later.

The TE metric for a path , called TEMP, is defined as:

$$= \sum_{\in \cap} (\ ).$$

Minimizing path cost

In this context, each link has a cost that depends on its bandwidth capacity (–capacity ( )). Consequently, the link cost is static and attributed or modified only by the network administrator. We denote –cos ( ) the cost of link .

$$\text{–}^{\cos}(\ ) = \text{–capacity}(\ ) * 10^{8}.$$

Minimizing path length:

To minimize path length, CSPF Hop Count assigns a weight equal to 1 for each link. Using this fact to compromise between minimizing path cost, reducing path length and distributing network load, the objective function as follows:

$$(\ -\text{load}(\ )) = -\cos(\ ) * -\text{load}(\ ) + 1.$$

Finally, the TE metric of link is reformulated as follows:

$$\text{TEML}(\ ) = -\cos(\ ) * -\text{load}(\ ) + 1.$$

INPUT:

G(N,L), an LSP request r(s,d,B)

OUTPUT: A route between s and d having a capacity of B units of bandwidth

ALGORITHM

1. Compute the weight TEML(i) for all i in L according to above formula

2. Eliminate all links that have residual bandwidth less than B and form a reduced network topology with remaining links and nodes.

3. Using Dijkstra algorithm to compute the shortest path in the reduced network using TEML(i) as weight of link.

4. Route the bandwidth requirement of B units from s to d along this shortest path and update the residual link capacity.

1.Flow Decrease on Links

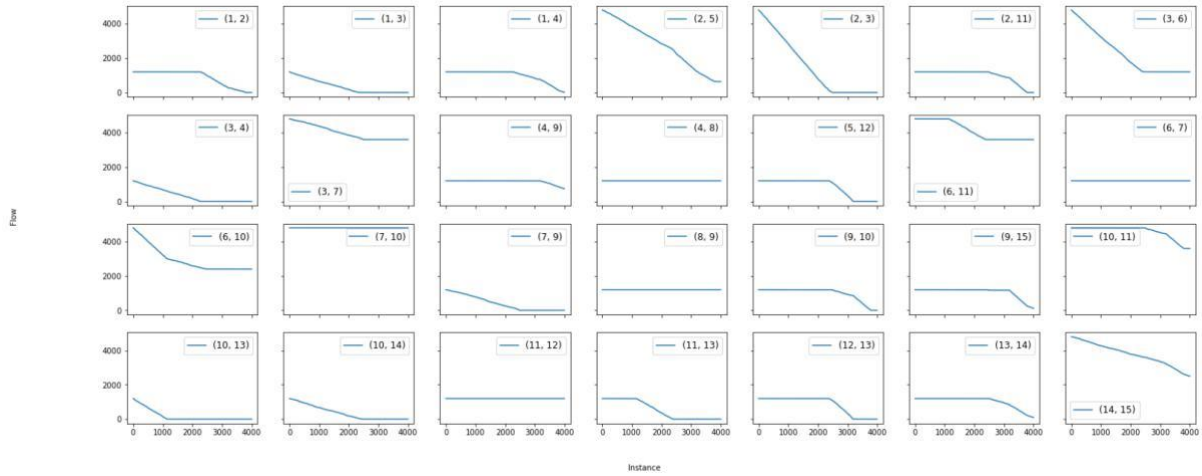

Fig 2.35 Flow Decrease on Links for BCRA

Although, links like (7,10), (11,12), (4,8), (6,7) are not used, the rejection count is quite low. This, states the fact that, these links might not be important for routing and hence do not necessarily improve the rejection count.

2.Rejection Count

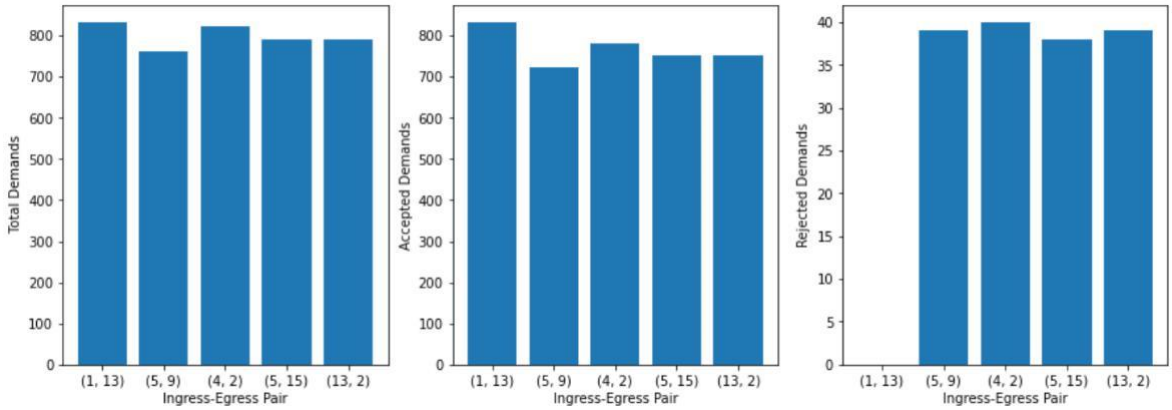

Fig 2.36 Rejection Count for BCRA

IEP (1,13) has 0 rejections.

3.Link Usage

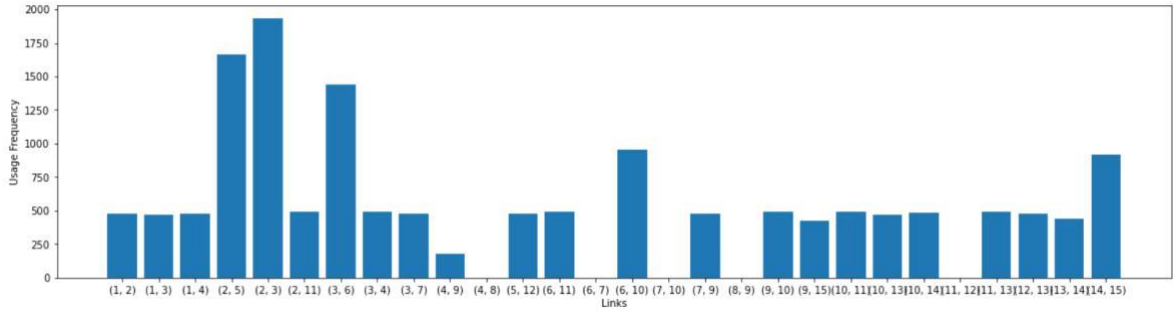

Fig 2.37 Link Usage for BCRA

Link Usage by IEP

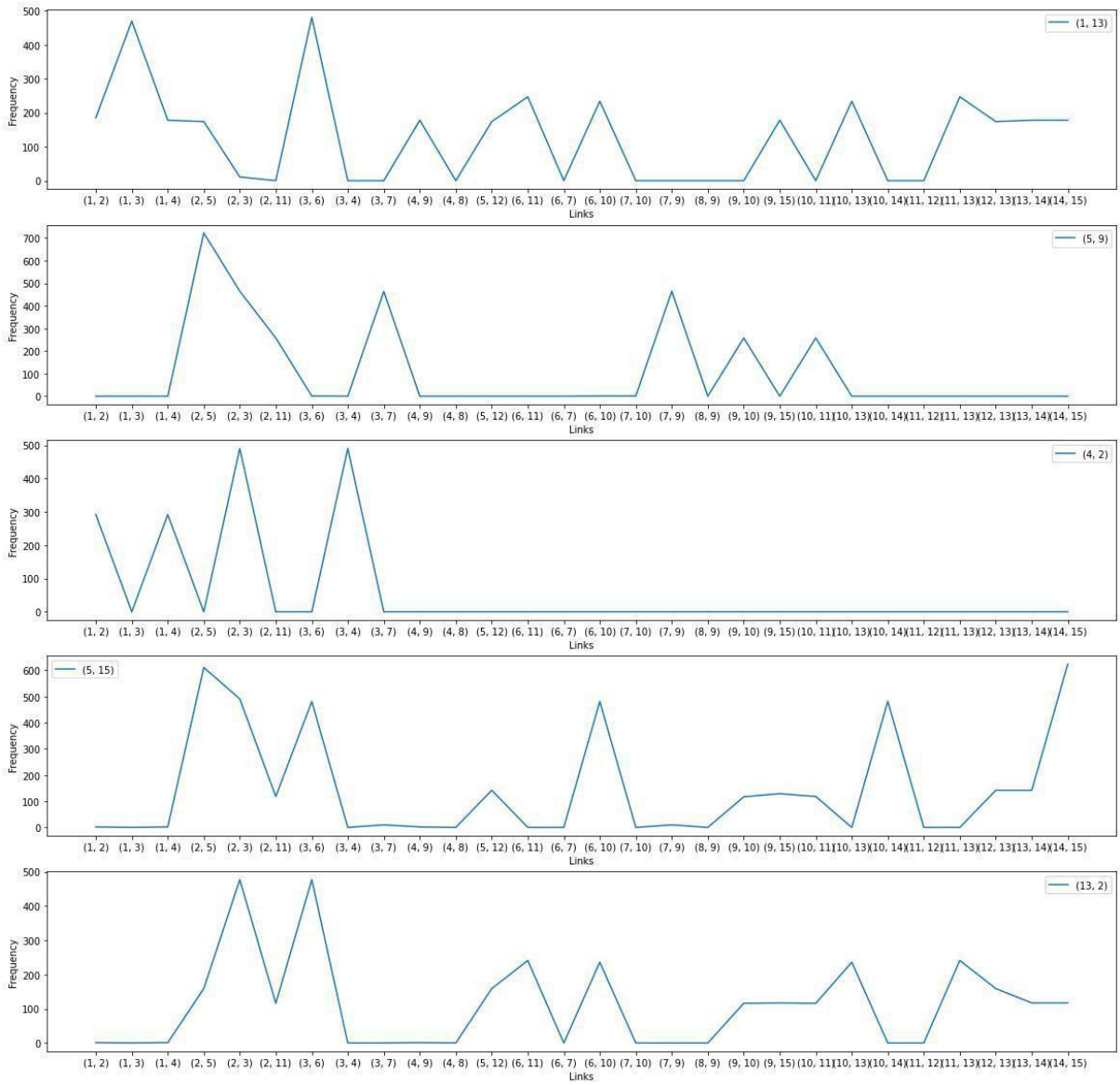

Fig 2.38 Link Usage by IEP for BCRA

IEP (4,2) uses links (1,2),(1,4),(2,3),(3,4) to a very high extent. It uses the other links to a very negligible extent.

4.Path Length

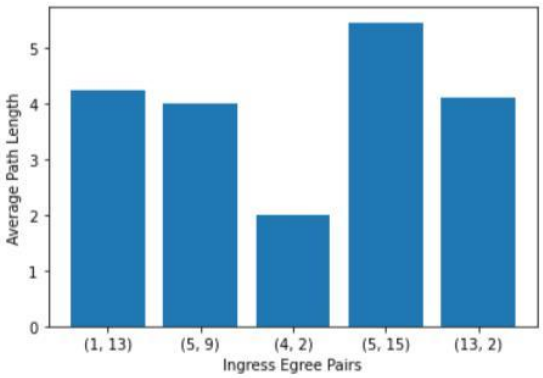

Fig 2.39 Average Path Length for BCRA

IEP (4,2) has path length averaging at 2.

5.Max Flow Decrease

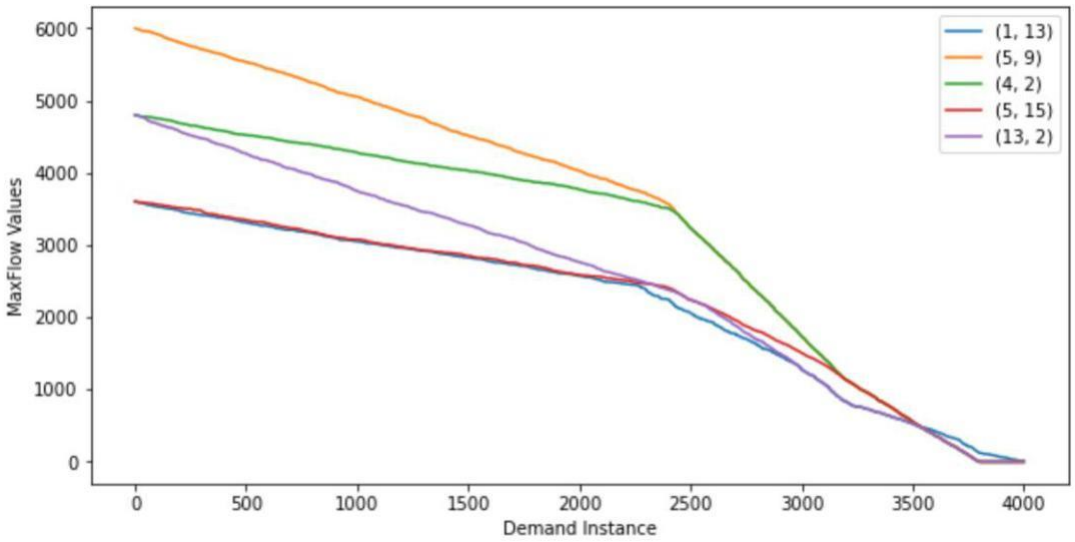

Fig 2.40 Max Flow Decrease for BCRA

IEP (4,2) faces a steep decline in maxflow value when compared to other IEPs.

2.8 RESIDUAL NETWORK AND LINK CAPACITY ALGORITHM(RNLC)

Salient features and Notes on RNLC:

Trying to avoid exacting calculations for each on-demand LSP request (e.g., maximum flow computation), which causes high computation effort, a new link weight function is introduced, which combines the following three criteria: saving of residual link bandwidth, optimal usage of network capacity, and minimization of path lengths. The link weights are calculated as a function of residual network capacity, link capacity, and a constant.

MIRA focuses exclusively on the interference effect on single ingress-egress pairs, and is not able to estimate the bottleneck created on links that are critical for clusters of nodes.

RNLC provides a new link weight function which combines three criteria: saving of residual link bandwidth, optimal usage of network capacity, and minimization of path lengths. The weight function is given by

( ) = ( ) + ,

where (= ∑∀ ∈ ( )) is the current residual network capacity and ( ) is the current residual link capacity on link at the arrival event of request . The constant determines the dynamic behavior of the generated link weights. Below Figure shows the dependence of this weight calculation scheme on ( ) and .

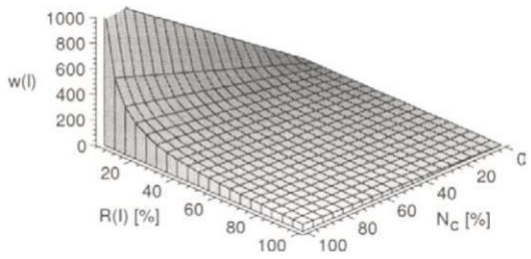

Fig 2.41 Dependence of Weight on R(l) and N_c

If the residual link capacity approaches zero, the weight approaches infinite, thus eliminating all links with insufficient residual capacity. The dependence of the weight function on residual network and link capacity needs some detailed consideration: In case of low network load (i.e., high residual network capacity), links with less residual capacity (higher load) are assigned considerably higher weights than less loaded links. Consequently, paths over lightly loaded links are preferred and heavily loaded links are avoided. This keeps as many links as possible available for future requests, i.e., intends to avoid congestion. In case of high network load (i.e., low residual network capacity), all link weights are approximately the same as long as there is sufficient residual capacity on the links. Therefore, the minimum-hop path is preferred, and routing is performed subject to minimum resource occupation, leaving a maximum of resources available for additional requests. Reflecting, if the same weighting as the one used for low loads is applied, detours would be preferred, which save some residual link capacity (bandwidth) on individual links, but due to their higher number of hops, these paths would occupy more network capacity, and consequently reduce the available resources for future requests.

With the additive constant the dynamic of the scheme can be controlled. If is chosen very big, the scheme behaves like minimum-hop routing, still having the advantage of eliminating links with vanishing residual capacity. The smaller is set, the stronger the link weights reflect the distribution of the load on the links, i.e., smaller increases the variance of the link weights. However, the value of the constant should be chosen according to topology, meshing degree and traffic distribution.

In this paper, we have presented studies on a fast and promising on-line routing algorithm for dynamic routing of bandwidth guaranteed LSPs. Our proposed routing algorithm is different from the two commonly used minimum-hop algorithm (MHA) and widest-shortest path (WSP) algorithm (not described in this paper). Our algorithm avoids using residual capacity of congested links by means of routing new traffic demands away from hot spots, even if the traffic then flows via slightly longer paths. This increases the acceptance rate for new demands, and reduces the rejection rate, which is especially required when re-routing due to a link failure is performed.

INPUT:

A graph G(N, L) and a set B of all residual link capacities An ingress node a and an egress node b between which a flow of D units have to routed.

OUTPUT:

A route between a and b having a capacity of D units of bandwidth.

ALGORITHM:

1. Compute weights of links w(l) according to above equations.

2. Eliminate all links which have residual bandwidth less than D and form a reduced network.

3. Using Dijkstra's algorithm compute shortest path in reduced network.

4. Route the demand of D units from a to b along this shortest path and update the residual capacities.

Below results are for C=1

1.Flow Decrease on Links

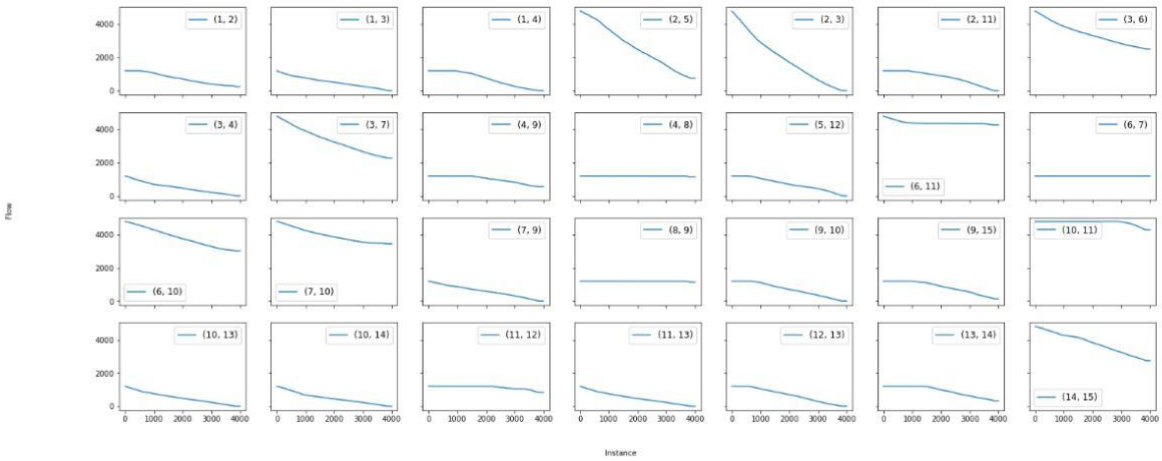

Fig 2.42 Flow Decrease on Links for RNLC

Link (1,4), (9,10), (9,15), (13,14) are dormant for the initial iterations. Links (11,12) and (10,11) get invoked at the end. Link (6,11) is used initially, but dormant for the remaining time.

2.Rejection Count

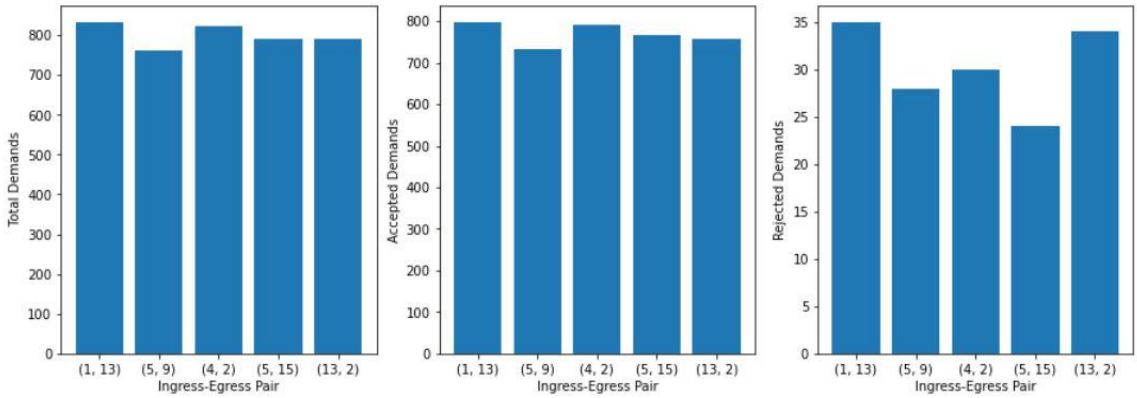

Fig 2.43 Rejection Count for RNLC

3.Link Usage

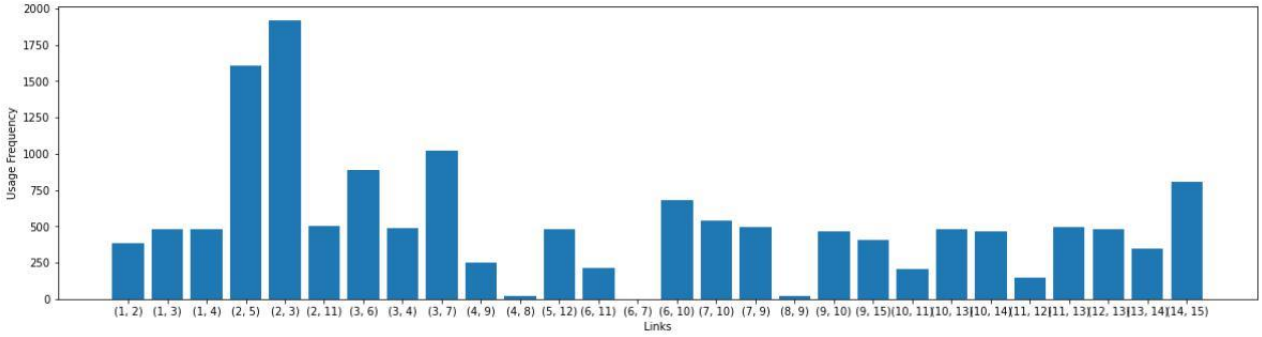

Fig 2.44 Link Usage for RNLC

Link Usage by IEP

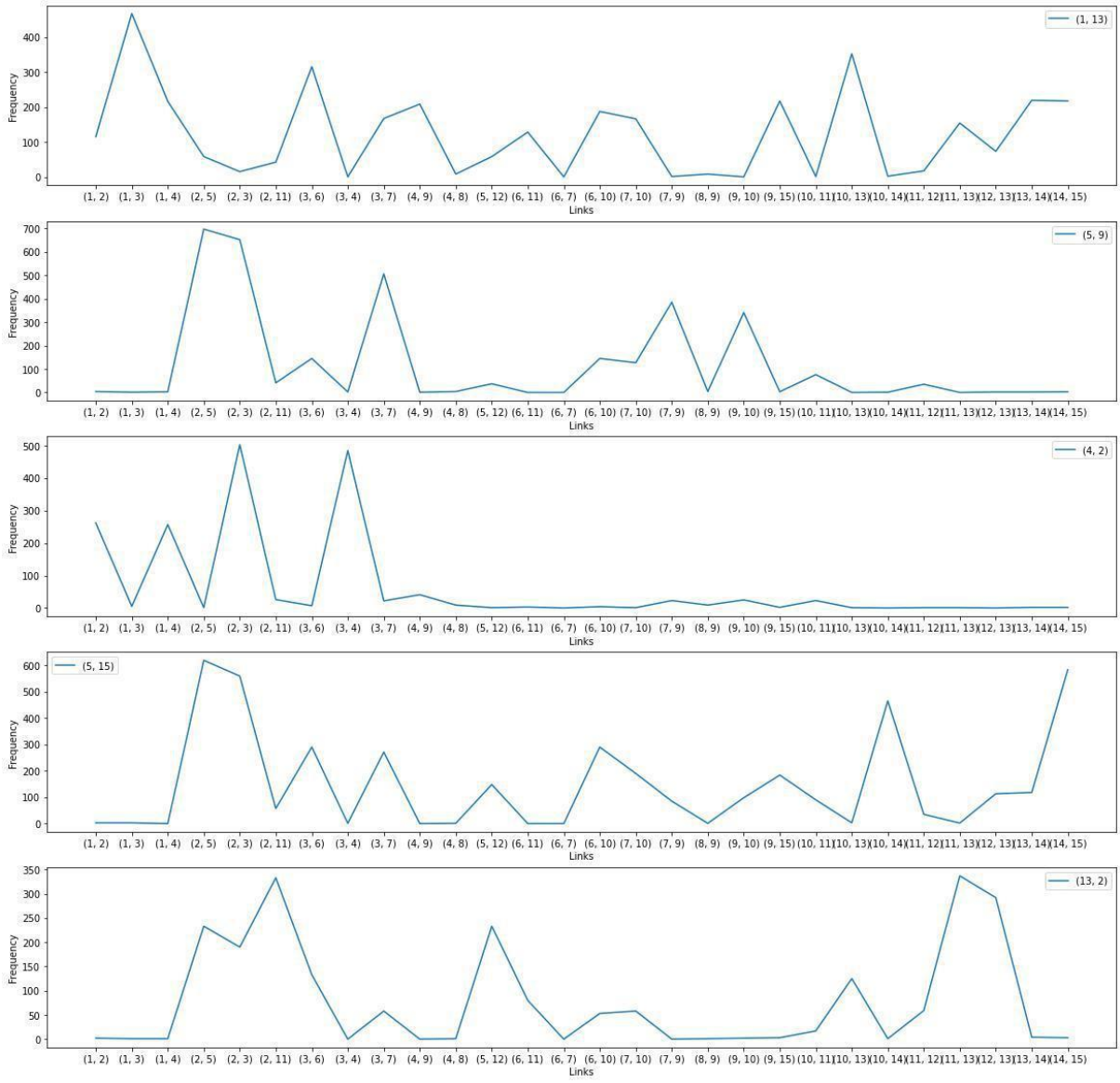

Fig 2.45 Link Usage by IEP for RNLC

For IEP (5,9), only links (7,9) and (10,9) are used for entry into node 9. Other links like (8.9) and (15,9) are not favourite entry points into node 9.

4.Path Length

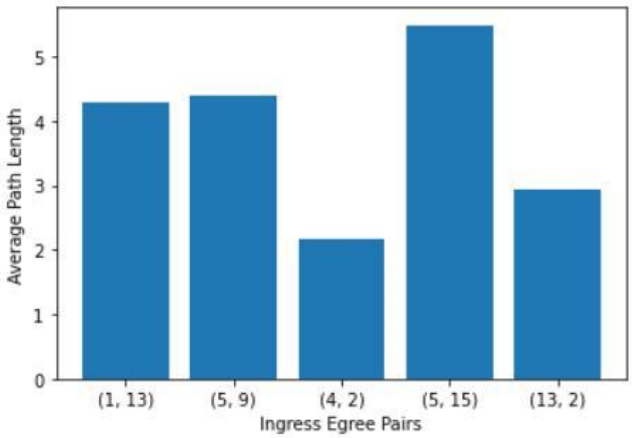

Fig 2.46 Average Path Length for RNLC

Again, Lowest path length is for IEP(4,2) averaging at 2.

5.Max Flow Decrease

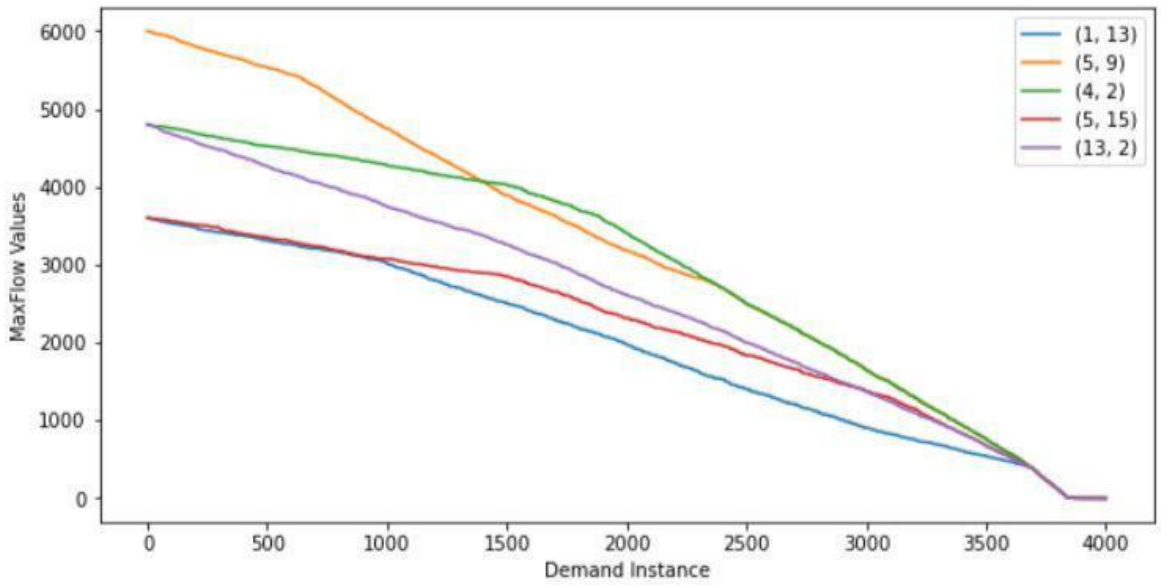

Fig 2.47 Max Flow Decrease for RNLC

No rapid decrease in Maxflow values of IEPs around instance 2500. IEPs (4,2) and (5,15), initially face a very slow decrease in maxflow values. Then their negative slope increases, causing a high drop in maxflow values at around instance 1500

Bonus: Trials with Various values of C

C	Rejection
0.0001	151
0.001	151
0.01	151

0.1	152
1	151
2	152
3	151
4	151
5	151
6	151
7	151
8	151
9	152
10	151
100	192
1000	475

Table 2.4 Rejection Count for RNLC with Various Values of C

2.9 MAXIMIZE RESIDUAL BANDWIDTH AND LINK CAPACITY(MAX-RC) MINIMIZE FLOWS(MIN-F)

Salient features and Notes on Max-RC Min-F:

Objectives of MPLS routing are:

- Minimizing interference level among source-destination node pairs, in order to reserve more resource for future bandwidth demands.
- Balancing traffic loads through under-utilized paths other than a shortest path, in order to reduce network congestion.
- Taking into account both static and dynamic information of link states, in order to improve performance of routing algorithm over dynamic condition.

To satisfy above objectives, link state information which is residual bandwidth, link capacity, and total flow are to be involved in the algorithm.

Firstly, links with high number of total flow could indicate high interference level on those links. Therefore, we try to select LSP with minimal number of total flow in order to avoid link with high interference level. This could satisfy the first objective. Further, total flows over link are a dynamic link state information. This could also satisfy the third objective.

Secondly, new incoming traffic should be traversed onto under-utilized paths having high residual bandwidth (or available bandwidth). Then, we try to select LSP with high amount of residual bandwidth. Since, the residual bandwidth is a dynamic link state information. Therefore, this approach could satisfy both the second and the third objectives.

Lastly, link capacity is also a static information which could be used for load balancing purpose. Thus, we try to choose LSP with high link capacity in order to share traffic load over the network. Similarly, residual bandwidth, this method can satisfy with both the second and the third objectives.

Calculation of Link Weight

Let us first define some parameters involved in the calculation of link weight.

$G(N, L)$: a network graph

N : set of nodes
L : set of links
S : a source node, $S \in N$
D : a destination node, $D \in N$

$\{S, D\}$: set of source-destination node pairs

F_j : total flows over link j, $j \in L$
R_j : residual bandwidth of link j, $j \in L$
C_j : capacity of link j, $j \in L$
w_j : weight of link j, $j \in L$

$W_{\{D\}}$: weight of path belonging to source-destination node pair $\{S, D\}$

Here, weight of link could be determined by below equation.

=

From the above equation, the higher link weight value ( ) indicates the higher total flows and the lower residual bandwidth and link capacity. On the other hand, the lower link weight value signifies the lower total flows and the higher residual bandwidth and link capacity. So, Max RC-Min F algorithm intends to choose a LSP cooperating with those low weight links.

Path Selection

The weight of path belonging to source-destination node pair {S, D}($W_{,D\}}$) could be obtained by below equation.

$$\{ , \} = \sum_{\in \{ , \}}$$

This path weight ($_{\{S,D\}}$) is a key for MaxRC-MinF algorithm to route LSP from ingress node S to egress node D. The constraint of path selection could be expressed below as:

$$\min(\{ , \})$$

However, if there are many result paths with the same minimum path weight, the algorithm would pick a shortest path (or minimum hop path) between those result paths in order to reserve network bandwidth.

INPUT: A graph G (N, L) and a set B of all residual link capacities. An ingress node a and an egress node b between which a flow of D units have to routed.

OUTPUT: A route between a and b having a capacity of D units of bandwidth.

ALGORITHM:

1.Reduce network graph by eliminate all links that have residual bandwidth less than request bandwidth.

2.Calculate link weight (w_j).

3.Use Dijkstra routing algorithm to obtain shortest path with minimum path weight $(W_{s,D\}})$.

4.Establish the result path found in Step 3. Deduct capacities from links.

5.If no path is selected, the demand is rejected.

1.Flow Decrease On Links

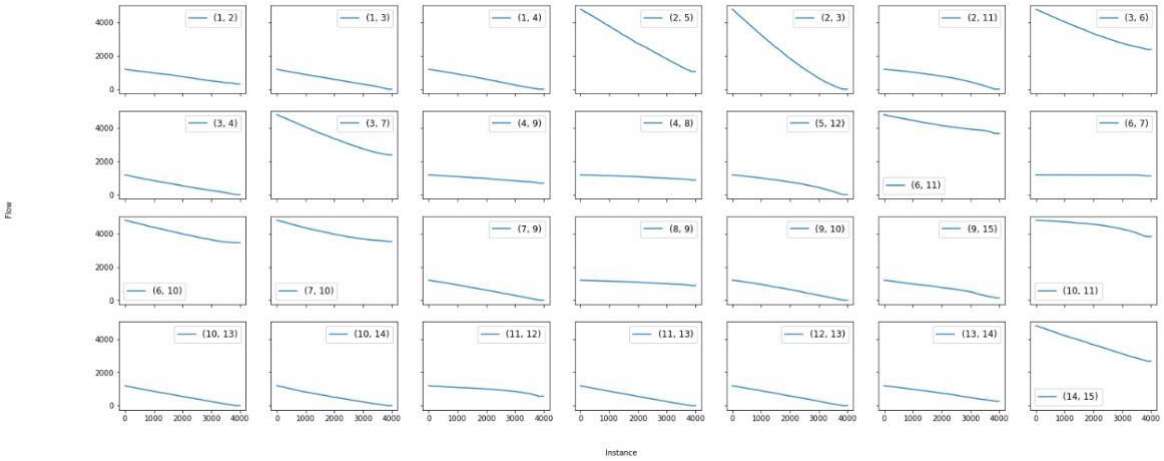

Fig 2.48 Flow Decrease on Links for Max RC Min F

An important observation: Except links (6,7),(8,9) and (4,8), all links are used from the start. Some links have a rapid decrease in their capacity(eg (2,5), (2,3), (14,15) etc) , while others have a slow decrease.

2.Rejection Count

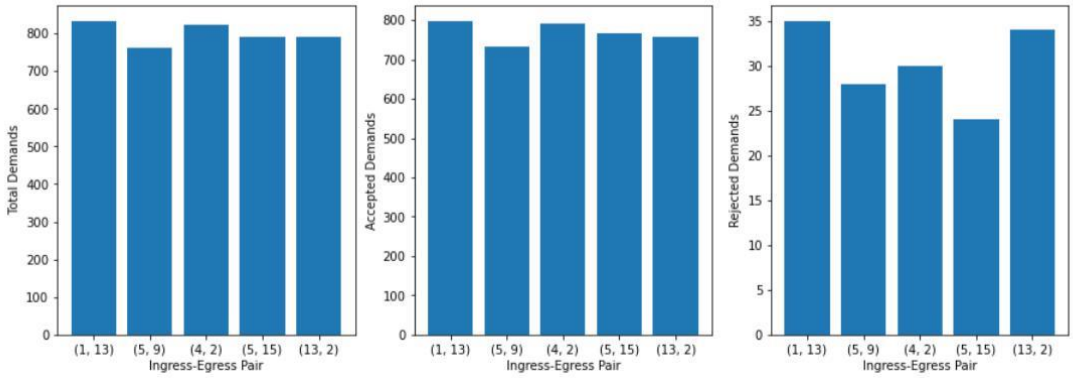

Fig 2.49 Rejection Count for Max RC Min F

3.Link Usage

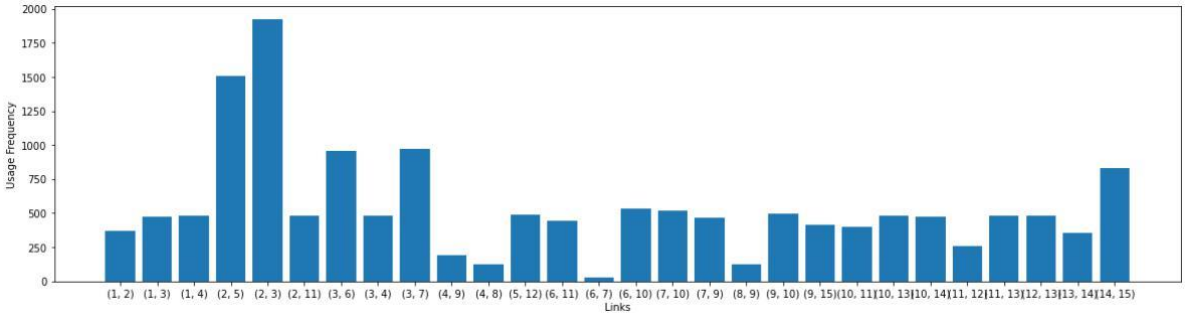

Fig 2.50 Link Usage for Max RC Min F

Link Usage by IEP

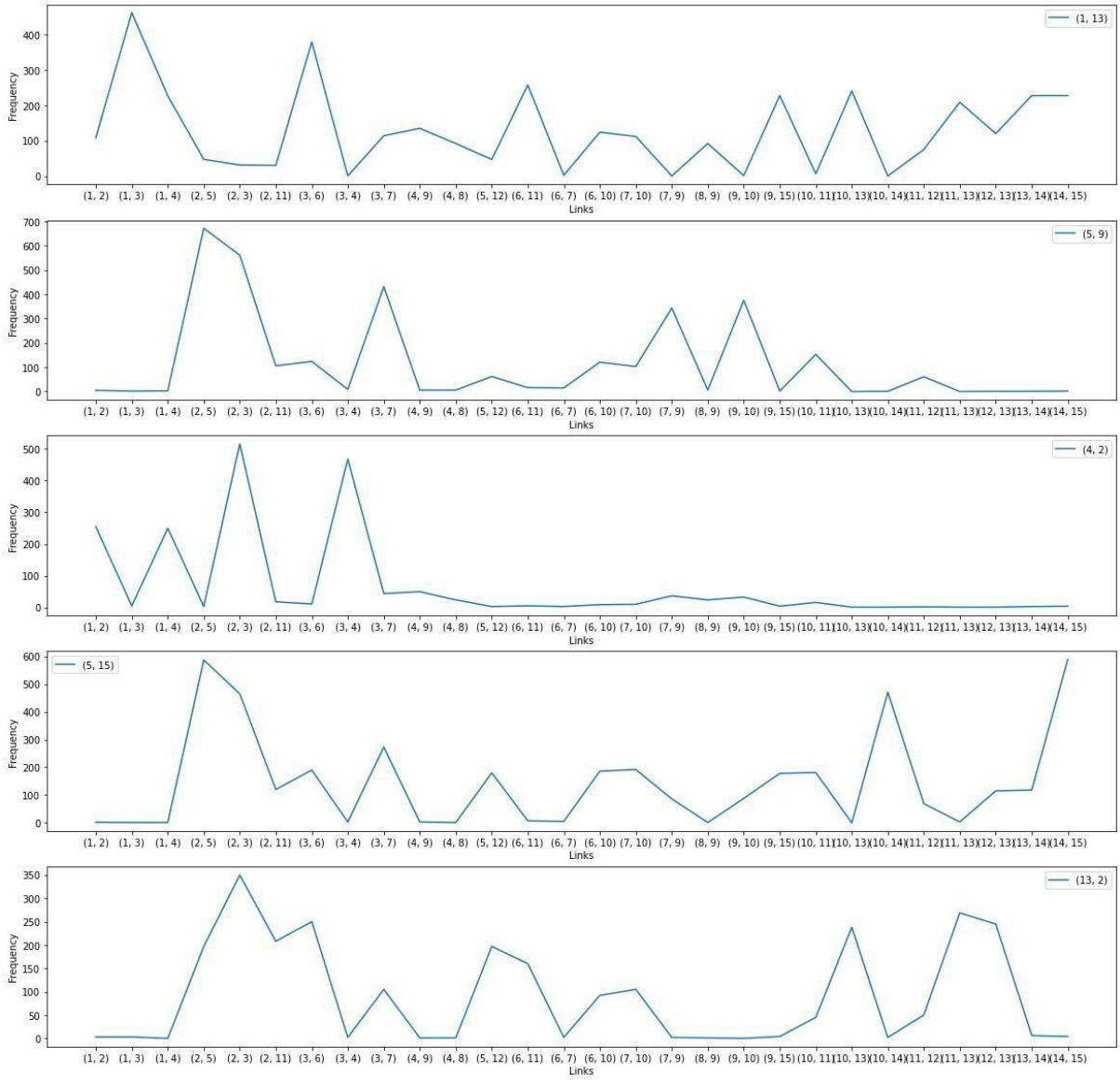

Fig 2.51 Link Usage by IEP for Max RC Min F

Except IEP(4,2), all IEPs put many of the links to good use.

4.Path Length

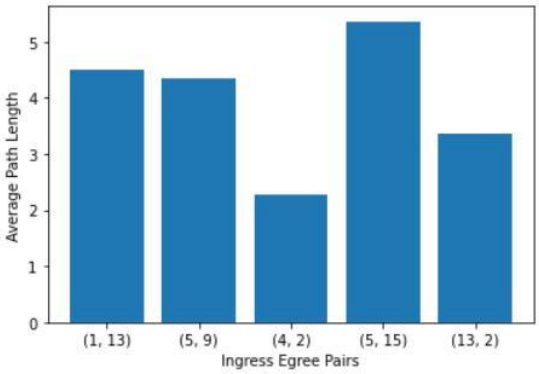

Fig 2.51 Average Path Length for Max RC Min F

5.Max Flow Decrease

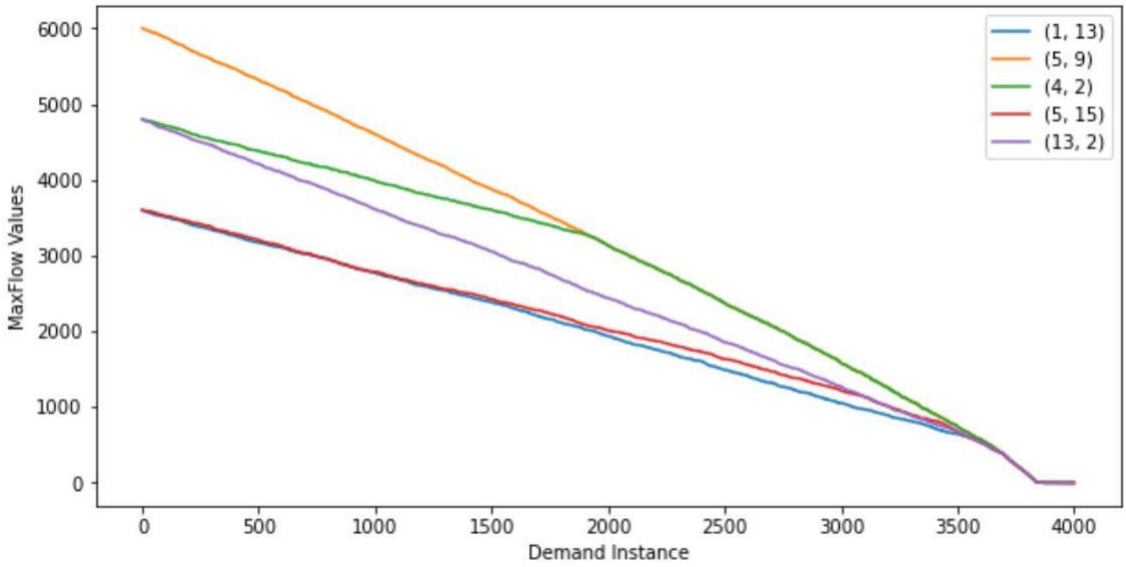

Fig 2.53 Max Flow Decrease for Max RC Min F

Gradual/Linear decrease in maxflows of IEPs, with no sudden drops even at instance 2500.

2.10 NEW HYBRID MINIMUM INTERFERENCE(NHMI)

Salient Features and Notes on NHMI:

Each LSP set-up request arrives at ingress node, which is responsible to determine an explicit-route for the LSP. To determine the route, each ingress router should know the whole network's topology and state information of the links. In other words, this is source routing. The initial capacity of link is denoted as ( ). The current available bandwidth of link is denoted as ( ). The request for an LSP set-up is defined by a triple (, ,), where (,) ∈ , is the ingress node, is the egress node, and is the amount of bandwidth required by the LSP. The requests arrive on line, one by one, and there is no prior knowledge for future demands. The objective is to find a feasible path (if exists) for LSP request in the network from to , otherwise the request will be rejected. At the same time, the route selection must make efficient use of network resources. In this algorithm, more focus is on the route selection of bandwidth guaranteed paths. No re-routing or request splitting is allowed.

When selecting route for a LSP request, Not only the hop counts and the link residual capacity are considered, but also the link criticality.

Given an LSP request (, ,) to be routed, A new link weight function is defined as below.

$$( ) = 1 + 2 \ ( ) - ( ) + \Sigma(,) \in \backslash(,) \quad \frac{}{( )} \qquad \frac{}{( )}$$

1 and 2 are tunable parameters which allow the operators to adjust the comprise among different objectives.

The first term 1 represents the objective of minimizing hop counts. A shortest path will bate the consumption of network resources while routing the same bandwidth demanded. But the simple shortest path first algorithm may produce the hot spots, which will degrade the performance of operational network. So we should take link utilization and interference into consideration. When the network load is high, the algorithm will trend to hop counts minimize by setting a big value of 1.

The second term 2(( ) − ( ))/ ( ) represents the objective of load balancing. If the link utilization is high, our algorithm will try to avoid routing LSP on such links. If the network operators want to emphasize the objective of load balancing, they can realize this by setting a bigger value of 2.

The third term (∑(,)∈ \(,) ( ))/(+ ( )) represents the objective of minimizing interference.

Where b is the bandwidth demanded, represents the relative importance of the SD pair (,), ( ) is the amount of sub-flow traveling through the link when the maximum flow between (,) is achieved. ∑(,)∈ \(,) ( ) represents the weighted sum of sub-flows of all other SD pairs traveling through link . If + ( ) is bigger and ∑(,)∈ \(,) ( ) is smaller, we believe that routing current request (, ,) on link will cause less impact on other SD pairs; so the link weight will be smaller. The algorithm will trend to route LSP on such links. Otherwise the algorithm will trend to avoid such links.

Given a network represented by a directed graph (,) where is a set of nodes and is a set of links. The number of nodes is and the number of links is . The LSP setup requests are between specific source nodes (ingress) and destination nodes (egress). The SD pairs are { 0, 0}, { 1, 1}, ... , { , }, where is the number of SD pairs. We denote all these SD pairs by a set .

INPUT:

A residual graph $G = (V, E)$ and LSP request $r(s, d, b)$, which is a request for b bandwidth units between pair (s, d).

OUTPUT:

A path from s to d with b bandwidth units.

ALGORITHM:

1: Compute the maximum network flow for all $(a, b) \in P$.

2: Compute the weight $w(l)$ for all $l \in E$ according to above equation.

3: Eliminate all the links whose residual bandwidth less than b and get a reduced network topology.

4: Run Dijkstra algorithm using $w(l)$ as the weights in the reduced network.

5: Create an LSP connecting s to d with b bandwidth units and update the links' available bandwidth.

1.Flow Decrease on Links

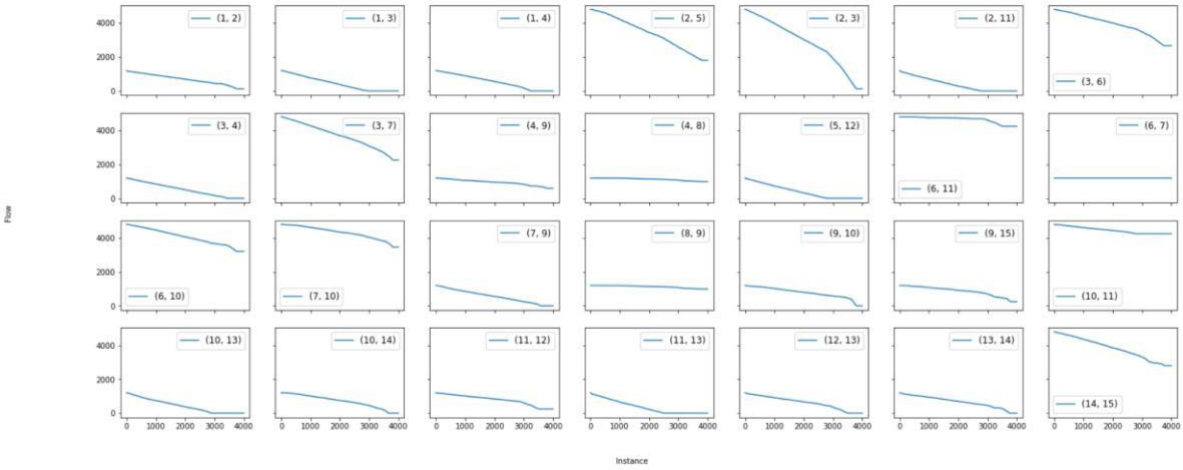

Fig 2.54 Flow Decrease on Links for NHMI

2.Rejection Count

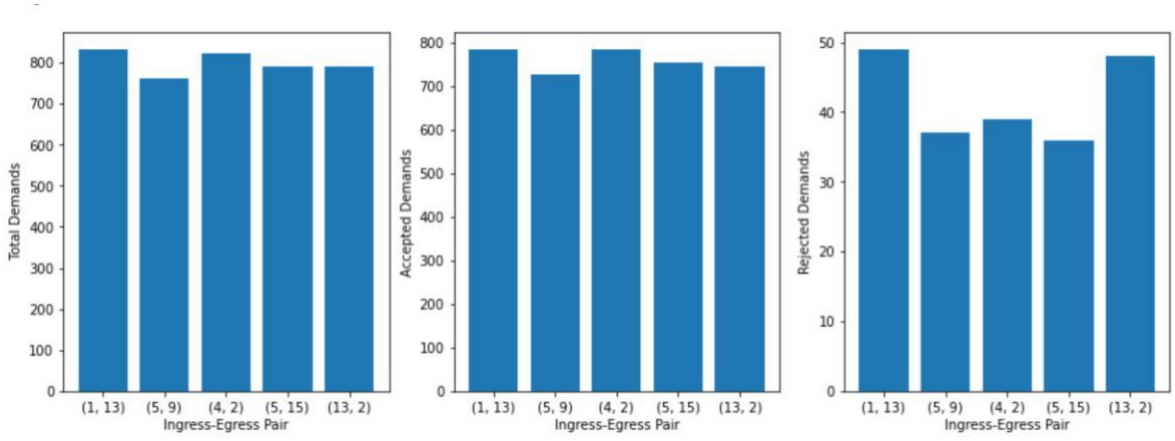

Fig 2.55 Rejection Count for NHMI

3.Link Usage

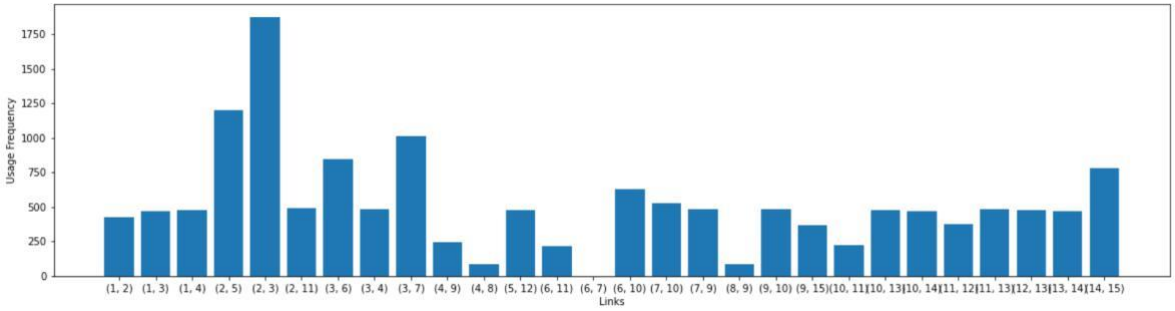

Fig 2.46 Link Usage for NHMI

Link Usage By IEPs

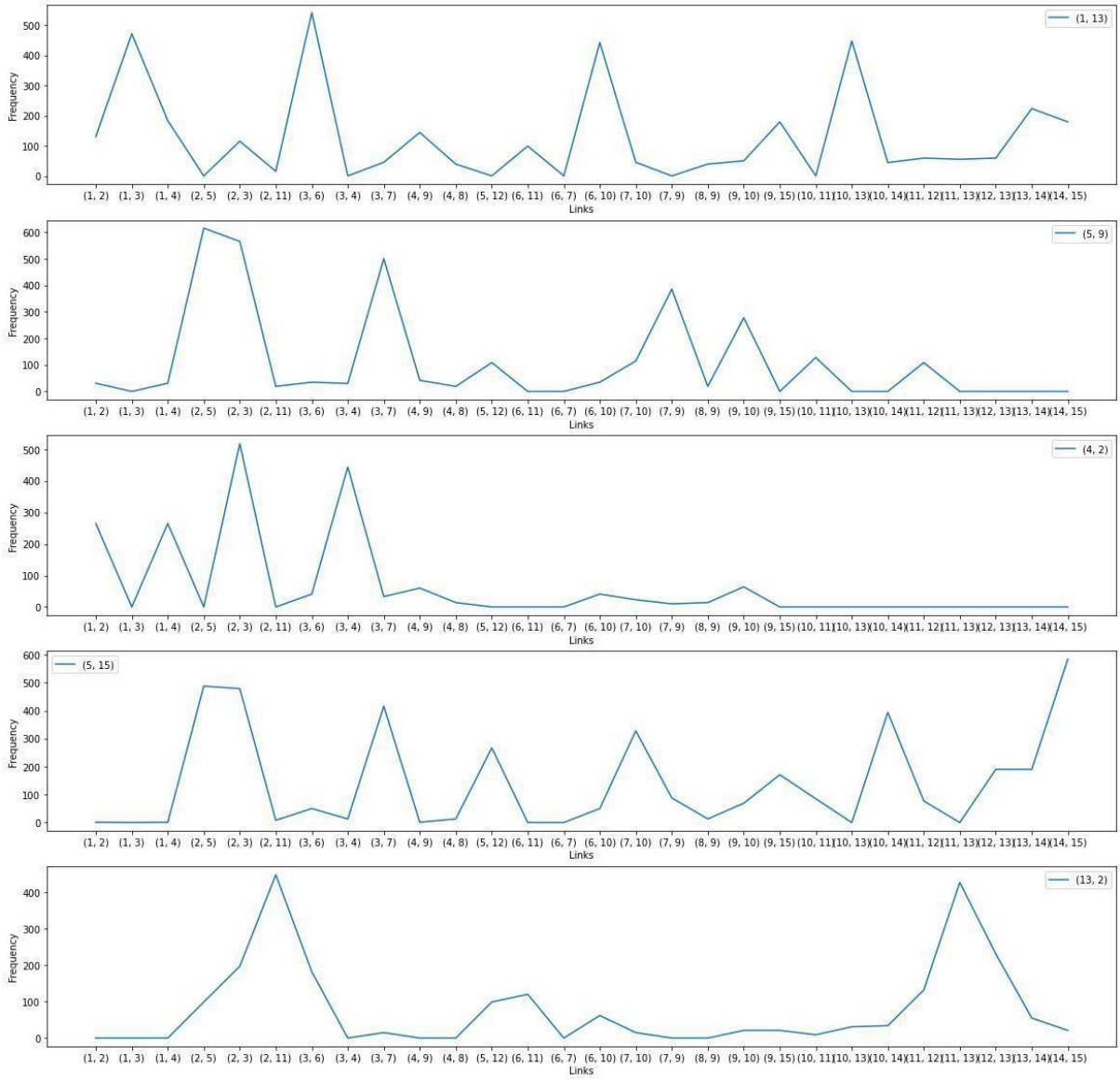

Fig 2.57 Link Usage by IEP for NHMI

4.Path Length

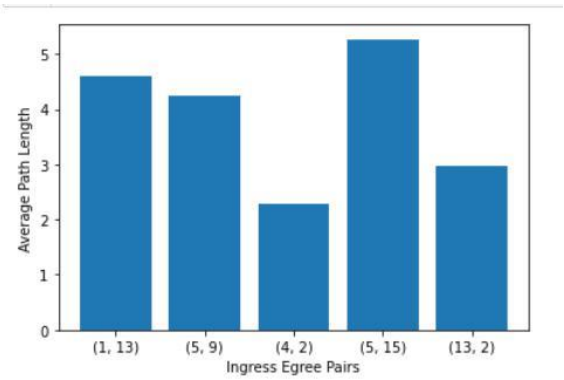

Fig 2.58 Average Path Length for NHMI

5.Max Flow Decrease

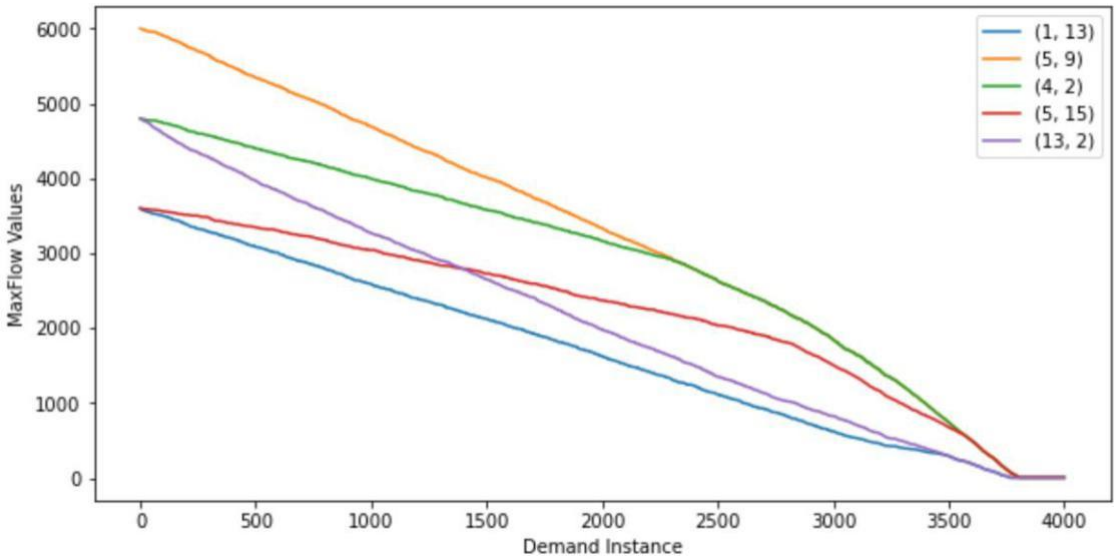

Fig 2.59 Max Flow Decrease for NHMI

IEPs (4,2) and (5,15), seem to very gradually lose their maxflow values upto instance 2500-3000. At this point, the slope decreases slightly, resulting in a faster decrease in max flow value.

2.11 LEXICOGRAPHIC MAX FLOW(LEX-MAX)

Salient Features and Notes on LEX-MAX:

B. Δ-Critical Links

A Δ-critical link for an ingress-egress pair is a link such that if the capacity of the link is decreased by Δ, the maxflow value for the ingress-egress pair decreases.

In the definition of Δ critical links we just insist that the maxflow value decreases (not necessarily by Δ units). The notion of a link being Δ critical captures the notion that there may be links that are close to being critical that we may want to identify. **This is especially important if the computation of critical links is done periodically, i.e., once every demands.** In this case it is important to "protect" links that are currently not critical but are close to being critical because they may become critical before

the computation is performed again. Thus the weight function would be

$$(, \Delta) = \sum_{: \in '(\Delta)} $$

where $' '(\Delta)$ is the set of Δ-critical links for (′, ′), the ingress-egress pair with th smallest maxflow.

Next comes the question of finding the set of Δ-critical links for a given value of Δ. One naive way to find whether a link is Δ-critical for an ingress-egress pair is to reduce the capacity of the link by an amount Δ, and compute the maxflow all over again, and check if the maxflow value decreases. However, this approach is prohibitively expensive, since it requires times the time for a single maxflow computation to find links that are critical. A faster approach of finding Δ-critical links is an open algorithmic problem. Instead of determining the set of Δ critical links exactly, we determine the approximate set of Δ critical links.

APPROX Δ-CRITICAL LINKS

INPUT:

A graph $G(N, L)$ and a set B of all residual link capacities
A threshold Δ and a source-destination pair $(s, d) \in \mathcal{P}$

OUTPUT:

$C_{sd}(\Delta)$, the set of Δ-critical links for source-destination pair (s, d)

ALGORITHM:

1.$C_{sd}(\Delta) \leftarrow \phi$

2.Compute maxflow f for source-destination pair (s, d)
Let $G_f(N, L)$ be the flow residual graph after the maxflow computation, and $R_f(l)$ be the residual capacity of any link $l \in L$ in $G_f(N, L)$.

3. For each link $l = (i, j) \in L$, include l in $C_{\text{sd}}(\Delta)$ if and only if all the following two conditions hold:

- $R_f(l) < \Delta$
- There is no path between i and j in graph $G_f(N, L)$ with capacity greater than $\Delta - R_f(l)$

MINIMUM INTERFERENCE ROUTING ALGORITHM ($\mathcal{L}-\mathcal{MLRA}$)

INPUT:

A graph $G(N, L)$ and a set B of all residual link capacities
An ingress node a and an egress node b between which a flow of D units have to routed.

OUTPUT:

A route between a and b having a capacity of D units of bandwidth.

ALGORITHM:

1. Choose an appropriate Δ
2. For all $(s, d) \in \mathcal{P}$, use APPROX Δ-CRITICAL LINKS to compute the maxflow and the set of critical links $C_{sd}(\Delta)$
3. Order the ingress-egress pairs $(s, d) \in \mathcal{P}$ in the ascending order of their maxflow values. Let (s_i', d_i') be the i th ingress-egress pair in that order, $i = 1$ to p.
4. Compute the weights $w(l, \Delta) = \sum_{i:l \in C'_{s_i' d_i'}(\Delta)} \alpha_i \; \forall l \in L$.
   (α_i are chosen according to the expression in (12))
5. Eliminate all links which have residual bandwidth less than D and form a reduced network.
6. Using Dijkstra's algorithm compute shortest path in reduced network using $w(l, \Delta)$ as the weight on link l.
7. Route the demand of D units from a to b along this shortest path and update the residual capacities.

For the weighted case, we just have to modify step 3 of the algorithm and order the ingress-egress pairs according to their weighted maxflow values, instead of their actual maxflow values.

An appropriate choice of Δ is very important for a good performance of the algorithm. If Δ is chosen to be larger than the maximum capacity of the links, then our algorithm reduces to the min-hop algorithm. On the other hand, if Δ is set to zero, then we may miss out some near-critical links, and thus the demand may be routed on those links thus reducing the maxflows of some ingress-egress pairs considerably. If the new demand that is to be routed is units, then may be a reasonable choice for Δ, since no link whose index of criticality is more than can turn critical by routing a demand of units. One interesting case is when we are not computing the maxflows (and hence the critical links) on every demand arrival. Note that the maxflow calculation is computationally expensive, and when the demands are much smaller than the capacities of the links, we do not expect the maxflow values and the critical links to change very drastically. Therefore, we may not degrade the performance of the system significantly by doing the maxflow and the critical link calculations after, say, a few demand arrivals, or after a fixed time, or after the residual capacity of some link has changed beyond a certain threshold. In that case, we may be able to make a rough estimate of how much of bandwidth demand can arrive and be routed between the current maxflow computation and the next one, and use that information to determine Δ. The algorithm reduces to a Min Hop Algorithm for Δ >Max Link Capacity.

For most of the experiments that the researchers have carried out, they have observed that the minimum rejection ratio is achieved either at Δ = , or Δ = 1, when the demand size is kept fixed at . However, in a practical scenario, the demand size would not be fixed, and would vary over a range. What value of Δ

achieves the minimum rejection ratio in that case remains to be seen. However, one would expect that a good choice of Δ would have to take into account the distribution of the demand size.

Note that when the demand sizes are small compared to the link capacities (which is typically the case), setting = may not be a bad choice, if the critical links are computed on every request arrival. However, if the maxflow computation, and hence the determination of the critical links, is done less frequently, then setting $\Delta = 1$ may be a bad choice, even if the demand sizes are small. In this case, a lot of flow might be pushed into the network between two successive maxflow computations, and hence links that were not the most critical during a particular maxflow computation might become the most critical links before the next maxflow computation occurs. Thus we would like to set Δ to some larger value, such that it identifies the near-critical links, i.e., the links that can potentially become bottlenecks before the next maxflow computation. When Δ is too small, then we are missing out too many near-critical links, while when Δ is too large, then too many links are being included in the set of critical links, reducing the algorithm to minhop. The value of Δ which achieves the minimum rejection ratio, would, in general, depend on the demand size distribution and the interval (in terms of LSP request arrivals) between successive maxflow computations.

1.Flow Decrease On Links

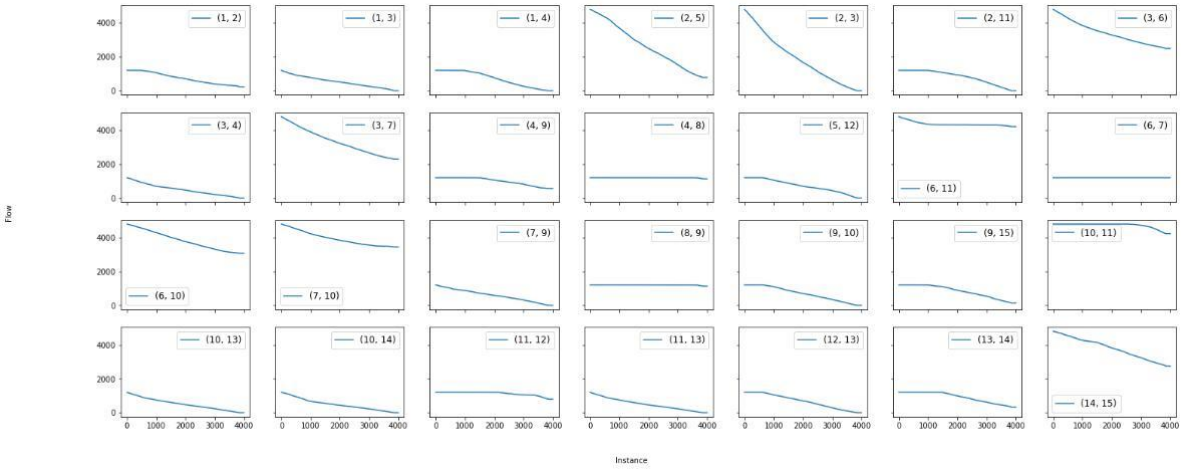

Fig 2.60 Flow Decrease on Links for LEX MAX

2.Rejection Count

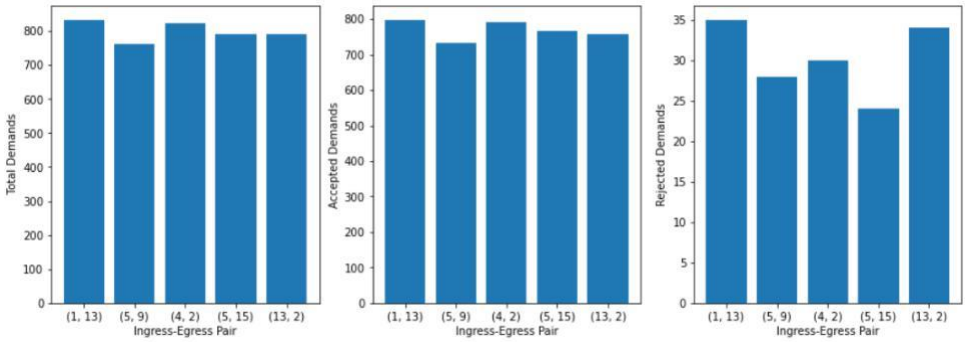

Fig 2.61 Rejection Count for LEX MAX

3.Link Usage

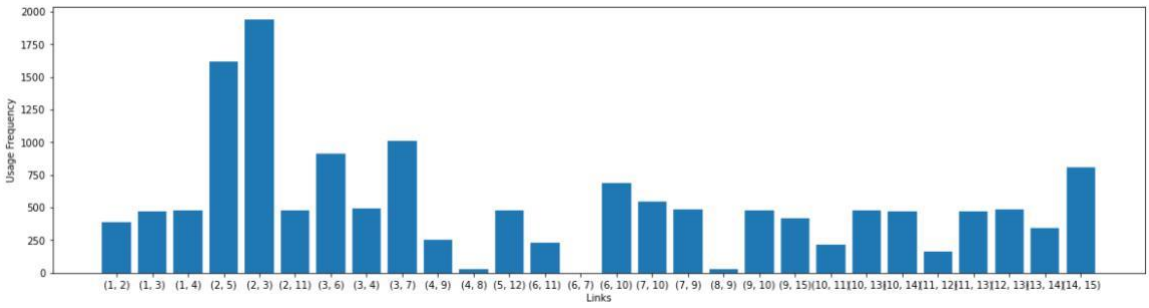

Fig 2.62 Link Usage for LEX MAX

Link usage by IEP

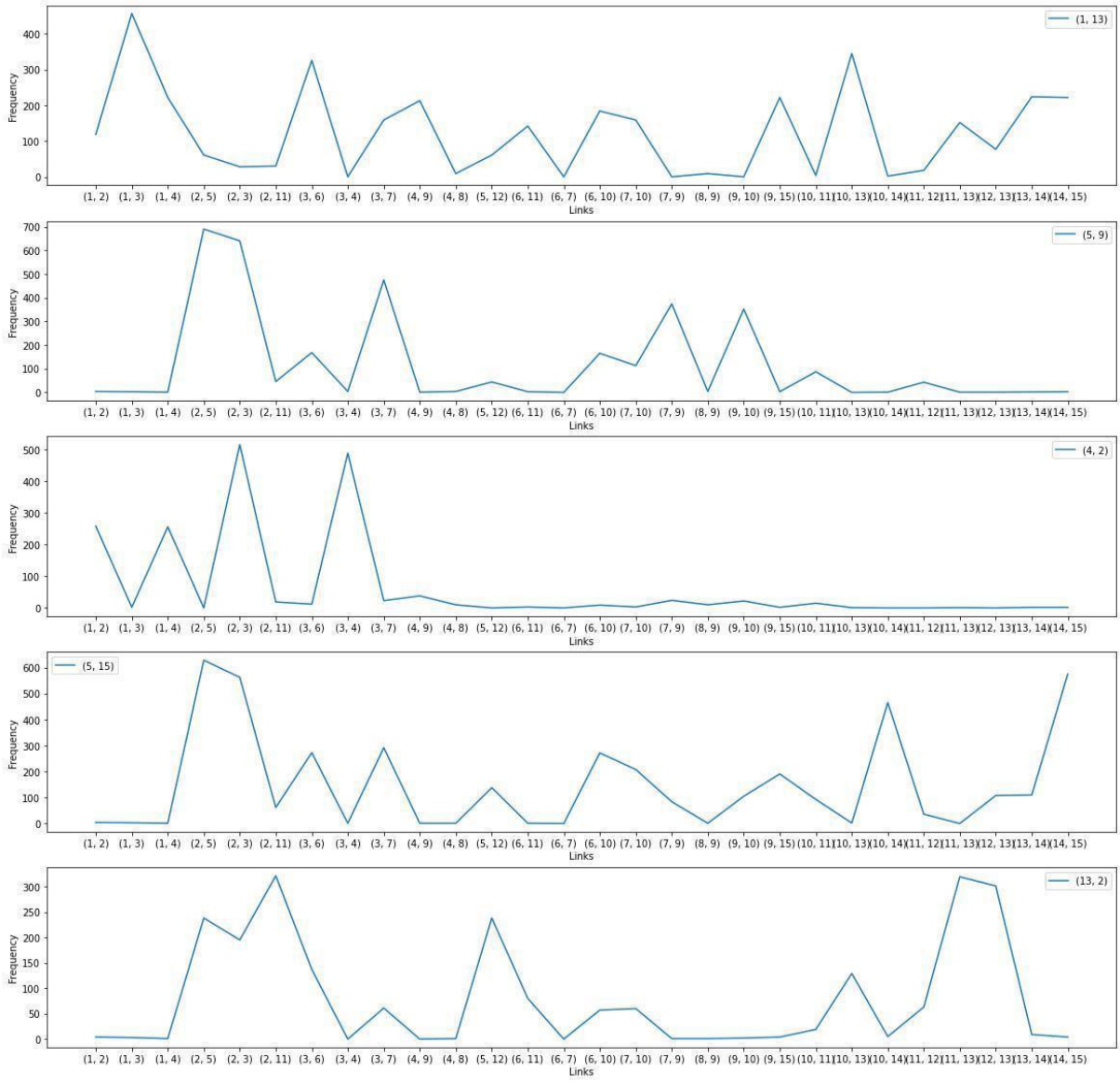

Fig 2.63 Link Usage by IEP for LEX MAX

4.Path Length

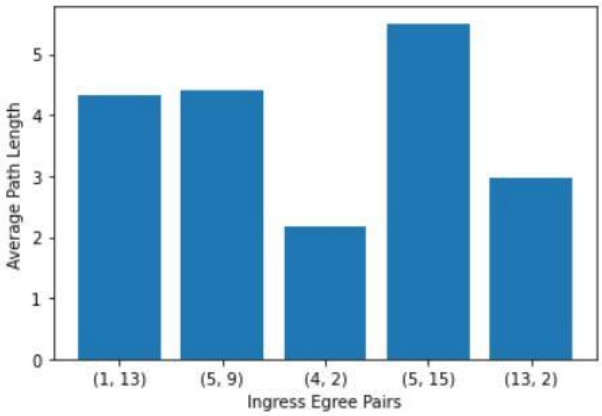

Fig 2.64 Average Path Length for LEX MAX

5.Max Flow Decrease

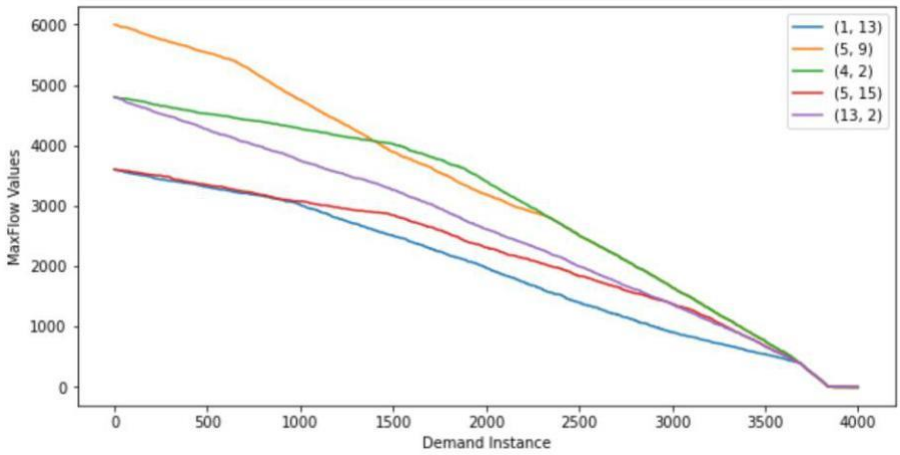

Fig 2.65 Max Flow Decrease for LEX MAX

2.12 WIDEST SHORTEST PATH(WSP)

Salient Features and Notes on WSP:

For using WSP as a routing algorithm, we first compute the shortest paths between a given a source and destination having sufficient residual bandwidth >d on each link.
Then, if we get 2 or more paths, that have the same path length, we then find the widest path, which is the path that has highest residual bandwidth.
Residual Bandwidth of a path= min (Residual Bandwidth of l, where $l \in L_p$, where L_p is the set of all links in the chosen path.

$$\text{Residual Bandwidth } (p) = \min (\text{Residual Bandwidth } (l)) \quad - (1)$$

Where $p \in P$ and $l \in L_p$, P={set of shortest paths of same path length} and L_p={set of links in path p}

INPUT:

A graph G (N, L) and a set B of all residual link capacities. An ingress node a and an egress node b between which a flow of D units have to routed.

OUTPUT:

A route between a and b having a capacity of D units of bandwidth.

ALGORITHM:

1. Set the weights of all links =1.

2. Eliminate all links which have residual bandwidth less than D and form a reduced network.

3. Using Dijkstra's algorithm compute shortest path (Shortest) in reduced network.

4. Now find the path that has the highest residual bandwidth (Widest) according to (1)

5. Route the demand of D units from a to b along this path (Widest Shortest) and update the residual capacities.

1.Flow Decrease on Links

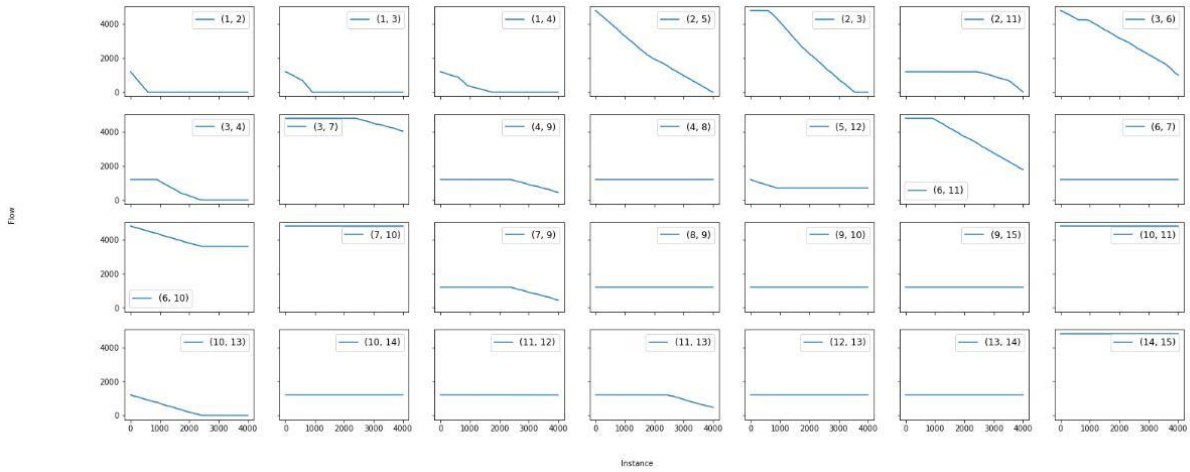

Fig 2.66 Flow Decrease on Links for WSP

Links (2,11), (7,9), (3,7) get used only during the last iterations.

2.Rejection Count

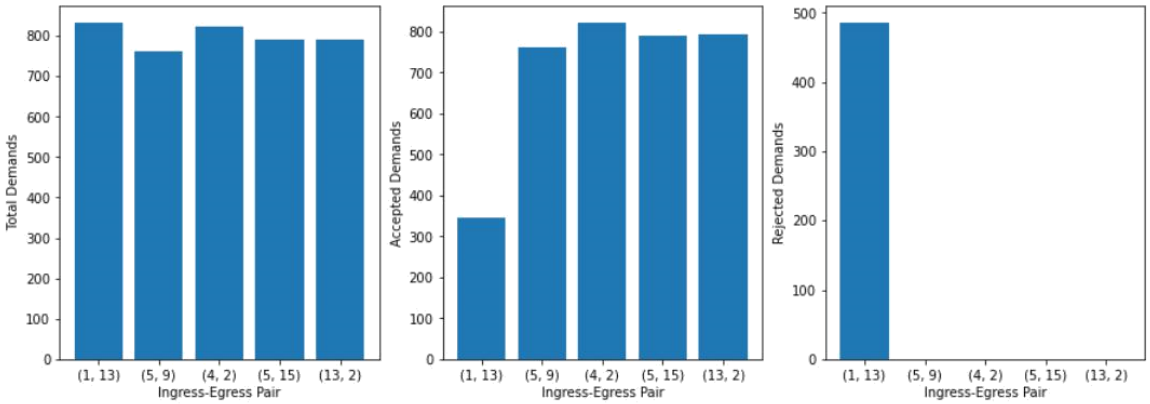

Fig 2.67 Rejection Count for WSP

IEPs other than (1,13), have 0 rejections.

3.Link Usage

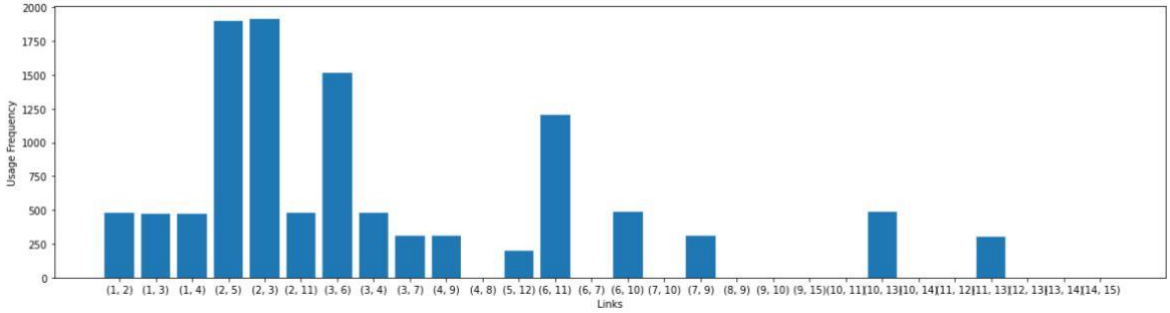

Fig 2.68 Link Usage for WSP

Many links are not used, in contrast to other routing algorithms.

Link Usage by IEP

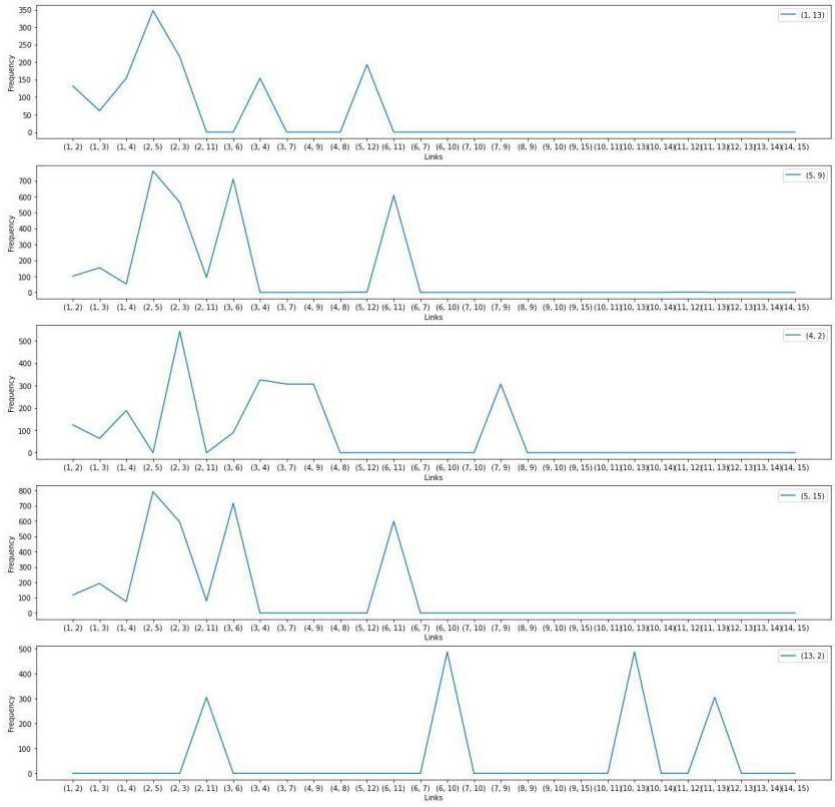

Fig 2.69 Link Usage by IEP for WSP

We see a particular affinity of an IEP towards certain links.

4.Path Length

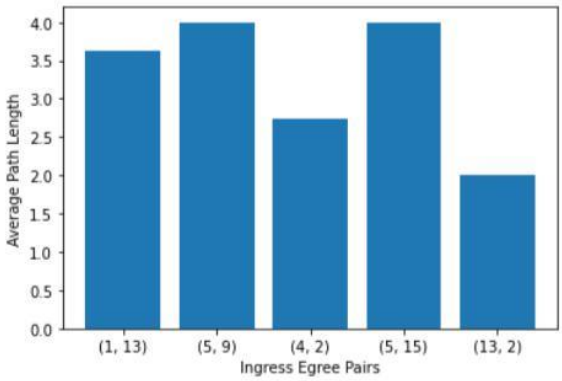

Fig 2.70 Average Path Length for WSP

Avg path length is little high, especially for IEP (4,2) averaging at 2.5. However, for IEP (13,2) the IEP has decreased to 2.0. But, this is because, IEP (13,2) had 500 rejections, and hence the average path length for this IEP is based on a small set of observations.

5.Max Flow Decrease

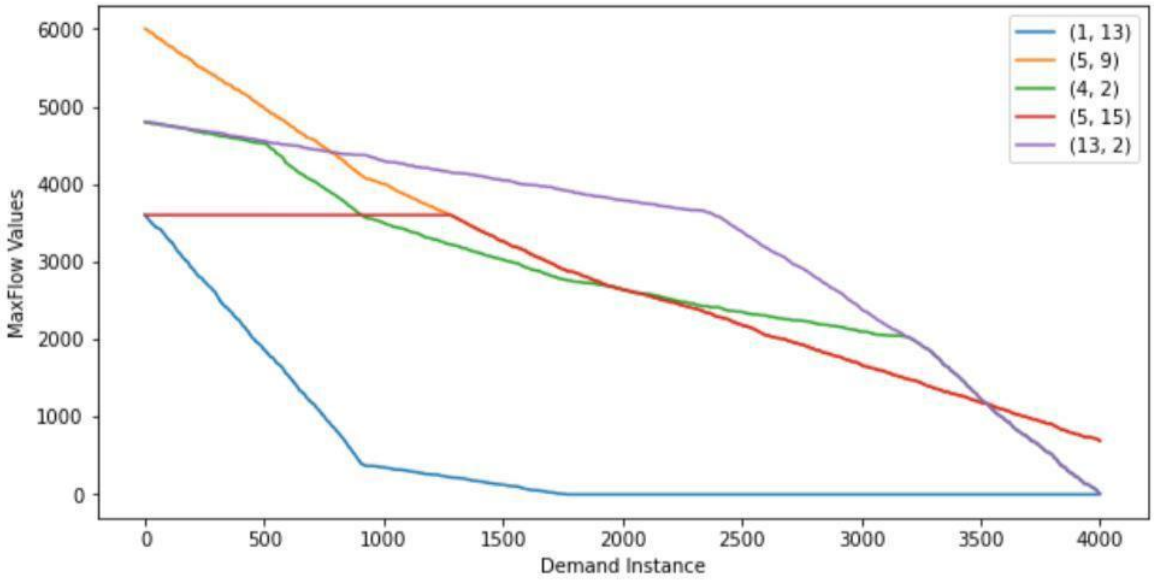

Fig 2.71 Max Flow Decrease for WSP

IEP (13,2) has very high decrease in maxflow value from instance 2500. IEP (1,13) s' flows are routed only initially and the maxflow value decreases to 0, by instance 1000.
This is because, (1,13) always goes in shortest path extinguishing the maxflow value.

2.13 SHORTEST WIDEST PATH(SWP)

Salient Features and Notes on SWP:

For using SWP as a routing algorithm, we first compute the widest paths between a given a source and destination having sufficient residual bandwidth >d on each link.
We first compute all paths b/w source and destination. Then, we choose paths such that their residual bandwidth is maximum according to below formula:

Residual Bandwidth of a path= min (Residual Bandwidth of l, where $l \in L_p$, where L_p is the set of all links in the chosen path.

$$\text{Residual Bandwidth (p)} = \min(\text{Residual Bandwidth (l)}) \quad - (1)$$

Then, if we get 2 or more paths, that have the same wideness, we then choose the shortest path.

INPUT:
A graph $G(N, L)$ and a set B of all residual link capacities. An ingress node a and an egress node b between which a flow of D units have to routed.
OUTPUT:
A route between a and b having a capacity of D units of bandwidth.
ALGORITHM:

1.Find all simple paths between source and destination

2.Using the wideness of a path formula given above, find the wideness of all paths found(Widest)

3.If there are 2 or more paths with same wideness, set all link weights =1 of the conflicting paths

4.Find shortest path of minimum hop length(Shortest)

5.Route the demand of D units from a to b along this path (Shortest Widest) and update the residual capacities.

1.Flow Decrease on Links

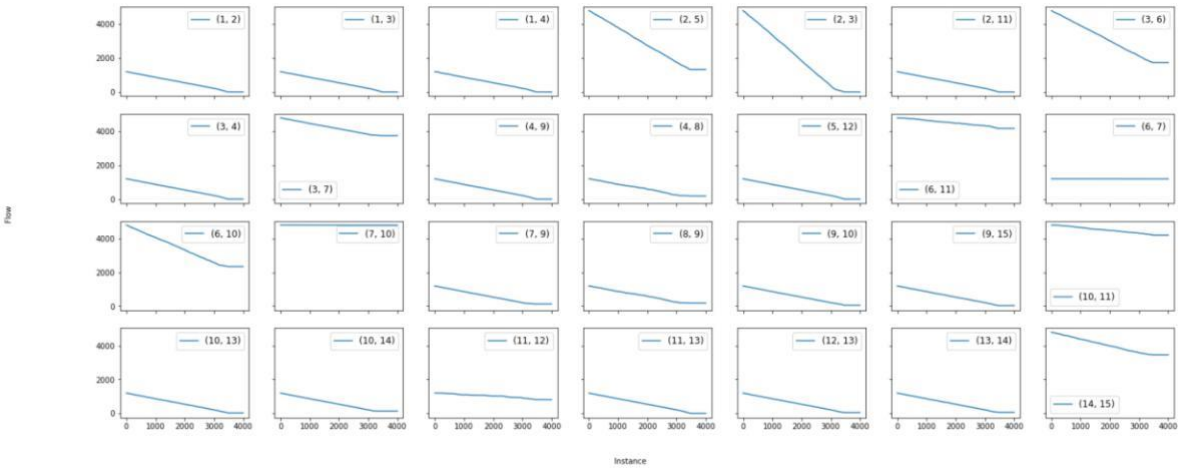

Fig 2.72 Flow Decrease on Links for SWP

2.Rejection Count

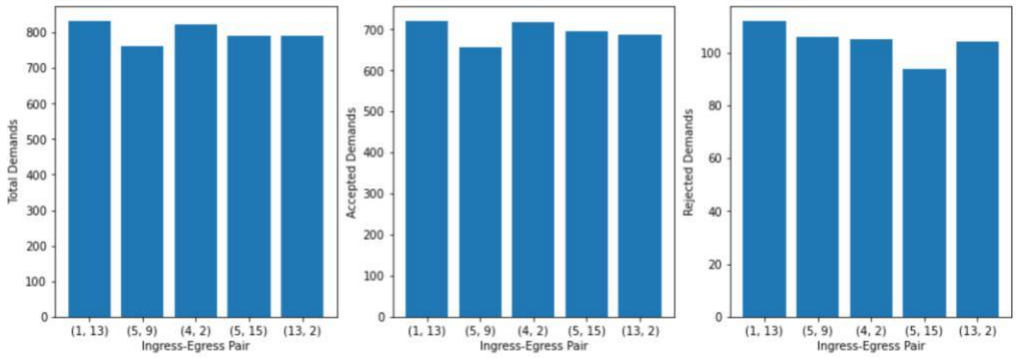

Fig 2.73 Rejection Count for SWP

Rejections are quite high, averaging at around 90 for each IEP.

3.Link Usage

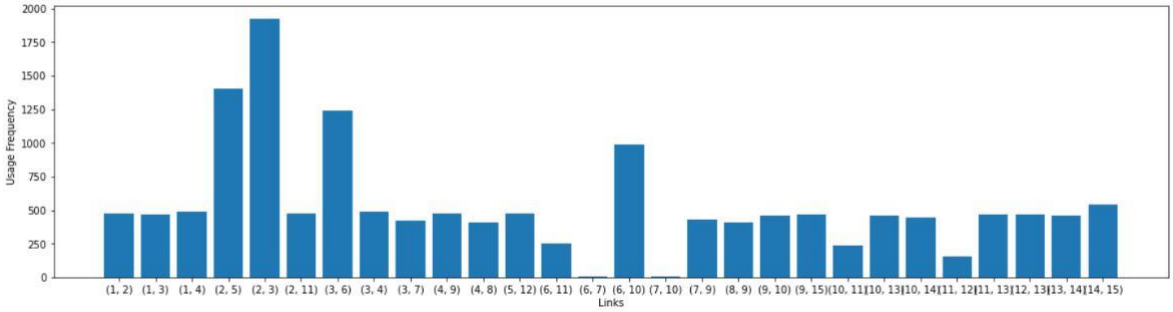

Fig 2.74 Link Usage for SWP

Links are equally used, in contrast to WSP.

Link Usage by IEP

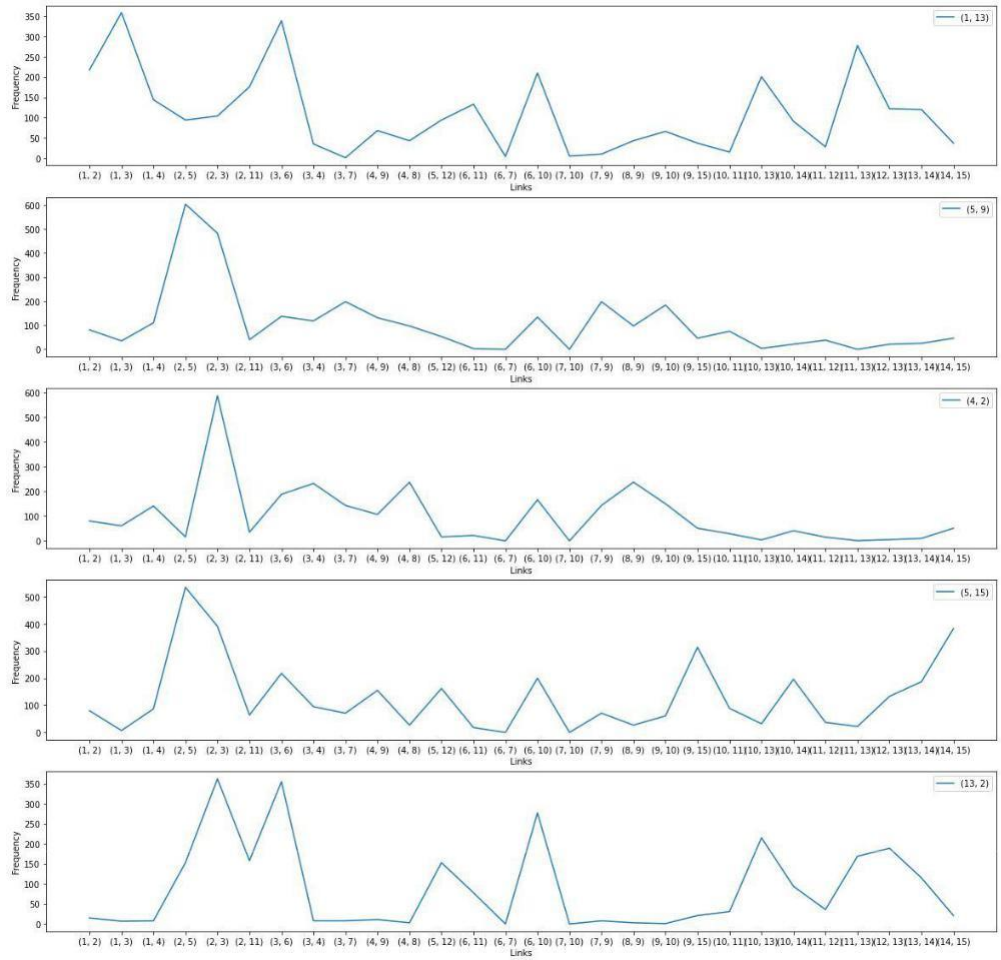

Fig 2.75 Link Usage by IEP for SWP

Here, IEP(4,2) uses links like(9,10),(8,9) etc.

4.Path Length

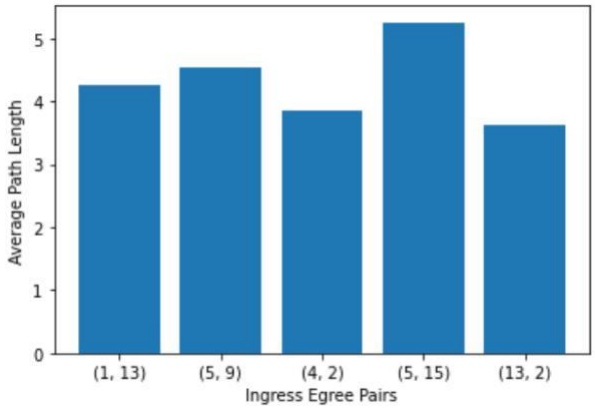

Fig 2.76 Average Path Length for SWP

Path length for all IEPs(including 4,2) is quite high, since its SWP, and the first criteria is to find the highest residual bandwidth or link capacity. In this algorithm, path length is a 2nd criteria and used only for conflicting cases of widest path. Hence, in general the avg path length is high.

5.Max Flow Decrease

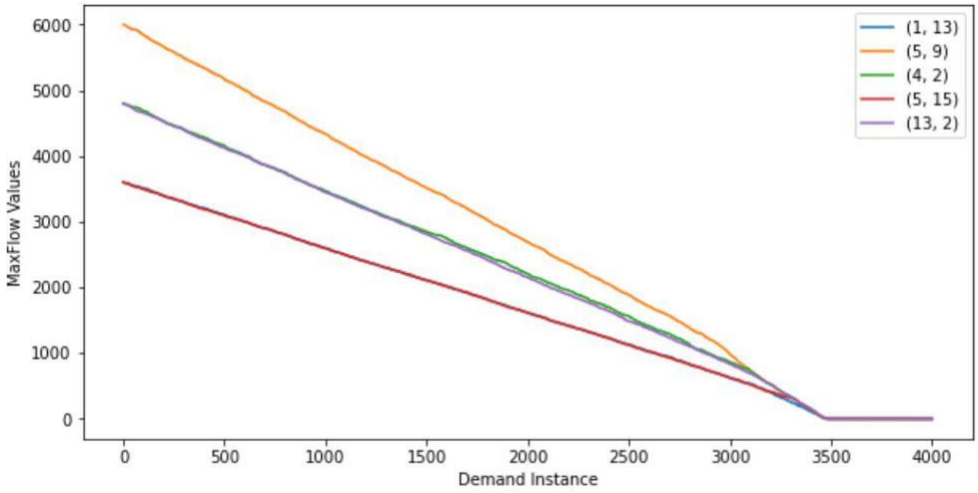

Fig 2.77 Max Flow Decrease for SWP

We see a steady decline in maxflow values from the start. At Instance 3500, many IEPs have reached maxflow =0. When compared to the previous algorithms, the max flow usually gradually decreases little slowly in the start and more rapidly in the end. However, due to a steady constant decline or slope, this has led to higher no. of rejections and almost no demand acceptance from instance 3500.

CHAPTER 3
COMPARATIVE ANALYSIS OF ALL ALGORITHMS

1.Rejection Count Comparison

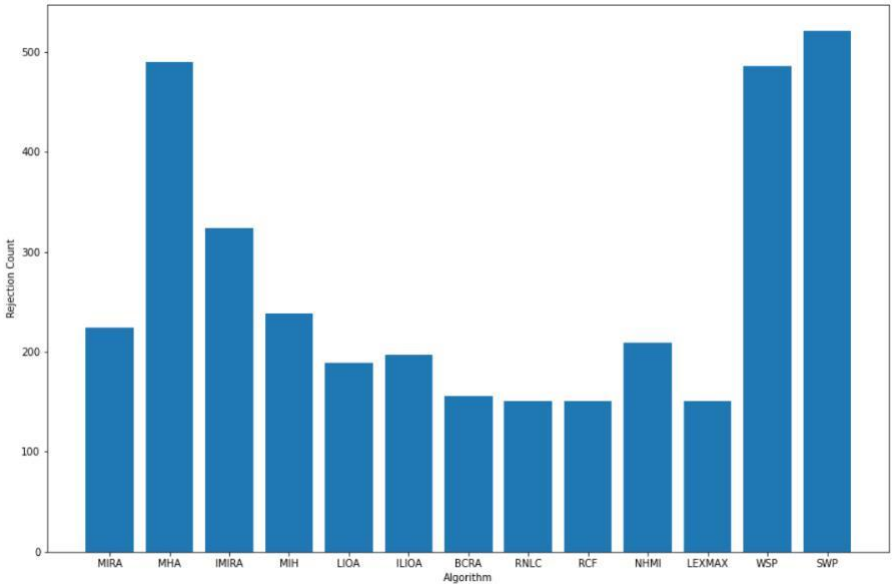

Fig 3.1 Rejection Count Comparison

As seen in previous studies, MHA,WSP,SWP perform in a very bad manner when compared to MIRA and other hybrid algorithms.

BCRA, RNLC, RCF, LEXMAX all perform quite well, with rejections around 150.

2. Path Length

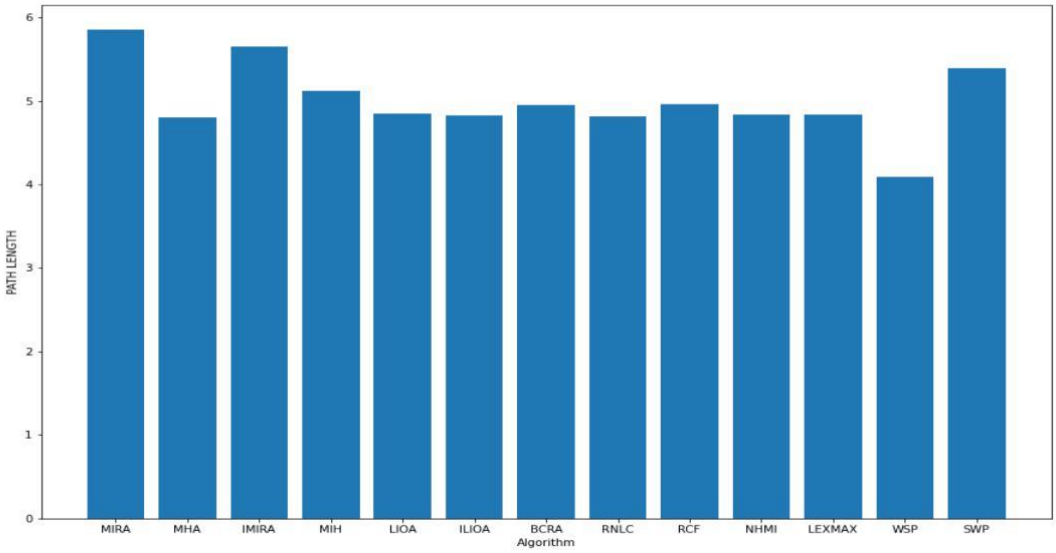

Fig 3.2 Average Path Length Comparison

We see that, MIRA, and IMIRA have quite high average path lengths around 6. WSP and MHA have the lowest average path lengths since they are mainly designed to have less hops.

3.Link Usage

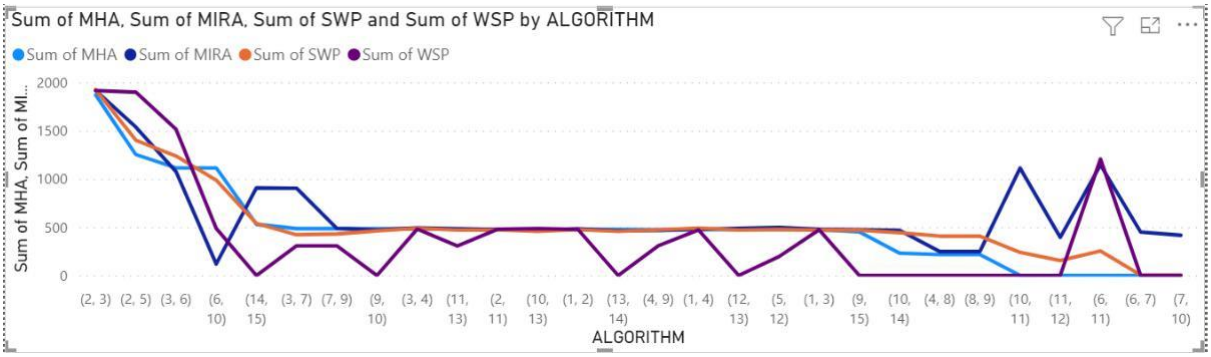

Fig 3.3 MHA, SWP, WSP, MIRA Comparison on Link Usage Comparison

As seen before, WSP does not use many links to a high extent. SWP on the other had uses almost all links and in an equal manner. MHA behaves in some way similar to WSP but it uses some links to a little higher extent than WSP.

4.Max Flow Decrease

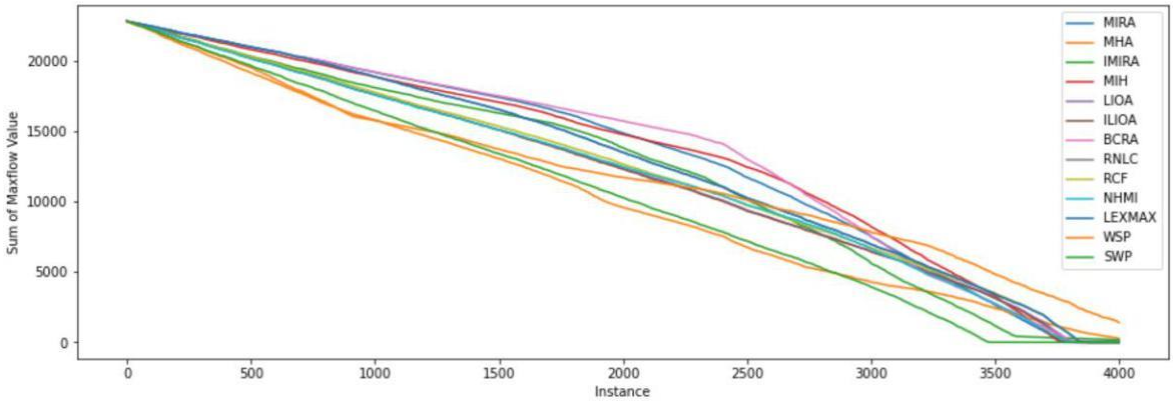

Fig 3.4 Max Flow Decrease Comparison

We see that, WSP, SWP, MHA perform quite badly and their maxflow values are at the bottom of the chart, meaning that the IEPs in these algorithms lose their maxflow quickly tending to more rejections. BCRA seems to perform the best in the overall sense, since it is at the top of the chart.

5.Link Flow Decrease Analysis for Selected Links

Link(1,2)

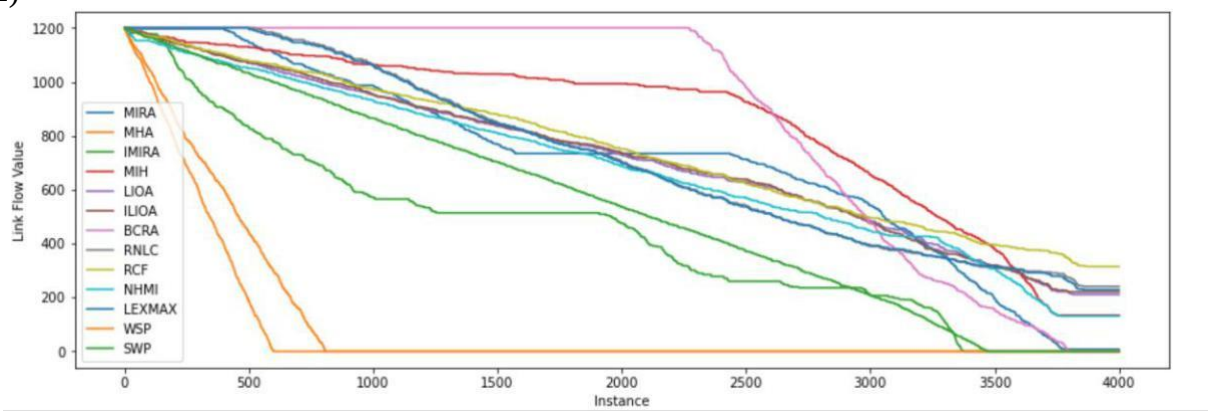

Fig 3.5 Link(1,2) Flow Decrease Comparison

We are able to see that, WSP, MHA use this link rashly, and hence maxflow value decreases heavily within the 500 iterations. Whereas, BCRA uses this judiciously, and starts using it only at instance 2500. Same goes with MIH. This is because, BCRA and MIH mainly work on the link utilization. If other links, have more residual bandwidth, the links with low residual bandwidth will not be disturbed.

Link(2,5)

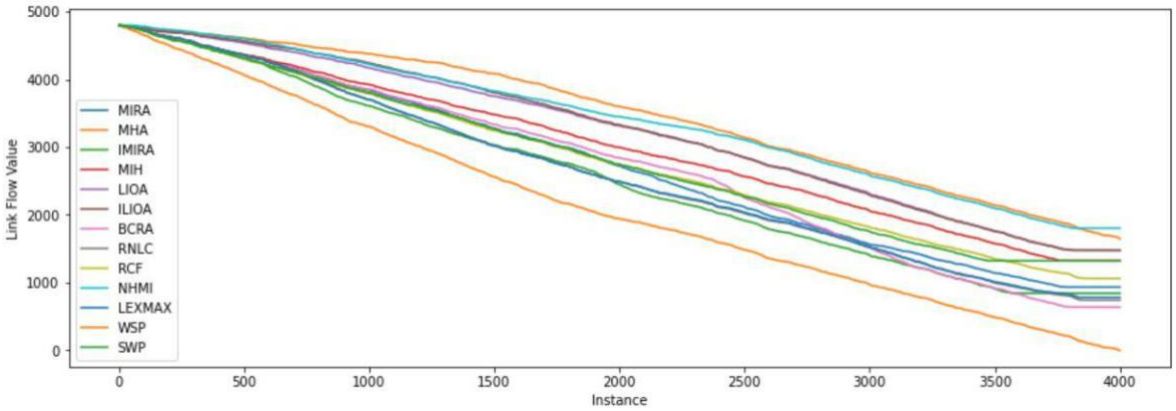

Fig 3.6 Link(2,5) Flow Decrease Comparison

This link seems to be used by all algorithms in a similar manner, more quickly used by WSP

Link(2,3)

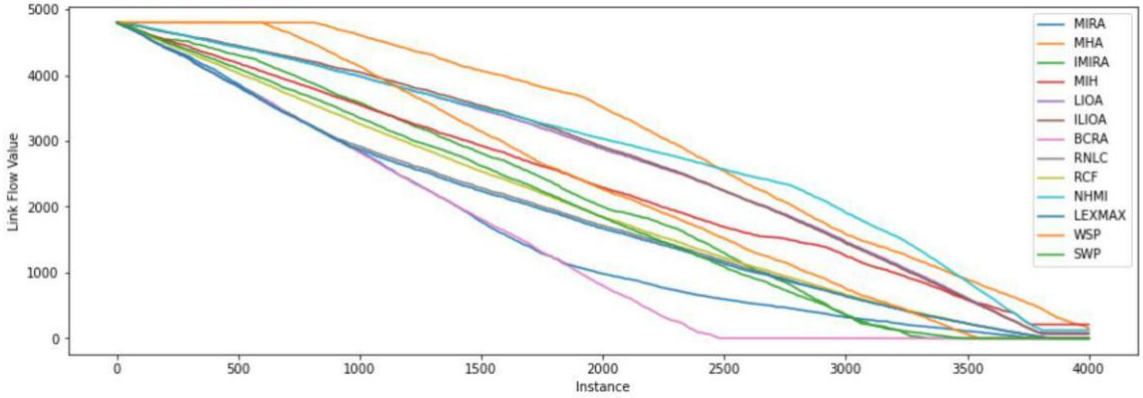

Fig 3.7 Link(2,3) Flow Decrease Comparison

BCRA likes to use links with high residual bandwidth and hence this link starts getting used from the first iteration itself.

Rejection Table

ALGORITHM	REJECTION
MIRA	224
MHA	490
IMIRA	324
MIH	238
LIOA	189
ILIOA	197
BCRA	156
RNLC	151
RCF	151
NHMI	209
LEXMAX	151
WSP	486
SWP	521

Table 3.1 Rejection Count for all algorithms

CHAPTER 4
SOURCE CODE AND OUTPUT

3.1 SOURCE CODE

```
#Source Code

#Graph Initialization
import networkx as nx
G=nx.Graph()
 for i in range(1,16):
    G.add_node(i)

G.add_edge(1,2)
G.add_edge(2,5)
G.add_edge(5,12)
G.add_edge(1,3)
G.add_edge(3,6)
G.add_edge(6,11)
G.add_edge(11,12)
G.add_edge(2,3)
G.add_edge(2,11)
G.add_edge(1,4)
G.add_edge(3,4)
G.add_edge(3,7)
G.add_edge(6,7)
G.add_edge(6,10)
G.add_edge(11,10)
G.add_edge(7,10)
G.add_edge(11,13)
G.add_edge(10,13)
G.add_edge(12,13)
G.add_edge(4,9)
G.add_edge(7,9)
G.add_edge(10,9)
G.add_edge(15,9)
G.add_edge(8,9)
G.add_edge(4,8)
G.add_edge(15,14)
G.add_edge(10,14)
G.add_edge(13,14)
for edge in G.edges():
    G[edge[0]][edge[1]]['capacity']=12*100
G[2][3]['capacity']=48*100
G[2][5]['capacity']=48*100
G[14][15]['capacity']=48*100
G[3][6]['capacity']=48*100
G[6][11]['capacity']=48*100
G[3][7]['capacity']=48*100
G[6][10]['capacity']=48*100
G[11][10]['capacity']=48*100
G[7][10]['capacity']=48*100
IEP=[[1,13],[5,9],[4,2],[5,15],[13,2]]
```

```
#Demand Initialization

# import random
# DEMAND=[]
# PAIR=[]
# for d in range(4000):
#     DEMAND.append(random.randint(1,4))
#     PAIR.append(random.randint(0,4))

import csv
with open(r"C:\Users\HARIHARAN SUBRAMANIA\Desktop\ACTUAL CN PROJ\demands.csv","r")
as csv:
    f=csv.read()
f=f[26:].split("\n")
for i in range(len(f)):
    f[i]=f[i].split(",")
DEMAND=[]
PAIR=[]
for i in range(4000):
    DEMAND.append(int(f[i][2]))
    pair=tuple((int(f[i][0]),int(f[i][1])))
    for k in range(len(IEP)):
        if tuple(IEP[k])==pair:
            PAIR.append(k)



#MIRA Algorithm

def subtracted_graph(G,flow_dict):
    for k in flow_dict.keys():
        for j in flow_dict[k].keys():
            if flow_dict[k][j]>0:
                G[k][j]['capacity']-=flow_dict[k][j]
    return G.copy()
def critical_checker(G,s,d):
    critical_links=[]
    new_graph=G.copy()
    for edge in G.edges():
        if new_graph[edge[0]][edge[1]]['capacity']==0:
            new_graph.remove_edge(edge[0],edge[1])
    for edge in G.edges():
        if G[edge[0]][edge[1]]['capacity']==0 and edge[1] not in nx.descendants(new_graph,s) and edge[0] not
in nx.descendants(new_graph,d) and nx.has_path(new_graph, edge[0],edge[1])==False:
            critical_links.append(edge)
    return critical_links
def critical_value_checker(G,IEP_without_ab,flow_dict_all):
    critical_links={}
    critical_links_overall={}

    for sd in IEP_without_ab:
        critical_links_overall[tuple((sd[0],sd[1]))]=[]
```

```
        critical_links={}
        for edge in G.edges():
            critical_links[tuple((edge[0],edge[1]))]=0
        flow_dict=flow_dict_all[tuple((sd[0],sd[1]))].copy()
        for k in flow_dict.keys():
            for j in flow_dict[k].keys():
                if flow_dict[k][j]>0:
                    try:
                        critical_links[tuple((k,j))]+=flow_dict[k][j]
                    except:
                        critical_links[tuple((j,k))]+=flow_dict[k][j]
        critical_links_overall[tuple((sd[0],sd[1]))]=critical_links.copy()
    return critical_links_overall
def non_vital_flow_contributor(G,IEP_without_ab,flow_dict_all):
    critical_links={}
    for edge in G.edges():
        critical_links[tuple((edge[0],edge[1]))]=0

    for sd in IEP_without_ab:
        flow_dict=flow_dict_all[tuple((sd[0],sd[1]))].copy()
        for k in flow_dict.keys():
            for j in flow_dict[k].keys():
                if flow_dict[k][j]>0:
                    try:
                        critical_links[tuple((k,j))]+=flow_dict[k][j]
                    except:
                        critical_links[tuple((j,k))]+=flow_dict[k][j]
    return critical_links
def MIRA(G,ab,IEP,D):
    from networkx.algorithms.flow import preflow_push
    #1. Compute Maximum flow values for all (s,d) in P excluding (a,b)
    max_flow_calc={}
    IEP_without_ab=IEP.copy()
    IEP_without_ab.remove(ab)
    aaaa=[]
    max_flow_dict={}
    for sd in IEP_without_ab:
        #print(nx.maximum_flow(G,sd[0],sd[1])[0])
        max_flow_calc[tuple(sd)]=nx.maximum_flow(G,sd[0],sd[1])[0]
        max_flow_dict[tuple(sd)]=nx.maximum_flow(G,sd[0],sd[1])[1]
    #2. Compute Critical links C_sd
    critical_links={}
    for sd in IEP_without_ab:

        critical_links[tuple(sd)]=[]
        subt_graph=nx.Graph()
        subt_graph=subtracted_graph(G.copy(),max_flow_dict[tuple(sd)])
        critical_links[tuple(sd)]=critical_checker(subt_graph.copy(),sd[0],sd[1])
    flow_contributor=critical_value_checker(G.copy(),IEP_without_ab,max_flow_dict)
    inverse_max_flow_calc={}

    for sd in IEP_without_ab:
        if max_flow_calc[tuple(sd)]==0:
            max_flow_calc[tuple(sd)]=0.000000000001
```

```
        inverse_max_flow_calc[tuple(sd)]=1/max_flow_calc[tuple(sd)]
    #3 Compute the weight of links w(l)
    w={}
    curr_sum=0
    c_length=[]
    for c in critical_links.keys():
        c_length.append(len(critical_links[c]))
    for edge in G.edges:
        curr_sum=0
        for keys,values in critical_links.items():
            if edge in critical_links[keys]:
                curr_sum+=inverse_max_flow_calc[keys]#(1/flow_contributor[keys][edge])
        w[edge]=curr_sum
    non_vital_edge_flow_contributor=non_vital_flow_contributor(G.copy(),
    IEP_without_ab,max_flow_dict) import random as random
    #print(w)
    for keys,values in w.items():
        if w[keys]==0:
            if non_vital_edge_flow_contributor[keys]!=0:
                w[keys]=1/(non_vital_edge_flow_contributor[keys]*1000)
            else:
                w[keys]=0
    #Later on whoseover w(l)=0, choose the minimum no of
    hops #4.Eliminate <D edges and form reduced network
    new_G=nx.Graph()
    for node in G.nodes():
        new_G.add_node(node)
    for edge in G.edges():
        if G[edge[0]][edge[1]]['capacity']-D>=0:
            new_G.add_edge(edge[0],edge[1])
            new_G[edge[0]][edge[1]]['capacity']=w[edge]/G[edge[0]][edge[1]]['capacity']
    try:
        shortest_path=nx.shortest_path(new_G,ab[0],ab[1],weight='capacity')
    except:
        print("No Path Found from",ab[0],"to",ab[1])
        return G.copy(),0,0
    for i in range(len(shortest_path)-1):
        G[shortest_path[i]][shortest_path[i+1]]['capacity']-=D
    format_modified_shortest_path=[]
    for i in range(len(shortest_path)-1):
        l=[shortest_path[i],shortest_path[i+1]]
        format_modified_shortest_path.append(l)

    return G.copy(),aaaa,format_modified_shortest_path


#MHA Algorithm

def MHA(G,ab,D):
    G_temp=G.copy()
    G_temp_edge_list=list(G_temp.edges())
    for edge in G_temp_edge_list:
        if G_temp.get_edge_data(edge[0],edge[1])['capacity']<D:
            G_temp.remove_edge(edge[0],edge[1])
```

```
    for edge in G_temp.edges():
        G_temp[edge[0]][edge[1]]['capacity']=1
    try:
        shortest_path=nx.shortest_path(G_temp,ab[0],ab[1],weight='capacity')
    except:
        print("No Path Found from",ab[0],"to",ab[1])
        return G.copy(),0
    for i in range(len(shortest_path)-1):
        G[shortest_path[i]][shortest_path[i+1]]['capacity']-=D
    format_modified_shortest_path=[]
    for i in range(len(shortest_path)-1):
        l=[shortest_path[i],shortest_path[i+1]]
        format_modified_shortest_path.append(l)
#     for edge in G.edges():
#         print(G.get_edge_data(edge[0],edge[1])['capacity'])
    return G.copy(),format_modified_shortest_path


#IMIRA Algorithm


def how_many_paths(G,sd,ij):
    all_paths=list(nx.all_simple_paths(G,sd[0],sd[1]))
    usage=0
    for path in all_paths:
        for i in range(len(path)-1):
            if (path[i]==ij[0] and path[i+1]==ij[1]) or (path[i]==ij[1] and path[i+1]==ij[0]):
                usage+=1
    return usage/len(all_paths)

def MIRA_1(G,ab,IEP,D):
    from networkx.algorithms.flow import preflow_push
    #1. Compute Maximum flow values for all (s,d) in P excluding (a,b)
    max_flow_calc={}
    IEP_without_ab=IEP.copy()
    IEP_without_ab.remove(ab)
    aaaa=[]
    max_flow_dict={}
    for sd in IEP_without_ab:
        #print(nx.maximum_flow(G,sd[0],sd[1])[0])
        max_flow_calc[tuple(sd)]=nx.maximum_flow(G,sd[0],sd[1])[0]
        max_flow_dict[tuple(sd)]=nx.maximum_flow(G,sd[0],sd[1])[1]
    #2. Compute Critical links C_sd
    link_weight={}
    for edge in G.edges():
        link_weight[edge]=0
    for edge in G.edges():
        for sd in IEP_without_ab:

maxflow_ratio=max_flow_dict[tuple(sd)][edge[0]][edge[1]]/((max_flow_calc[tuple(sd)]*G[edge[0]][edge[1]]
['capacity'])+0.00000000000000000000001)
            link_weight[edge]+=maxflow_ratio
    new_G=nx.Graph()
    for node in G.nodes():
```

```
        new_G.add_node(node)
    for edge in G.edges():
        if G[edge[0]][edge[1]]['capacity']-D>=0:
            new_G.add_edge(edge[0],edge[1])
            new_G[edge[0]][edge[1]]['capacity']=link_weight[edge]
    try:
        shortest_path=nx.shortest_path(new_G,ab[0],ab[1],weight='capacity')
    except:
        print("No Path Found from",ab[0],"to",ab[1])
        return G.copy(),0,0

    for i in range(len(shortest_path)-1):
        G[shortest_path[i]][shortest_path[i+1]]['capacity']-=D
    format_modified_shortest_path=[]
    for i in range(len(shortest_path)-1):
        l=[shortest_path[i],shortest_path[i+1]]
        format_modified_shortest_path.append(l)
#     for edge in G.edges():
#         print(G.get_edge_data(edge[0],edge[1])['capacity'])
    return G.copy(),aaaa,format_modified_shortest_path

#MIH Algorithm

def MIH(G,ab,IEP,D,fij,entire_cap,T1=8,T2=4):
    from networkx.algorithms.flow import preflow_push
    #1. Compute Maximum flow values for all (s,d) in P excluding (a,b)
    max_flow_calc={}
    max_flow_dict={}
    IEP_without_ab=IEP.copy()
    IEP_without_ab.remove(ab)
    aaaa=[]
    for sd in IEP_without_ab:
        #print(nx.maximum_flow(G,sd[0],sd[1])[0])
        max_flow_calc[tuple(sd)]=nx.maximum_flow(G,sd[0],sd[1])[0]
        max_flow_dict[tuple(sd)]=nx.maximum_flow(G,sd[0],sd[1])[1]
    #2. Compute Critical links C_sd
    critical_links={}
#     for sd in IEP_without_ab:
#         critical_links[tuple(sd)]=[]
#         current_max_flow_graph=preflow_push(G,sd[0],sd[1])
#         new_graph=nx.Graph()
#         for n_node in current_max_flow_graph.nodes:
#             new_graph.add_node(n_node)
#         for n_edge in current_max_flow_graph.edges:
#             #print(current_max_flow_graph[n_edge[0]][n_edge[1]]['flow'])
#             if current_max_flow_graph[n_edge[0]][n_edge[1]]['flow']>0
or current_max_flow_graph[n_edge[1]][n_edge[0]]['flow']>0:
#                 new_graph.add_edge(n_edge[0],n_edge[1])
#         for edge in G.edges:
#             if current_max_flow_graph[edge[0]][edge[1]]['flow']<=0 and
nx.has_path(new_graph,edge[0],edge[1])==False and edge[0] not in nx.descendants(new_graph,sd[1])
and edge[1] not in nx.descendants(new_graph,sd[0]):
#                 critical_links[tuple(sd)].append(edge)
#         aaaa.append(new_graph)
```

```
    for sd in IEP_without_ab:

        critical_links[tuple(sd)]=[]
        subt_graph=nx.Graph()
        subt_graph=subtracted_graph(G.copy(),max_flow_dict[tuple(sd)])
        critical_links[tuple(sd)]=critical_checker(subt_graph.copy(),sd[0],sd[1])
    inverse_max_flow_calc={}
    for sd in IEP_without_ab:
        inverse_max_flow_calc[tuple(sd)]=1/(0.00000000000001+max_flow_calc[tuple(sd)])
    #3 Compute Criticality of each link g(ij)
    criticality={}
    for edge in G.edges():
        criticality[edge]=0
    for c in critical_links.keys():
        for edge in critical_links[c]:
            criticality[edge]=criticality.get(edge,0)+1
    #4 Compute max link criticality
    beta=-1
    for k in criticality.keys():
        if criticality[k]>beta:
            beta=criticality[k]
    #5 Compute link utilization fij
    link_utilization={}
    for edge in G.edges():
        link_utilization[edge]=fij[edge]/entire_cap[edge] #D/G[edge[0]][edge[1]]['capacity'] #6
    Max link utilization alpha
    alpha=-1
    for k in link_utilization.keys():
        if link_utilization[k]>alpha:
            alpha=link_utilization[k]



    #7 Compute the weight of links w(l)
    w={}
    for edge in G.edges():
        w[edge]=((fij[edge]+D)/entire_cap[edge])+T1*max(0,((fij[edge]+D)/entire_cap[edge])-
alpha)+T2*(criticality[edge]/beta)


    #8.Eliminate <D edges and form reduced network
    new_G=nx.Graph()
    for node in G.nodes():
        new_G.add_node(node)
    for edge in G.edges():
        if G[edge[0]][edge[1]]['capacity']-D>0:
            new_G.add_edge(edge[0],edge[1])
            new_G[edge[0]][edge[1]]['capacity']=w[edge]
    #9 Use Djikstra algo for shortest path
    try:
        shortest_path=nx.shortest_path(new_G,ab[0],ab[1],weight='capacity')
    except:
        #print("No Path Found from",ab[0],"to",ab[1])
        return G.copy(),0,0
```

```
    for i in range(len(shortest_path)-1):
        G[shortest_path[i]][shortest_path[i+1]]['capacity']-=D
        try:
            fij[tuple((shortest_path[i],shortest_path[i+1]))]+=D
        except:
            fij[tuple((shortest_path[i+1],shortest_path[i]))]+=D
    format_modified_shortest_path=[]
    for i in range(len(shortest_path)-1):
        l=[shortest_path[i],shortest_path[i+1]]
        format_modified_shortest_path.append(l)
#     for edge in G.edges():
#         print(G.get_edge_data(edge[0],edge[1])['capacity'])
    return G.copy(),aaaa,format_modified_shortest_path


#LIOA Algorithm

def LIOA(G, ab, IEP,D,capacity_dict,demand_dict,alpha):
    aaaa=[]

    # 1. Set weight of all
    links w={}
    for edge in G.edges():
        w[edge]=0
    for edge in G.edges():
        if G[edge[0]][edge[1]]['capacity']!=0:
            w[edge]=((demand_dict[edge])**alpha)/((G[edge[0]][edge[1]]['capacity'])**(1-alpha))
        else:
            w[edge]=((demand_dict[edge])**alpha)/0.00000000000001
    for edge in G.edges():
        if G[edge[0]][edge[1]]['capacity']<=
            D: w[edge]=10**30

    #2.Eliminate <D edges and form reduced network
    new_G=nx.Graph()
    for node in G.nodes():
        new_G.add_node(node)
    for edge in G.edges():
        if G[edge[0]][edge[1]]['capacity']-D>=0:
            new_G.add_edge(edge[0],edge[1])
            new_G[edge[0]][edge[1]]['capacity']=w[edge]
    try:
        shortest_path=nx.shortest_path(new_G,ab[0],ab[1],weight='capacity')
    except:
        #print("No Path Found from",ab[0],"to",ab[1])
        return G.copy(),0,0

    for i in range(len(shortest_path)-1):
        G[shortest_path[i]][shortest_path[i+1]]['capacity']-=D
        try:
            demand_dict[tuple((shortest_path[i],shortest_path[i+1]))]+=1
        except:
            demand_dict[tuple((shortest_path[i+1],shortest_path[i]))]+=1
    format_modified_shortest_path=[]
```

```
    for i in range(len(shortest_path)-1):
        l=[shortest_path[i],shortest_path[i+1]]
        format_modified_shortest_path.append(l)
#     for edge in G.edges():
#         print(G.get_edge_data(edge[0],edge[1])['capacity'])
    return G.copy(),aaaa,format_modified_shortest_path


#ILIOA Algorithm

def ILIOA(G, ab, IEP,D,capacity_dict,demand_dict,alpha,beta):
    aaaa=[]

    # 1. Set weight of all
    links w={}

    for edge in G.edges():
        w[edge]=0
    for edge in G.edges():

        if G[edge[0]][edge[1]]['capacity']!=0:
            ul=1-(G[edge[0]][edge[1]]['capacity']/capacity_dict[edge])
            w[edge]=(1-ul)*((demand_dict[edge])**beta)/(capacity_dict[edge]**(1-beta))+
ul*((demand_dict[edge]**alpha)/(G[edge[0]][edge[1]]['capacity'])**(1-alpha))
        else:
            w[edge]=((demand_dict[edge])**alpha)/0.00000000000001
    for edge in G.edges():
        if G[edge[0]][edge[1]]['capacity']<= D:
            w[edge]=10**30

    #2.Eliminate <D edges and form reduced network
    new_G=nx.Graph()
    for node in G.nodes():
        new_G.add_node(node)
    for edge in G.edges():
        if G[edge[0]][edge[1]]['capacity']-D>=0:
            new_G.add_edge(edge[0],edge[1])
            new_G[edge[0]][edge[1]]['capacity']=w[edge]
    try:
        shortest_path=nx.shortest_path(new_G,ab[0],ab[1],weight='capacity')
    except:
        #print("No Path Found from",ab[0],"to",ab[1])
        return G.copy(),0,0

    for i in range(len(shortest_path)-1):
        G[shortest_path[i]][shortest_path[i+1]]['capacity']-=D
        try:
            demand_dict[tuple((shortest_path[i],shortest_path[i+1]))]+=1
        except:
            demand_dict[tuple((shortest_path[i+1],shortest_path[i]))]+=1
    format_modified_shortest_path=[]
    for i in range(len(shortest_path)-1):
        l=[shortest_path[i],shortest_path[i+1]]
        format_modified_shortest_path.append(l)
```

```
#     for edge in G.edges():
#         print(G.get_edge_data(edge[0],edge[1])['capacity'])
    return G.copy(),aaaa,format_modified_shortest_path



#BCRA Algorithm

def BCRA(G, ab, IEP,D,capacity_dict):
    aaaa=[]

    # 1. Find Load on
    links load={}
    for edge in G.edges():
        load[edge]=G[edge[0]][edge[1]]['capacity']/capacity_dict[edge]
    # 2.Find cost of links
    cost={}
    for edge in G.edges():
        cost[edge]=10**8/(capacity_dict[edge])
    # 3. Find weight of
    edge w={}
    for edge in G.edges():
        w[edge]=0
    for edge in G.edges():
        w[edge]=cost[edge]+load[edge]+1

    #2.Eliminate <D edges and form reduced network
    new_G=nx.Graph()
    for node in G.nodes():
        new_G.add_node(node)
    for edge in G.edges():
        if G[edge[0]][edge[1]]['capacity']-D>=0:
            new_G.add_edge(edge[0],edge[1])
            new_G[edge[0]][edge[1]]['capacity']=w[edge]
    try:
        shortest_path=nx.shortest_path(new_G,ab[0],ab[1],weight='capacity')
    except:
        print("No Path Found from",ab[0],"to",ab[1])
        return G.copy(),0,0

    for i in range(len(shortest_path)-1):
        G[shortest_path[i]][shortest_path[i+1]]['capacity']-=D
    format_modified_shortest_path=[]
    for i in range(len(shortest_path)-1):
        l=[shortest_path[i],shortest_path[i+1]]
        format_modified_shortest_path.append(l)
#     for edge in G.edges():
#         print(G.get_edge_data(edge[0],edge[1])['capacity'])
    return G.copy(),aaaa,format_modified_shortest_path


#RNLC Algorithm
def RNLC(G, ab, IEP,D,C):
```

```
    aaaa=[]
    # 1. Set weight of all links
    current_residual_capacity=0
    for edge in G.edges():
        current_residual_capacity+=G[edge[0]][edge[1]]['capacity']
    w={}
    for edge in G.edges():
        w[edge]=0
    for edge in G.edges():
        if G[edge[0]][edge[1]]['capacity']!=0:
            w[edge]=(current_residual_capacity/G[edge[0]][edge[1]]['capacity'])+C

    #2.Eliminate <D edges and form reduced network
    new_G=nx.Graph()
    for node in G.nodes():
        new_G.add_node(node)
    for edge in G.edges():
        if G[edge[0]][edge[1]]['capacity']-D>=0:
            new_G.add_edge(edge[0],edge[1])
            new_G[edge[0]][edge[1]]['capacity']=w[edge]
    try:
        shortest_path=nx.shortest_path(new_G,ab[0],ab[1],weight='capacity')
    except:
        #print("No Path Found from",ab[0],"to",ab[1])
        return G.copy(),0,0

    for i in range(len(shortest_path)-1):
        G[shortest_path[i]][shortest_path[i+1]]['capacity']-=D
    format_modified_shortest_path=[]
    for i in range(len(shortest_path)-1):
        l=[shortest_path[i],shortest_path[i+1]]
        format_modified_shortest_path.append(l)
#     for edge in G.edges():
#         print(G.get_edge_data(edge[0],edge[1])['capacity'])
    return G.copy(),aaaa,format_modified_shortest_path


#MAX RC MIN F

def RCF(G, ab, IEP,D,capacity_dict,demand_dict):
    aaaa=[]

    # 1. Set weight of all
    links w={}

    for edge in G.edges():
        w[edge]=0
    for edge in G.edges():

w[edge]=demand_dict[edge]/((capacity_dict[edge]*G[edge[0]][edge[1]]['capacity'])+0.0000000000000001)
    #2.Eliminate <D edges and form reduced network
    new_G=nx.Graph()
    for node in G.nodes():
        new_G.add_node(node)
```

```
    for edge in G.edges():
        if G[edge[0]][edge[1]]['capacity']-D>=0:
            new_G.add_edge(edge[0],edge[1])
            new_G[edge[0]][edge[1]]['capacity']=w[edge]
    try:
        shortest_path=nx.shortest_path(new_G,ab[0],ab[1],weight='capacity')
    except:
        print("No Path Found from",ab[0],"to",ab[1])
        return G.copy(),0,0

    for i in range(len(shortest_path)-1):
        G[shortest_path[i]][shortest_path[i+1]]['capacity']-=D
        try:
            demand_dict[tuple((shortest_path[i],shortest_path[i+1]))]+=1
        except:
            demand_dict[tuple((shortest_path[i+1],shortest_path[i]))]+=1
    format_modified_shortest_path=[]
    for i in range(len(shortest_path)-1):
        l=[shortest_path[i],shortest_path[i+1]]
        format_modified_shortest_path.append(l)
#     for edge in G.edges():
#         print(G.get_edge_data(edge[0],edge[1])['capacity'])
    return G.copy(),aaaa,format_modified_shortest_path




#NMIH Algorithm

def critical_value_checker(G,IEP_without_ab,flow_dict_all):
    critical_links={}
    critical_links_overall={}

    for sd in IEP_without_ab:
        critical_links_overall[tuple((sd[0],sd[1]))]=[]
        critical_links={}
        for edge in G.edges():
            critical_links[tuple((edge[0],edge[1]))]=0
        flow_dict=flow_dict_all[tuple((sd[0],sd[1]))].copy()
        for k in flow_dict.keys():
            for j in flow_dict[k].keys():
                if flow_dict[k][j]>0:
                    try:
                        critical_links[tuple((k,j))]+=flow_dict[k][j]
                    except:
                        critical_links[tuple((j,k))]+=flow_dict[k][j]
        critical_links_overall[tuple((sd[0],sd[1]))]=critical_links.copy()
    return critical_links_overall


def NMIH(G,ab,IEP,D,entire_cap,T1=8,T2=4):
    from networkx.algorithms.flow import preflow_push
    #1. Compute Maximum flow values for all (s,d) in P excluding (a,b)
    max_flow_calc={}
```

```
    max_flow_dict={}
    IEP_without_ab=IEP.copy()
    IEP_without_ab.remove(ab)
    aaaa=[]
    for sd in IEP:
        #print(nx.maximum_flow(G,sd[0],sd[1])[0])
        max_flow_calc[tuple(sd)]=nx.maximum_flow(G,sd[0],sd[1])[0]
        max_flow_dict[tuple(sd)]=nx.maximum_flow(G,sd[0],sd[1])[1]
    #2. Compute Critical links C_sd
    critical_values={}
    critical_values=critical_value_checker(G.copy(),IEP,max_flow_dict)
    alphasd={}
    for sd in IEP:
        alphasd[tuple(sd)]=1/max_flow_calc[tuple(sd)]


    w={}
    for edge in G.edges():
        w[edge]=T1+T2*((entire_cap[edge]-G[edge[0]][edge[1]]['capacity'])/entire_cap[edge])
        s=0
        for sd in IEP_without_ab:
            s+=(alphasd[tuple(sd)]*critical_values[tuple(sd)][edge])
        s/(D+(alphasd[tuple(ab)]*critical_values[tuple(ab)][edge]))
        w[edge]+=s


    #8.Eliminate <D edges and form reduced network
    new_G=nx.Graph()
    for node in G.nodes():
        new_G.add_node(node)
    for edge in G.edges():
        if G[edge[0]][edge[1]]['capacity']-D>0:
            new_G.add_edge(edge[0],edge[1])
            new_G[edge[0]][edge[1]]['capacity']=w[edge]
    #9 Use Djikstra algo for shortest path
    try:
        shortest_path=nx.shortest_path(new_G,ab[0],ab[1],weight='capacity')
    except:
        #print("No Path Found from",ab[0],"to",ab[1])
        return G.copy(),0,0
    for i in range(len(shortest_path)-1):
        G[shortest_path[i]][shortest_path[i+1]]['capacity']-=D
    format_modified_shortest_path=[]
    for i in range(len(shortest_path)-1):
        l=[shortest_path[i],shortest_path[i+1]]
        format_modified_shortest_path.append(l)
#     for edge in G.edges():
#         print(G.get_edge_data(edge[0],edge[1])['capacity'])
    return G.copy(),aaaa,format_modified_shortest_path


#LEX MAX Algorithm

def subtracted_graph(G,flow_dict):
```

```
    for k in flow_dict.keys():
        for j in flow_dict[k].keys():
            if flow_dict[k][j]>0:
                G[k][j]['capacity']-=flow_dict[k][j]
    return G.copy()
def critical_checker_LEX(G,s,d,delta):
    critical_links=[]
    new_graph=G.copy()
    for edge in G.edges():
        if new_graph[edge[0]][edge[1]]['capacity']==0:
            new_graph.remove_edge(edge[0],edge[1])
    for edge in G.edges():
        if G[edge[0]][edge[1]]['capacity']<delta:
            new_new_graph=nx.Graph()
            for node in new_graph.nodes():
                new_new_graph.add_node(node)
            for edge in new_graph.edges():
                if delta-new_graph[edge[0]][edge[1]]['capacity']>=0:
                    new_new_graph.add_edge(edge[0],edge[1])
            if nx.has_path(new_graph, edge[0],edge[1])==False:
                critical_links.append(edge)
    return critical_links
def critical_value_checker(G,IEP_without_ab,flow_dict_all):
    critical_links={}
    critical_links_overall={}

    for sd in IEP_without_ab:
        critical_links_overall[tuple((sd[0],sd[1]))]=[]
        critical_links={}
        for edge in G.edges():
            critical_links[tuple((edge[0],edge[1]))]=0
        flow_dict=flow_dict_all[tuple((sd[0],sd[1]))].copy()
        for k in flow_dict.keys():
            for j in flow_dict[k].keys():
                if flow_dict[k][j]>0:
                    try:
                        critical_links[tuple((k,j))]+=flow_dict[k][j]
                    except:
                        critical_links[tuple((j,k))]+=flow_dict[k][j]
        critical_links_overall[tuple((sd[0],sd[1]))]=critical_links.copy()
    return critical_links_overall
def non_vital_flow_contributor(G,IEP_without_ab,flow_dict_all):
    critical_links={}
    for edge in G.edges():
        critical_links[tuple((edge[0],edge[1]))]=0

    for sd in IEP_without_ab:
        flow_dict=flow_dict_all[tuple((sd[0],sd[1]))].copy()
        for k in flow_dict.keys():
            for j in flow_dict[k].keys():
                if flow_dict[k][j]>0:
                    try:
                        critical_links[tuple((k,j))]+=flow_dict[k][j]
                    except:
```

```
                critical_links[tuple((j,k))]+=flow_dict[k][j]
    return critical_links

def LEXMAX(G,ab,IEP,D,delta):
    from networkx.algorithms.flow import preflow_push
    #1. Compute Maximum flow values for all (s,d) in P excluding (a,b)
    max_flow_calc={}
    IEP_without_ab=IEP.copy()
    IEP_without_ab.remove(ab)
    aaaa=[]
    max_flow_dict={}
    for sd in IEP:
        #print(nx.maximum_flow(G,sd[0],sd[1])[0])
        max_flow_calc[tuple(sd)]=nx.maximum_flow(G,sd[0],sd[1])[0]
        max_flow_dict[tuple(sd)]=nx.maximum_flow(G,sd[0],sd[1])[1]
    #2. Compute Critical links C_sd
    critical_links={}
    for sd in IEP:

        critical_links[tuple(sd)]=[]
        subt_graph=nx.Graph()
        subt_graph=subtracted_graph(G.copy(),max_flow_dict[tuple(sd)])
        critical_links[tuple(sd)]=critical_checker_LEX(subt_graph.copy(),sd[0],sd[1],delta)
    flow_contributor=critical_value_checker(G.copy(),IEP_without_ab,max_flow_dict)


    #3 Order IEP via maxflow values
    max_flow_sorted=sorted(max_flow_calc.items(), key=lambda kv:(kv[1]))
    inverse_max_flow_calc={}
    for sd in IEP_without_ab:
        if max_flow_calc[tuple(sd)]==0:
            max_flow_calc[tuple(sd)]=0.000000000001
        inverse_max_flow_calc[tuple(sd)]=1/max_flow_calc[tuple(sd)]
    #3 Compute the weight of links w(l)
    w={}
    curr_sum=0
    c_length=[]
    for c in critical_links.keys():
        c_length.append(len(critical_links[c]))
    p=0
    for edge in G.edges:
        curr_sum=0
        p=len(IEP)
        i=0
        for keys,values in max_flow_sorted:
            i+=1
            if i!=p:
                if edge in critical_links[keys]:
                    curr_sum+=((len(G.edges()))*D(1+(len(G.edges())*D)))**(p-i-
1)#(1/flow_contributor[keys][edge])
            else:
                curr_sum+=1
        w[edge]=curr_sum
    non_vital_edge_flow_contributor=non_vital_flow_contributor(G.copy(), IEP_without_ab,max_flow_dict)
```

```
    import random as random
    #print(w)
    for keys,values in w.items():
        if w[keys]==0:
            if non_vital_edge_flow_contributor[keys]!=0:
                w[keys]=1/(non_vital_edge_flow_contributor[keys]*1000)
            else:
                w[keys]=0
    #Later on whoseover w(l)=0, choose the minimum no of
    hops #4.Eliminate <D edges and form reduced network
    new_G=nx.Graph()
    for node in G.nodes():
        new_G.add_node(node)
    for edge in G.edges():
        if G[edge[0]][edge[1]]['capacity']-D>=0:
            new_G.add_edge(edge[0],edge[1])
            new_G[edge[0]][edge[1]]['capacity']=w[edge]/G[edge[0]][edge[1]]['capacity']
    try:
        shortest_path=nx.shortest_path(new_G,ab[0],ab[1],weight='capacity')
    except:
        print("No Path Found from",ab[0],"to",ab[1])
        return G.copy(),0,0
    for i in range(len(shortest_path)-1):
        G[shortest_path[i]][shortest_path[i+1]]['capacity']-=D
    format_modified_shortest_path=[]
    for i in range(len(shortest_path)-1):
        l=[shortest_path[i],shortest_path[i+1]]
        format_modified_shortest_path.append(l)
#     for edge in G.edges():
#         print(G.get_edge_data(edge[0],edge[1])['capacity'])
    return G.copy(),aaaa,format_modified_shortest_path


#SWP Algorithm

def SWP(G,ab,D):
    new_G=G.copy()
    for edge in G.edges():
        if G[edge[0]][edge[1]]['capacity']<D:
            new_G.remove_edge(edge[0],edge[1])
    all_paths=list(nx.all_simple_paths(new_G,ab[0],ab[1]))
    if len(all_paths)==0:
        print("No Path found from",ab[0],"to",ab[1])
        return G.copy(),0,0
    all_paths_dict={}
    ctr=0
    for path in all_paths:
        all_paths_dict[ctr]=10000000000000000
        for i in range(len(path)-1):
                if new_G[path[i]][path[i+1]]['capacity']<all_paths_dict[ctr]:
                    all_paths_dict[ctr]=new_G[path[i]][path[i+1]]['capacity']
        ctr+=1
    max_value=0
    max_key=0
```

```
    new_path_list={}
    for keys,values in all_paths_dict.items():
        if all_paths_dict[keys]>max_value:
            max_value=all_paths_dict[keys]
    for keys,values in all_paths_dict.items():
        if all_paths_dict[keys]==max_value:
            new_path_list[keys]=all_paths[keys].copy()
    min_value=1000000000000
    min_key=0
    for keys,values in new_path_list.items():
        if len(new_path_list[keys])<min_value:
            min_value=len(new_path_list[keys])
            min_key=keys
    format_modified_shortest_path=[]
    for i in range(len(all_paths[min_key])-1):
        l=tuple((all_paths[min_key][i],all_paths[min_key][i+1]))
        G[l[0]][l[1]]['capacity']-=D
        format_modified_shortest_path.append(l)
    return G.copy(),0,format_modified_shortest_path


#WSP Algorithm

def WSP(G,ab,D):
    new_G=G.copy()
    for edge in G.edges():
        if G[edge[0]][edge[1]]['capacity']<D:
            new_G.remove_edge(edge[0],edge[1])
    all_paths=list(nx.all_simple_paths(new_G,ab[0],ab[1]))
    if len(all_paths)==0:
        print("No Path found from",ab[0],"to",ab[1])
        return G.copy(),0,0
    all_paths_dict={}
    ctr=0
    for path in all_paths:
        all_paths_dict[ctr]=len(path)
        ctr+=1
    min_value=1000000000000000
    min_key=0
    new_path_list={}
    for keys,values in all_paths_dict.items():
        if all_paths_dict[keys]<min_value:
            min_value=all_paths_dict[keys]
    for keys,values in all_paths_dict.items():
        if all_paths_dict[keys]==min_value:
            new_path_list[keys]=all_paths[keys].copy()
    max_value=0
    for keys,values in new_path_list.items():
        max_value=0
        for i in range(len(new_path_list[keys])-1):
                if new_G[new_path_list[keys][i]][new_path_list[keys][i+1]]['capacity']>max_value:
                    max_value=new_G[new_path_list[keys][i]][new_path_list[keys][i+1]]['capacity']
        new_path_list[keys]=max_value
    new_path_list_sorted= sorted(new_path_list.items(), key=lambda kv:(kv[1]),reverse=True)
```

```
    format_modified_shortest_path=[]
    for i in range(len(all_paths[new_path_list_sorted[0][0]])-1):
        l=tuple((all_paths[min_key][i],all_paths[min_key][i+1]))
        G[l[0]][l[1]]['capacity']-=D
        format_modified_shortest_path.append(l)
    return G.copy(),0,format_modified_shortest_path


#PLOTTING FUNCTION

def flow_decrease_on_links(flow_tracker,R,G):
    for edge in G.edges():
        try:
            flow_tracker[tuple(edge)].append(G[edge[0]][edge[1]]['capacity'])
        except:
            flow_tracker[tuple((edge[1],edge[0]))].append(G[edge[0]][edge[1]]['capacity'])

def link_usage(link_user,link_user_by_IEP,R,G,iep):
    for e in R:
        try:
            link_user[tuple(e)]+=1
        except:
            link_user[tuple((e[1],e[0]))]+=1
        try:
            link_user_by_IEP[tuple(iep)][tuple(e)]+=1
        except:
            link_user_by_IEP[tuple(iep)][tuple((e[1],e[0]))]+=1


def link_user_plotter(link_user):
    link_string_list=[]
    for l in link_user.keys():
        link_string_list.append(str(tuple(l)))
    plt.figure(figsize=(20,5))
    plt.bar(link_string_list,link_user.values())
    plt.xlabel("Links")
    plt.ylabel("Usage Frequency")
def link_user_by_IEP_plotter(link_user_by_IEP):
    fig,axis=plt.subplots(5,1,figsize=(20,20))
    link_string_list=[]
    temp=list(link_user_by_IEP.keys())[0]
    for l in link_user_by_IEP[temp]:
        link_string_list.append(str(tuple(l)))
    i=0
    for keys in link_user_by_IEP.keys():
        #print(len(link_string_list),len(link_user_by_IEP[keys]))
        axis[i].plot(link_string_list,link_user_by_IEP[keys].values())
        axis[i].legend([str(keys)])
        axis[i].set(ylabel="Frequency",xlabel="Links")
        i+=1
    fig.savefig('link_usage_by_IEP.jpg', bbox_inches='tight')
#     plt.figure(figsize=(25,8))
#     plt.bar(link_string_list,link_user.values())
#     plt.xlabel("Links")
```

```
#     plt.ylabel("Usage Frequency")

import matplotlib.pyplot as plt

def flow_tracker_plotter(dictionary):
    plt.figure(figsize=(50,50))
    figure, axis = plt.subplots(4, 7,figsize=(25,10),sharex=True,sharey=True)
    figure.text(0.5, 0.04, 'Instance', ha='center')
    figure.text(0.04, 0.5, 'Flow', va='center', rotation='vertical')
    r,c=0,0
    for keys,values in dictionary.items():
        axis[r,c].plot(range(len(dictionary[keys])),dictionary[keys])
        axis[r,c].legend([str(keys)],fontsize=12)

        c+=1
        if c%7==0:
            r+=1
            c=0
def rejection_plotter(rejection,IEP,PAIR):
    demand_by_IEP={}
    for iep in IEP:
        demand_by_IEP[tuple(iep)]=0
    for p in PAIR:
        demand_by_IEP[tuple(IEP[p])]+=1
    accepted={}
    for iep in IEP:
        accepted[tuple(iep)]=demand_by_IEP[tuple(iep)]-
    rejection[tuple(iep)] IEP_string_list=[]
    plt.figure(figsize=(50,50))
    figure, axis = plt.subplots(1, 3,figsize=(15,5))
    #figure.text(0.5, 0.04, 'Instance', ha='center')
    #figure.text(0.04, 0.5, 'Flow', va='center',
    rotation='vertical') for iep in IEP:
        IEP_string_list.append(str(tuple(iep)))

    axis[0].bar(IEP_string_list,demand_by_IEP.values())
    axis[1].bar(IEP_string_list,accepted.values())
    axis[2].bar(IEP_string_list,rejection.values())
    axis[0].set(xlabel="Ingress-Egress Pair",ylabel=" Total Demands")
    axis[1].set(xlabel="Ingress-Egress Pair",ylabel="Accepted Demands")
    axis[2].set(xlabel="Ingress-Egress Pair",ylabel="Rejected Demands")

def path_length_plotter(path_length,IEP):
    iep_string_list=[]
    for iep in IEP:
        iep_string_list.append(str(tuple(iep)))
    plt.bar(iep_string_list,path_length.values())
    plt.xlabel("Ingress Egree Pairs")
    plt.ylabel("Average Path Length")


def max_flow_decrease(G,max_flow_tracker, IEP):
    for iep in IEP:
        max_flow_tracker[tuple(iep)].append(nx.maximum_flow(G,iep[0],iep[1])[0])
```

```
def max_flow_decrease_plotter(max_flow_tracker,IEP):

    #fig,axis=plt.subplots(1,5,figsize=(20,5))
    i=0
    plt.figure(figsize=(10,5))
    IEP_string_list=[]
    for iep in IEP:
        IEP_string_list.append(str(tuple(iep)))
    for iep in IEP:
        plt.plot(list(range(4001)),max_flow_tracker[tuple(iep)])
        plt.legend(IEP_string_list)
        plt.xlabel("Demand Instance")
        plt.ylabel("MaxFlow Values")
#         axis[i].plot(list(range(4001)),max_flow_tracker[tuple(iep)])
#         axis[i].set(xlabel="Demand Instance",ylabel="MaxFlow Value")
#         axis[i].legend(str(tuple(iep)))

        i+=1
def router_traversal(node_frequency,R):
    distinct=[]
    for r in R:
        if r[0] not in distinct:
            distinct.append(r[0])
        if r[1] not in distinct:
            distinct.append(r[1])
    for r in distinct:
        node_frequency[r]+=1
def router_traversal_plotter(node_frequency):
    plt.figure(figsize=(15,5))
    plt.bar(node_frequency.keys(),node_frequency.values())
    a=[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
    plt.xticks(a,a)
    plt.xlabel("Router Index")
    plt.ylabel("Usage or Traversal Frequency")


#Plotting Function Caller
#Call the Below after running the appropriate algorithm driver code for the plots. Uncomment the
appropriate line for the required plot.
# flow_tracker_plotter(flow_tracker)
# rejection_plotter(rejection,IEP,PAIR)
#link_user_plotter(link_user)
#link_user_by_IEP_plotter(link_user_by_IEP)
# path_length_plotter(path_length,IEP)
# max_flow_decrease_plotter(max_flow_tracker,IEP)
#router_traversal_plotter(node_traversal)



#MIRA DRIVER CODE


flow_tracker={}
max_flow_tracker={}
```

```
node_traversal={}
for node in G.nodes():
    node_traversal[node]=0
for edge in G.edges():
    flow_tracker[edge]=[G[edge[0]][edge[1]]['capacity'],]
for iep in IEP:
    max_flow_tracker[tuple(iep)]=[]
    max_flow_tracker[tuple(iep)].append(nx.maximum_flow(G,iep[0],iep[1])[0])
rejection={}
for iep in IEP:
    rejection[tuple(iep)]=0
link_user={}
 for edge in G.edges():
    link_user[edge]=0
 link_user_by_IEP={}
for iep in IEP:
    link_user_by_IEP[tuple(iep)]=link_user.copy()
path_length={}
for iep in IEP:
    path_length[tuple(iep)]=[]
G_copy=G.copy()
G_list_MIRA=[]
accepted=0
not_accepted=0
for demand in range(4000):
    G_copy,aaaa,R=MIRA(G_copy.copy(),IEP[PAIR[demand]],IEP,DEMAND[demand])
    flow_decrease_on_links(flow_tracker,R,G_copy)
    max_flow_decrease(G_copy,max_flow_tracker,IEP)
    if R==0:
        not_accepted+=1
        rejection[tuple(IEP[PAIR[demand]])]+=1
    else:
        G_list_MIRA.append((G_copy.copy(),R,IEP[PAIR[demand]],DEMAND[demand]))
        link_usage(link_user,link_user_by_IEP,R,G,IEP[PAIR[demand]])
        accepted+=1
        path_length[tuple(IEP[PAIR[demand]])].append(len(R))
        router_traversal(node_traversal,R)
for keys in path_length.keys():
    path_length[keys]=sum(path_length[keys])/len(path_length[keys])

print("MIRA")
print("ACCEPTED!!!!",accepted)
print("NOT ACCEPTED!!!",not_accepted)

comparison_db["MIRA"]={}
comparison_db["MIRA"]["FLOWLINKDECREASE"]=flow_tracker
comparison_db["MIRA"]["REJECTION"]=rejection
comparison_db["MIRA"]["LINKUSAGE"]=link_user
comparison_db["MIRA"]["LINKUSAGEBYIEP"]=link_user_by_IEP
comparison_db["MIRA"]["PATHLENGTH"]=path_length
comparison_db["MIRA"]["MAXFLOWDECREASE"]=max_flow_tracker
```

```
#MHA DRIVER CODE

G_copy=G.copy()
G_list_MHA=[]
flow_tracker={}
max_flow_tracker={}
node_traversal={}
for node in G.nodes():
    node_traversal[node]=0
for edge in G.edges():a
    flow_tracker[edge]=[G[edge[0]][edge[1]]['capacity'],]
for iep in IEP:
    max_flow_tracker[tuple(iep)]=[]
    max_flow_tracker[tuple(iep)].append(nx.maximum_flow(G,iep[0],iep[1])[0])
rejection={}
for iep in IEP:
    rejection[tuple(iep)]=0
link_user={}
 for edge in G.edges():
    link_user[edge]=0
 link_user_by_IEP={}
for iep in IEP:
    link_user_by_IEP[tuple(iep)]=link_user.copy()
path_length={}
for iep in IEP:
    path_length[tuple(iep)]=[]
accepted=0
not_accepted=0
for demand in range(4000):
    G_copy,R=MHA(G_copy.copy(),IEP[PAIR[demand]],DEMAND[demand])
    flow_decrease_on_links(flow_tracker,R,G_copy)
    max_flow_decrease(G_copy,max_flow_tracker,IEP)
    if R==0:

        not_accepted+=1
        rejection[tuple(IEP[PAIR[demand]])]+=1
    else:
        G_list_MHA.append((G_copy.copy(),R,IEP[PAIR[demand]],DEMAND[demand]))
        link_usage(link_user,link_user_by_IEP,R,G,IEP[PAIR[demand]])
        accepted+=1
        path_length[tuple(IEP[PAIR[demand]])].append(len(R))
        router_traversal(node_traversal,R)
for keys in path_length.keys():
    path_length[keys]=sum(path_length[keys])/len(path_length[keys])

print("MHA")
print("ACCEPTED!!!!",accepted)
print("NOT ACCEPTED!!!",not_accepted)


comparison_db["MHA"]={}
comparison_db["MHA"]["FLOWLINKDECREASE"]=flow_tracker
comparison_db["MHA"]["REJECTION"]=rejection
```

```
comparison_db["MHA"]["LINKUSAGE"]=link_user
comparison_db["MHA"]["LINKUSAGEBYIEP"]=link_user_by_IEP
comparison_db["MHA"]["PATHLENGTH"]=path_length
comparison_db["MHA"]["MAXFLOWDECREASE"]=max_flow_tracker


#IMIRA Driver Code
flow_tracker={}
max_flow_tracker={}
node_traversal={}
for node in G.nodes():
    node_traversal[node]=0
for edge in G.edges():
    flow_tracker[edge]=[G[edge[0]][edge[1]]['capacity'],]
for iep in IEP:
    max_flow_tracker[tuple(iep)]=[]
    max_flow_tracker[tuple(iep)].append(nx.maximum_flow(G,iep[0],iep[1])[0])
rejection={}
for iep in IEP:
    rejection[tuple(iep)]=0
link_user={}
 for edge in G.edges():
    link_user[edge]=0
 link_user_by_IEP={}
for iep in IEP:
    link_user_by_IEP[tuple(iep)]=link_user.copy()
path_length={}
for iep in IEP:
    path_length[tuple(iep)]=[]
G_copy=G.copy()
G_list_MIRA=[]
accepted=0
not_accepted=0
for demand in range(4000):
    G_copy,aaaa,R=MIRA_1(G_copy.copy(),IEP[PAIR[demand]],IEP,DEMAND[demand])
    flow_decrease_on_links(flow_tracker,R,G_copy)
    max_flow_decrease(G_copy,max_flow_tracker,IEP)
    if R==0:
        not_accepted+=1
        rejection[tuple(IEP[PAIR[demand]])]+=1
    else:
        G_list_MIRA.append((G_copy.copy(),R,IEP[PAIR[demand]],DEMAND[demand]))
        accepted+=1
        link_usage(link_user,link_user_by_IEP,R,G,IEP[PAIR[demand]])
        path_length[tuple(IEP[PAIR[demand]])].append(len(R))
        router_traversal(node_traversal,R)
for keys in path_length.keys():
    path_length[keys]=sum(path_length[keys])/len(path_length[keys])

print("I-MIRA")
print("ACCEPTED!!!!",accepted)
print("NOT ACCEPTED!!!",not_accepted)
```

```
comparison_db["IMIRA"]={}
comparison_db["IMIRA"]["FLOWLINKDECREASE"]=flow_tracker
comparison_db["IMIRA"]["REJECTION"]=rejection
comparison_db["IMIRA"]["LINKUSAGE"]=link_user
comparison_db["IMIRA"]["LINKUSAGEBYIEP"]=link_user_by_IEP
comparison_db["IMIRA"]["PATHLENGTH"]=path_length
comparison_db["IMIRA"]["MAXFLOWDECREASE"]=max_flow_tracker



#MIH DRIVER CODE

flow_tracker={}
max_flow_tracker={}
node_traversal={}
for node in G.nodes():
    node_traversal[node]=0
for edge in G.edges():
    flow_tracker[edge]=[G[edge[0]][edge[1]]['capacity'],]
for iep in IEP:
    max_flow_tracker[tuple(iep)]=[]
    max_flow_tracker[tuple(iep)].append(nx.maximum_flow(G,iep[0],iep[1])[0])
rejection={}
for iep in IEP:
    rejection[tuple(iep)]=0
link_user={}
 for edge in G.edges():
    link_user[edge]=0
 link_user_by_IEP={}
for iep in IEP:
    link_user_by_IEP[tuple(iep)]=link_user.copy()
path_length={}
for iep in IEP:
    path_length[tuple(iep)]=[]
fij={}
entire_cap={}
for edge in G.edges():
    fij[edge]=0
    entire_cap[edge]=G[edge[0]][edge[1]]['capacity']
G_copy=G.copy()
G_list_MIH_8_4=[]
T1=8
T2=4
accepted=0
not_accepted=0
for demand in range(4000):
    G_copy,aaaa,R=MIH(G_copy.copy(),IEP[PAIR[demand]],IEP,DEMAND[demand],fij,entire_cap,T1,T2)
    flow_decrease_on_links(flow_tracker,R,G_copy)
    max_flow_decrease(G_copy,max_flow_tracker,IEP)
    if R==0:
        not_accepted+=1
        rejection[tuple(IEP[PAIR[demand]])]+=1
    else:
        G_list_MIH_8_4.append((G_copy.copy(),R,IEP[PAIR[demand]],DEMAND[demand]))
```

```
        accepted+=1
        link_usage(link_user,link_user_by_IEP,R,G,IEP[PAIR[demand]])
        path_length[tuple(IEP[PAIR[demand]])].append(len(R))
        router_traversal(node_traversal,R)
for keys in path_length.keys():
    path_length[keys]=sum(path_length[keys])/len(path_length[keys])
print("MIH")
print("ACCEPTED!!!!",accepted)
print("NOT ACCEPTED!!!",not_accepted)


comparison_db["MIH"]={}
comparison_db["MIH"]["FLOWLINKDECREASE"]=flow_tracker
comparison_db["MIH"]["REJECTION"]=rejection
comparison_db["MIH"]["LINKUSAGE"]=link_user
comparison_db["MIH"]["LINKUSAGEBYIEP"]=link_user_by_IEP
comparison_db["MIH"]["PATHLENGTH"]=path_length
comparison_db["MIH"]["MAXFLOWDECREASE"]=max_flow_tracker



#LIOA Driver Code
#def LIOA(G, ab, IEP,D,capacity_dict,demand_dict,alpha):

flow_tracker={}
max_flow_tracker={}
node_traversal={}
for node in G.nodes():
    node_traversal[node]=0
for edge in G.edges():
    flow_tracker[edge]=[G[edge[0]][edge[1]]['capacity'],]
for iep in IEP:
    max_flow_tracker[tuple(iep)]=[]
    max_flow_tracker[tuple(iep)].append(nx.maximum_flow(G,iep[0],iep[1])[0])
rejection={}
for iep in IEP:
    rejection[tuple(iep)]=0
link_user={}
 for edge in G.edges():
    link_user[edge]=0
 link_user_by_IEP={}
for iep in IEP:
    link_user_by_IEP[tuple(iep)]=link_user.copy()
path_length={}
for iep in IEP:
    path_length[tuple(iep)]=[]
G_copy=G.copy()
G_list_LIOA=[]
accepted=0
not_accepted=0
demand_dict={}
for edge in G.edges():
    demand_dict[edge]=0
capacity_dict={}
```

```
for edge in G.edges():
    capacity_dict[edge]=G[edge[0]][edge[1]]['capacity']
alpha=0.5
for demand in range(4000):

G_copy,aaaa,R=LIOA(G_copy.copy(),IEP[PAIR[demand]],IEP,DEMAND[demand],capacity_dict,demand_d
ict,alpha)
    flow_decrease_on_links(flow_tracker,R,G_copy)
    max_flow_decrease(G_copy,max_flow_tracker,IEP)
    if R==0:
        not_accepted+=1
        rejection[tuple(IEP[PAIR[demand]])]+=1
    else:
        G_list_LIOA.append((G_copy.copy(),R,IEP[PAIR[demand]],DEMAND[demand]))
        accepted+=1
        link_usage(link_user,link_user_by_IEP,R,G,IEP[PAIR[demand]])
        path_length[tuple(IEP[PAIR[demand]])].append(len(R))
        router_traversal(node_traversal,R)

for keys in path_length.keys():
    path_length[keys]=sum(path_length[keys])/len(path_length[keys])
print("LIOA",alpha)
print("ACCEPTED!!!!",accepted)
print("NOT ACCEPTED!!!",not_accepted)

comparison_db["LIOA"]={}
comparison_db["LIOA"]["FLOWLINKDECREASE"]=flow_tracker
comparison_db["LIOA"]["REJECTION"]=rejection
comparison_db["LIOA"]["LINKUSAGE"]=link_user
comparison_db["LIOA"]["LINKUSAGEBYIEP"]=link_user_by_IEP
comparison_db["LIOA"]["PATHLENGTH"]=path_length
comparison_db["LIOA"]["MAXFLOWDECREASE"]=max_flow_tracker



#ILIOA driver Code

flow_tracker={}
max_flow_tracker={}
node_traversal={}
for node in G.nodes():
    node_traversal[node]=0
for edge in G.edges():
    flow_tracker[edge]=[G[edge[0]][edge[1]]['capacity'],]
for iep in IEP:
    max_flow_tracker[tuple(iep)]=[]
    max_flow_tracker[tuple(iep)].append(nx.maximum_flow(G,iep[0],iep[1])[0])
rejection={}
for iep in IEP:
    rejection[tuple(iep)]=0
link_user={}
 for edge in G.edges():
    link_user[edge]=0
 link_user_by_IEP={}
```

```
for iep in IEP:
    link_user_by_IEP[tuple(iep)]=link_user.copy()
path_length={}
for iep in IEP:
    path_length[tuple(iep)]=[]
alpha=0.5
beta=0.5
G_copy=G.copy()
G_list_ILIOA=[]
accepted=0
not_accepted=0
demand_dict={}
for edge in G.edges():
    demand_dict[edge]=0
capacity_dict={}
for edge in G.edges():
    capacity_dict[edge]=G[edge[0]][edge[1]]['capacity']
for demand in range(4000):

G_copy,aaaa,R=ILIOA(G_copy.copy(),IEP[PAIR[demand]],IEP,DEMAND[demand],capacity_dict,demand_
dict,alpha,beta)
    flow_decrease_on_links(flow_tracker,R,G_copy)
    max_flow_decrease(G_copy,max_flow_tracker,IEP)

    if R==0:
        not_accepted+=1
        rejection[tuple(IEP[PAIR[demand]])]+=1
    else:
        G_list_ILIOA.append((G_copy.copy(),R,IEP[PAIR[demand]],DEMAND[demand]))
        accepted+=1
        link_usage(link_user,link_user_by_IEP,R,G,IEP[PAIR[demand]])
        path_length[tuple(IEP[PAIR[demand]])].append(len(R))
        router_traversal(node_traversal,R)

for keys in path_length.keys():
    path_length[keys]=sum(path_length[keys])/len(path_length[keys])

print("ILIOA",alpha,beta)
print("ACCEPTED!!!!",accepted)
print("NOT ACCEPTED!!!",not_accepted)



comparison_db["ILIOA"]={}
comparison_db["ILIOA"]["FLOWLINKDECREASE"]=flow_tracker
comparison_db["ILIOA"]["REJECTION"]=rejection
comparison_db["ILIOA"]["LINKUSAGE"]=link_user
comparison_db["ILIOA"]["LINKUSAGEBYIEP"]=link_user_by_IEP
comparison_db["ILIOA"]["PATHLENGTH"]=path_length
comparison_db["ILIOA"]["MAXFLOWDECREASE"]=max_flow_tracker


#BCRA Driver Code
```

```
flow_tracker={}
max_flow_tracker={}
node_traversal={}
for node in G.nodes():
    node_traversal[node]=0
for edge in G.edges():
    flow_tracker[edge]=[G[edge[0]][edge[1]]['capacity'],]
for iep in IEP:
    max_flow_tracker[tuple(iep)]=[]
    max_flow_tracker[tuple(iep)].append(nx.maximum_flow(G,iep[0],iep[1])[0])
rejection={}
for iep in IEP:
    rejection[tuple(iep)]=0
link_user={}
 for edge in G.edges():
    link_user[edge]=0
 link_user_by_IEP={}
for iep in IEP:
    link_user_by_IEP[tuple(iep)]=link_user.copy()
path_length={}
for iep in IEP:
    path_length[tuple(iep)]=[]
G_copy=G.copy()
G_list_LIOA=[]
accepted=0
not_accepted=0
demand_dict={}
for edge in G.edges():
    demand_dict[edge]=0
capacity_dict={}
for edge in G.edges():
    capacity_dict[edge]=G[edge[0]][edge[1]]['capacity']
G_copy=G.copy()
G_list_BCRA=[]
accepted=0
not_accepted=0
capacity_dict={}
for edge in G.edges():
    capacity_dict[edge]=G[edge[0]][edge[1]]['capacity']
for demand in range(10000):
    G_copy,aaaa,R=BCRA(G_copy.copy(),IEP[PAIR[demand]],IEP,DEMAND[demand],capacity_dict)
    flow_decrease_on_links(flow_tracker,R,G_copy)
    max_flow_decrease(G_copy,max_flow_tracker,IEP)
    if R==0:
        not_accepted+=1
        rejection[tuple(IEP[PAIR[demand]])]+=1
    else:
        G_list_BCRA.append((G_copy.copy(),R,IEP[PAIR[demand]],DEMAND[demand]))
        accepted+=1
        link_usage(link_user,link_user_by_IEP,R,G,IEP[PAIR[demand]])
        path_length[tuple(IEP[PAIR[demand]])].append(len(R))
        router_traversal(node_traversal,R)
for keys in path_length.keys():
    path_length[keys]=sum(path_length[keys])/len(path_length[keys])
```

```
print("BCRA")
print("ACCEPTED!!!!",accepted)
print("NOT ACCEPTED!!!",not_accepted)



comparison_db["BCRA"]={}
comparison_db["BCRA"]["FLOWLINKDECREASE"]=flow_tracker
comparison_db["BCRA"]["REJECTION"]=rejection
comparison_db["BCRA"]["LINKUSAGE"]=link_user
comparison_db["BCRA"]["LINKUSAGEBYIEP"]=link_user_by_IEP
comparison_db["BCRA"]["PATHLENGTH"]=path_length
comparison_db["BCRA"]["MAXFLOWDECREASE"]=max_flow_tracker


#RNLC Driver Code

#for C in [0.0001,0.001,0.01,0.1,1,2,3,4,5,6,7,8,9,10,100,1000,10000]:
#def RNLC(G, ab, IEP,D,C):


flow_tracker={}
max_flow_tracker={}
node_traversal={}
for node in G.nodes():
    node_traversal[node]=0
for edge in G.edges():
    flow_tracker[edge]=[G[edge[0]][edge[1]]['capacity'],]
for iep in IEP:
    max_flow_tracker[tuple(iep)]=[]
    max_flow_tracker[tuple(iep)].append(nx.maximum_flow(G,iep[0],iep[1])[0])
rejection={}
for iep in IEP:
    rejection[tuple(iep)]=0
link_user={}
 for edge in G.edges():
    link_user[edge]=0
 link_user_by_IEP={}
for iep in IEP:
    link_user_by_IEP[tuple(iep)]=link_user.copy()
path_length={}
for iep in IEP:
    path_length[tuple(iep)]=[]
G_copy=G.copy()
G_list_RNLC=[]
accepted=0
C=1
not_accepted=0
for demand in range(10000):
    G_copy,aaaa,R=RNLC(G_copy.copy(),IEP[PAIR[demand]],IEP,DEMAND[demand],C)
    flow_decrease_on_links(flow_tracker,R,G_copy)
    max_flow_decrease(G_copy,max_flow_tracker,IEP)
```

```
    if R==0:
        not_accepted+=1
        rejection[tuple(IEP[PAIR[demand]])]+=1
    else:
        G_list_RNLC.append((G_copy.copy(),R,IEP[PAIR[demand]],DEMAND[demand]))
        accepted+=1
        link_usage(link_user,link_user_by_IEP,R,G,IEP[PAIR[demand]])
        path_length[tuple(IEP[PAIR[demand]])].append(len(R))
        router_traversal(node_traversal,R)
for keys in path_length.keys():
    path_length[keys]=sum(path_length[keys])/len(path_length[keys])

print("RNLC",C)
print("ACCEPTED!!!!",accepted)
print("NOT ACCEPTED!!!",not_accepted)


comparison_db["RNLC"]={}
comparison_db["RNLC"]["FLOWLINKDECREASE"]=flow_tracker
comparison_db["RNLC"]["REJECTION"]=rejection
comparison_db["RNLC"]["LINKUSAGE"]=link_user
comparison_db["RNLC"]["LINKUSAGEBYIEP"]=link_user_by_IEP
comparison_db["RNLC"]["PATHLENGTH"]=path_length
comparison_db["RNLC"]["MAXFLOWDECREASE"]=max_flow_tracker

#Max RC Min F Driver Code

#def RCF(G, ab, IEP,D,capacity_dict,demand_dict):



#for k in [0.00001,0.0001,0.001,0.01,0.1,1,2,3,4,5,6,7,8,9,10,100,1000,10000,100000]:
flow_tracker={}
max_flow_tracker={}
node_traversal={}
for node in G.nodes():
    node_traversal[node]=0
for edge in G.edges():
    flow_tracker[edge]=[G[edge[0]][edge[1]]['capacity'],]
for iep in IEP:
    max_flow_tracker[tuple(iep)]=[]
    max_flow_tracker[tuple(iep)].append(nx.maximum_flow(G,iep[0],iep[1])[0])
rejection={}
for iep in IEP:
    rejection[tuple(iep)]=0
link_user={}
 for edge in G.edges():
    link_user[edge]=0
 link_user_by_IEP={}
for iep in IEP:
    link_user_by_IEP[tuple(iep)]=link_user.copy()
path_length={}
for iep in IEP:
    path_length[tuple(iep)]=[]
```

```
G_copy=G.copy()
G_list_RCF=[]
accepted=0
not_accepted=0
demand_dict={}
for edge in G.edges():
   demand_dict[edge]=0
capacity_dict={}
for edge in G.edges():
   capacity_dict[edge]=G[edge[0]][edge[1]]['capacity']
for demand in range(10000):
#    try:

G_copy,aaaa,R=RCF(G_copy.copy(),IEP[PAIR[demand]],IEP,DEMAND[demand],capacity_dict,demand_di ct)

   flow_decrease_on_links(flow_tracker,R,G_copy)
   max_flow_decrease(G_copy,max_flow_tracker,IEP)

   if R==0:
      not_accepted+=1
      rejection[tuple(IEP[PAIR[demand]])]+=1
   else:
      G_list_RCF.append((G_copy.copy(),R,IEP[PAIR[demand]],DEMAND[demand]))
      accepted+=1
      link_usage(link_user,link_user_by_IEP,R,G,IEP[PAIR[demand]])
      path_length[tuple(IEP[PAIR[demand]])].append(len(R))
      router_traversal(node_traversal,R)

for keys in path_length.keys():
   path_length[keys]=sum(path_length[keys])/len(path_length[keys])


print("Max Rc Min F")
print("ACCEPTED!!!!",accepted)
print("NOT ACCEPTED!!!",not_accepted)



comparison_db["RCF"]={}
comparison_db["RCF"]["FLOWLINKDECREASE"]=flow_tracker
comparison_db["RCF"]["REJECTION"]=rejection
comparison_db["RCF"]["LINKUSAGE"]=link_user
comparison_db["RCF"]["LINKUSAGEBYIEP"]=link_user_by_IEP
comparison_db["RCF"]["PATHLENGTH"]=path_length
comparison_db["RCF"]["MAXFLOWDECREASE"]=max_flow_tracker


#NHMI Driver Code

flow_tracker={}
max_flow_tracker={}
node_traversal={}
for node in G.nodes():
   node_traversal[node]=0
```

```
for edge in G.edges():
    flow_tracker[edge]=[G[edge[0]][edge[1]]['capacity'],]
for iep in IEP:
    max_flow_tracker[tuple(iep)]=[]
    max_flow_tracker[tuple(iep)].append(nx.maximum_flow(G,iep[0],iep[1])[0])
rejection={}
for iep in IEP:
    rejection[tuple(iep)]=0
link_user={}
 for edge in G.edges():
    link_user[edge]=0
 link_user_by_IEP={}
for iep in IEP:
    link_user_by_IEP[tuple(iep)]=link_user.copy()
path_length={}
for iep in IEP:
    path_length[tuple(iep)]=[]
T1=10
T2=100
import numpy as np
G_copy=G.copy()
G_list_NMIH=[]
fij={}
entire_cap={}
for edge in G.edges():
    fij[edge]=0
    entire_cap[edge]=G[edge[0]][edge[1]]['capacity']
accepted=0
not_accepted=0
for demand in range(4000):

        G_copy,aaaa,R=NMIH(G_copy.copy(),IEP[PAIR[demand]],IEP,DEMAND[demand],entire_cap,T1,T2)
        flow_decrease_on_links(flow_tracker,R,G_copy)
        max_flow_decrease(G_copy,max_flow_tracker,IEP)
        if R==0:
            not_accepted+=1
            rejection[tuple(IEP[PAIR[demand]])]+=1
        else:
            G_list_NMIH.append((G_copy.copy(),R,IEP[PAIR[demand]],DEMAND[demand]))
            accepted+=1
            link_usage(link_user,link_user_by_IEP,R,G,IEP[PAIR[demand]])
            path_length[tuple(IEP[PAIR[demand]])].append(len(R))
            router_traversal(node_traversal,R)


for keys in path_length.keys():
    path_length[keys]=sum(path_length[keys])/len(path_length[keys])




print("NMHI",T1,T2)
print("ACCEPTED!!!!",accepted)
print("NOT ACCEPTED!!!",not_accepted)
```

```
comparison_db["NHMI"]={}
comparison_db["NHMI"]["FLOWLINKDECREASE"]=flow_tracker
comparison_db["NHMI"]["REJECTION"]=rejection
comparison_db["NHMI"]["LINKUSAGE"]=link_user
comparison_db["NHMI"]["LINKUSAGEBYIEP"]=link_user_by_IEP
comparison_db["NHMI"]["PATHLENGTH"]=path_length
comparison_db["NHMI"]["MAXFLOWDECREASE"]=max_flow_tracker


#LEX MAX DRIVER CODE


flow_tracker={}
max_flow_tracker={}
node_traversal={}
for node in G.nodes():
    node_traversal[node]=0
for edge in G.edges():
    flow_tracker[edge]=[G[edge[0]][edge[1]]['capacity'],]
for iep in IEP:
    max_flow_tracker[tuple(iep)]=[]
    max_flow_tracker[tuple(iep)].append(nx.maximum_flow(G,iep[0],iep[1])[0])
rejection={}
for iep in IEP:
    rejection[tuple(iep)]=0
link_user={}
 for edge in G.edges():
    link_user[edge]=0
 link_user_by_IEP={}
for iep in IEP:
    link_user_by_IEP[tuple(iep)]=link_user.copy()
path_length={}
for iep in IEP:
    path_length[tuple(iep)]=[]

G_copy=G.copy()
G_list_LEXMAX=[]
accepted=0
not_accepted=0
delta=1
for demand in range(10000):
    G_copy,aaaa,R=LEXMAX(G_copy.copy(),IEP[PAIR[demand]],IEP,DEMAND[demand],delta)
    flow_decrease_on_links(flow_tracker,R,G_copy)
    max_flow_decrease(G_copy,max_flow_tracker,IEP)
    if R==0:
        not_accepted+=1
        rejection[tuple(IEP[PAIR[demand]])]+=1
    else:
        G_list_LEXMAX.append((G_copy.copy(),R,IEP[PAIR[demand]],DEMAND[demand]))
        accepted+=1
        link_usage(link_user,link_user_by_IEP,R,G,IEP[PAIR[demand]])
        path_length[tuple(IEP[PAIR[demand]])].append(len(R))
```

```
        router_traversal(node_traversal,R)



for keys in path_length.keys():
    path_length[keys]=sum(path_length[keys])/len(path_length[keys])

print("LEXMAX")
print("ACCEPTED!!!!",accepted)
print("NOT ACCEPTED!!!",not_accepted)

comparison_db["LEXMAX"]={}
comparison_db["LEXMAX"]["FLOWLINKDECREASE"]=flow_tracker
comparison_db["LEXMAX"]["REJECTION"]=rejection
comparison_db["LEXMAX"]["LINKUSAGE"]=link_user
comparison_db["LEXMAX"]["LINKUSAGEBYIEP"]=link_user_by_IEP
comparison_db["LEXMAX"]["PATHLENGTH"]=path_length
comparison_db["LEXMAX"]["MAXFLOWDECREASE"]=max_flow_tracker




#WSP Driver Code

flow_tracker={}
max_flow_tracker={}
node_traversal={}
for node in G.nodes():
    node_traversal[node]=0
for edge in G.edges():
    flow_tracker[edge]=[G[edge[0]][edge[1]]['capacity'],]
for iep in IEP:
    max_flow_tracker[tuple(iep)]=[]
    max_flow_tracker[tuple(iep)].append(nx.maximum_flow(G,iep[0],iep[1])[0])
rejection={}
for iep in IEP:
    rejection[tuple(iep)]=0
link_user={}
 for edge in G.edges():
    link_user[edge]=0
 link_user_by_IEP={}
for iep in IEP:
    link_user_by_IEP[tuple(iep)]=link_user.copy()
path_length={}
for iep in IEP:
    path_length[tuple(iep)]=[]




G_copy=G.copy()
G_list_WSP=[]
critical_progress_WSP=[]
```

```
max_flow_progress_WSP=[]
accepted=0
not_accepted=0
demand_dict={}

for demand in range(4000):
    G_copy,aaaa,R=WSP(G_copy.copy(),IEP[PAIR[demand]],DEMAND[demand])
    flow_decrease_on_links(flow_tracker,R,G_copy)
    max_flow_decrease(G_copy,max_flow_tracker,IEP)
    if R==0:
        not_accepted+=1
        rejection[tuple(IEP[PAIR[demand]])]+=1
    else:
        G_list_WSP.append((G_copy.copy(),R,IEP[PAIR[demand]],DEMAND[demand]))
        accepted+=1
        link_usage(link_user,link_user_by_IEP,R,G,IEP[PAIR[demand]])
        path_length[tuple(IEP[PAIR[demand]])].append(len(R))
        router_traversal(node_traversal,R)

for keys in path_length.keys():
    path_length[keys]=sum(path_length[keys])/len(path_length[keys])

print("WSP")
print("ACCEPTED!!!!",accepted)
print("NOT ACCEPTED!!!",not_accepted)


comparison_db["WSP"]={}
comparison_db["WSP"]["FLOWLINKDECREASE"]=flow_tracker
comparison_db["WSP"]["REJECTION"]=rejection
comparison_db["WSP"]["LINKUSAGE"]=link_user
comparison_db["WSP"]["LINKUSAGEBYIEP"]=link_user_by_IEP
comparison_db["WSP"]["PATHLENGTH"]=path_length
comparison_db["WSP"]["MAXFLOWDECREASE"]=max_flow_tracker


#SWP Driver COde


flow_tracker={}
max_flow_tracker={}
node_traversal={}
for node in G.nodes():
    node_traversal[node]=0
for edge in G.edges():
    flow_tracker[edge]=[G[edge[0]][edge[1]]['capacity'],]
for iep in IEP:
    max_flow_tracker[tuple(iep)]=[]
    max_flow_tracker[tuple(iep)].append(nx.maximum_flow(G,iep[0],iep[1])[0])
rejection={}
for iep in IEP:
    rejection[tuple(iep)]=0
link_user={}
for edge in G.edges():
```

```
    link_user[edge]=0
link_user_by_IEP={}
for iep in IEP:
    link_user_by_IEP[tuple(iep)]=link_user.copy()
path_length={}
for iep in IEP:
    path_length[tuple(iep)]=[]

G_copy=G.copy()
G_list_SWP=[]
accepted=0
not_accepted=0
demand_dict={}

for demand in range(4000):
    G_copy,aaaa,R=SWP(G_copy.copy(),IEP[PAIR[demand]],DEMAND[demand])
    flow_decrease_on_links(flow_tracker,R,G_copy)
    max_flow_decrease(G_copy,max_flow_tracker,IEP)
    if R==0:
        not_accepted+=1
        rejection[tuple(IEP[PAIR[demand]])]+=1


    else:
        G_list_SWP.append((G_copy.copy(),R,IEP[PAIR[demand]],DEMAND[demand]))
        accepted+=1
        link_usage(link_user,link_user_by_IEP,R,G,IEP[PAIR[demand]])
        path_length[tuple(IEP[PAIR[demand]])].append(len(R))
        router_traversal(node_traversal,R)

for keys in path_length.keys():
    path_length[keys]=sum(path_length[keys])/len(path_length[keys])




print("SWP")
print("ACCEPTED!!!!",accepted)
print("NOT ACCEPTED!!!",not_accepted)



comparison_db["SWP"]={}
comparison_db["SWP"]["FLOWLINKDECREASE"]=flow_tracker
comparison_db["SWP"]["REJECTION"]=rejection
comparison_db["SWP"]["LINKUSAGE"]=link_user
comparison_db["SWP"]["LINKUSAGEBYIEP"]=link_user_by_IEP
comparison_db["SWP"]["PATHLENGTH"]=path_length
comparison_db["SWP"]["MAXFLOWDECREASE"]=max_flow_tracker



#Comparison Plotter
algorithm_list=list(comparison_db.keys())
```

```
metric_list=list(comparison_db["MIRA"].keys())

#Comparison Rejection Plotter

rejection_list={}
for alg in algorithm_list:
    comparison_db[alg]["TOTALREJECTION"]=sum(comparison_db[alg]["REJECTION"].values())
    rejection_list[alg]=comparison_db[alg]["TOTALREJECTION"]
plt.figure(figsize=(15,10))
plt.bar(rejection_list.keys(),rejection_list.values())
plt.xlabel("Algorithm")
plt.ylabel("Rejection Count")

#Path Length Plotter

pathlength_list={}
for alg in algorithm_list:
    comparison_db[alg]["TOTALPATHLENGTH"]=sum(comparison_db[alg]["PATHLENGTH"].values())/4
    rejection_list[alg]=comparison_db[alg]["TOTALPATHLENGTH"]
plt.figure(figsize=(15,10))
plt.bar(rejection_list.keys(),rejection_list.values())
plt.xlabel("Algorithm")
plt.ylabel("PATH LENGTH")


#Max Flow Decrease Plotter

plt.figure(figsize=(15,5))
alg="MIRA"
for alg in algorithm_list:
    comparison_db[alg]["TOTALMAXFLOWDECREASE"]=0
    mf_list=[]
    total_mf_list={}
    for alg in algorithm_list:
        mf_list=[]
        for keys,values in comparison_db[alg]["MAXFLOWDECREASE"].items():
            mf_list.append(values)
        temp=[]
        index=0
        t_val=0
        for i in range(4000):
            for lst in mf_list:
                t_val+=lst[i]
                #print(al,t_val,i)
            temp.append(t_val)
            t_val=0
        total_mf_list[alg]=temp

for alg in algorithm_list:
    plt.plot(range(0,4000),total_mf_list[alg])
plt.legend(algorithm_list)
plt.xlabel("Instance")
plt.ylabel("Sum of Maxflow Value")
```

```
#Specific Link Usage Plotter

plt.figure(figsize=(15,5))
for alg in algorithm_list:
    plt.plot(range(0,4001),comparison_db[alg]["FLOWLINKDECREASE"][(2,3)]) #Instead of Link (2,3), Any link can be specified to get the plots
plt.legend(algorithm_list)
plt.xlabel("Instance")
plt.ylabel("Link Flow Value")
```

3.2 OUTPUT

No Path Found from 4 to 2
No Path Found from 4 to 2
No Path Found from 5 to 15
No Path Found from 5 to 9
No Path Found from 1 to 13
No Path Found from 4 to 2
No Path Found from 13 to 2
No Path Found from 13 to 2
No Path Found from 4 to 2
No Path Found from 1 to 13
No Path Found from 1 to 13
No Path Found from 5 to 15
No Path Found from 4 to 2
No Path Found from 5 to 15
No Path Found from 5 to 15
No Path Found from 13 to 2
No Path Found from 4 to 2
No Path Found from 5 to 9
No Path Found from 5 to 15
No Path Found from 13 to 2
No Path Found from 5 to 15
No Path Found from 1 to 13
No Path Found from 1 to 13
No Path Found from 5 to 15
No Path Found from 1 to 13
No Path Found from 5 to 9
No Path Found from 4 to 2
No Path Found from 1 to 13
No Path Found from 4 to 2
No Path Found from 13 to 2
No Path Found from 1 to 13
No Path Found from 4 to 2
No Path Found from 5 to 15
No Path Found from 13 to 2
No Path Found from 13 to 2
No Path Found from 5 to 9
No Path Found from 1 to 13
No Path Found from 1 to 13
No Path Found from 1 to 13
No Path Found from 5 to 9
No Path Found from 1 to 13
No Path Found from 4 to 2
No Path Found from 5 to 15
No Path Found from 5 to 15
No Path Found from 4 to 2
No Path Found from 5 to 9
No Path Found from 5 to 9
No Path Found from 4 to 2
No Path Found from 1 to 13

No Path Found from 1 to 13
No Path Found from 4 to 2
No Path Found from 13 to 2
No Path Found from 13 to 2
No Path Found from 5 to 9
No Path Found from 4 to 2
No Path Found from 4 to 2
No Path Found from 5 to 15
No Path Found from 5 to 9
No Path Found from 13 to 2
No Path Found from 13 to 2
No Path Found from 1 to 13
No Path Found from 5 to 15
No Path Found from 5 to 15
No Path Found from 4 to 2
No Path Found from 5 to 15
No Path Found from 5 to 9
No Path Found from 1 to 13
No Path Found from 5 to 15
No Path Found from 4 to 2
No Path Found from 1 to 13
No Path Found from 4 to 2
No Path Found from 5 to 15
No Path Found from 5 to 15
No Path Found from 5 to 9
No Path Found from 5 to 9
No Path Found from 13 to 2
No Path Found from 5 to 9
No Path Found from 5 to 15
No Path Found from 1 to 13
No Path Found from 13 to 2
No Path Found from 1 to 13
No Path Found from 5 to 15
No Path Found from 1 to 13
No Path Found from 4 to 2
No Path Found from 13 to 2
No Path Found from 1 to 13
No Path Found from 1 to 13
No Path Found from 4 to 2
No Path Found from 4 to 2
No Path Found from 1 to 13
No Path Found from 4 to 2
No Path Found from 1 to 13
No Path Found from 4 to 2
No Path Found from 5 to 9
No Path Found from 13 to 2
No Path Found from 5 to 9
No Path Found from 13 to 2
No Path Found from 1 to 13
No Path Found from 1 to 13
No Path Found from 13 to 2
No Path Found from 4 to 2

No Path Found from 13 to 2
No Path Found from 13 to 2
No Path Found from 4 to 2
No Path Found from 13 to 2
No Path Found from 13 to 2
No Path Found from 13 to 2
No Path Found from 5 to 9
No Path Found from 4 to 2
No Path Found from 5 to 9
No Path Found from 1 to 13
No Path Found from 1 to 13
No Path Found from 13 to 2
No Path Found from 5 to 9
No Path Found from 4 to 2
No Path Found from 1 to 13
No Path Found from 13 to 2
No Path Found from 5 to 9
No Path Found from 5 to 9
No Path Found from 13 to 2
No Path Found from 1 to 13
No Path Found from 1 to 13
No Path Found from 5 to 15
No Path Found from 1 to 13
No Path Found from 13 to 2
No Path Found from 5 to 9
No Path Found from 5 to 15
No Path Found from 13 to 2
No Path Found from 4 to 2
No Path Found from 4 to 2
No Path Found from 13 to 2
No Path Found from 5 to 9
No Path Found from 1 to 13
No Path Found from 13 to 2
No Path Found from 4 to 2
No Path Found from 5 to 15
No Path Found from 13 to 2
No Path Found from 5 to 15
No Path Found from 5 to 9
No Path Found from 5 to 9
No Path Found from 13 to 2
No Path Found from 5 to 9
No Path Found from 13 to 2
No Path Found from 5 to 9
No Path Found from 5 to 9
No Path Found from 1 to 13
No Path Found from 5 to 9
No Path Found from 5 to 15
LEXMAX
ACCEPTED!!!! 3849
NOT ACCEPTED!!! 151

CHAPTER 5
CONCLUSION AND FUTURE DIRECTION

5.1 CONCLUSION

The 3 main traffic engineering objectives: minimizing network cost, load balancing and minimizing the interference.

Minimizing network cost is an important objective. In order to do this, people usually use static metrics, such as hop count or link static costs in routing algorithms. An important example is Min Hop Algorithm(MHA) algorithm. Since MHA selects the shortest path from source to destination, less network resources are used by traffic flowing through the network. MHA's complexity is low, and is easy to implement and deploy. But it may potentially cause some links being bottleneck, which will leads to poor resource utilization.

In order to restrain the production of bottleneck links and hot spots, the other optimization objective - load balancing should be incorporated in. Widest-shortest-path (WSP) and shortest-widest-path (SWP) try to consider the path's capacity when choosing the route. SWP chooses the path which has the maximum bandwidth. If several such paths exist, SWP chooses the shortest one. WSP is on the contrary of the SWP. WSP chooses the shortest path. If there are several such paths, WSP chooses the one which has the maximum bandwidth. But these two algorithms lean to preserve network resources or lean to load balance.

Another important objective - minimizing interference between the source-destination pairs in the network is proposed by M. Kodialam et al. The Minimum interference routing algorithm (MIRA) is the first one which attempted to reduce blocking probabilities by finding links minimizing the maximum flow reduction between other SD pairs. MIRA chooses the path which interferes the least to other SD pairs. For a given SD pair (s, d), the concept of interference on (s, d) is the decrease of maximum flow (maxflow) value between (,) due to route a LSP on other SD pairs. The problem of minimum interference routing is to find a path that maximizes the maxflow between all other SD pairs. This problem is shown to be NP hard. But M. Kodialam et al. give a heuristic algorithm. The core notion of MIRA is "critical link". These links have the property that when an LSP is routed over those links the maxflow values of one or more SD pairs decreases. The critical links are the arcs belonging to the min-cuts of the SD pair. MIRA tries to avoid routing LSPs on such critical links of other SD pairs. So it performs better than former algorithms.

But the notion of "critical link" defined by MIRA also has some shortcomings. For example, some links which are believed as non-critical by MIRA are shown to be indeed very important. For solving this problem, M.S. Kodialam et al. try to model the potential critical links using the concept of Δ critical link

Usually there is a tradeoff among these three factors. So many authors try to combine different objectives in the algorithm.

Karl Hendling et al. proposed Residual Network and Link Capacity (RNLC) routing algorithm. They define a constant C to control the dynamic behavior of the algorithm. If C is chosen very big, the scheme behaves like minimum-hop routing. If C is chosen very small, the scheme trends to distribute the load throughout the network.

Kavitha Banglore proposed hybrid algorithm considering these three objectives together.In Banglore s' work, the authors give an integer linear programming formulation for the problem of achieving the balance between the three factors. The authors also proved it to be NP-hard and gave a heuristic solution.

Based on the above rejection table, it is good to choose BCRA, LEXMAX, RCF and RNLC. Since they have very similar rejection counts(=150).

5.2 FUTURE DIRECTION

In this project, we had taken into account a basic MPLS network, and then we considered various algorithms that had different objectives like maximising load balancing, minimizing usage of critical links, reducing blocking probability and future rejection requests etc. All the algorithms reviewed were online dynamic algorithms. In many algorithms, we used parameters that emphasised the particular importance towards a certain objective. We even ran trials, mainly to observe the drop/rise in rejection requests on the modification of these parameters. However, we did not experiment with the satisfaction of other objectives on changing these parameters. Also, we didn't talk about the dynamic modification of these parameters on the running of a particular algorithm. It could be very possible, that in the initial stages, in order to secure the future demand, interference might be an important objective. However, as time progresses this objective might lose importance, and another objective may gain importance(for eg: Load Balancing etc). Our future aim, is to run more experiments and see temporally which objective matters the most, and how does it help in satisfying other objectives. For Eg: Doing Load Balancing early might secure future demands, but it might lead to longer path lengths.

CHAPTER 6

REFERENCES

Kotti, Afef, Rached Hamza, and Kamel Bouleimen. "Bandwidth constrained routing algorithm for MPLS traffic engineering." In *International Conference on Networking and Services (ICNS'07)*, pp. 20-20. IEEE, 2007.

Kodialam, Murali, and T. V. Lakshman. "Minimum interference routing with applications to MPLS traffic engineering." In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No. 00CH37064)*, vol. 2, pp. 884-893. IEEE, 2000.

Olszewski, Ireneusz. "The improved least interference routing algorithm." In *Image Processing and Communications Challenges 2*, pp. 425-433. Springer, Berlin, Heidelberg, 2010.

Bagula, Antoine B., Marlene Botha, and Anthony E. Krzesinski. "Online traffic engineering: the least interference optimization algorithm." In *2004 IEEE International Conference on Communications (IEEE Cat. No. 04CH37577)*, vol. 2, pp. 1232-1236. IEEE, 2004.

Hendling, Karl, Gerald Franzl, Brikena Statovci-Halimi, and Artan Halimi. "Residual network and link capacity weighting for efficient traffic engineering in MPLS networks." In *Teletraffic Science and Engineering*, vol. 5, pp. 51-60. Elsevier, 2003.

Meng, Zhaowei, Jinshu Su, and Stefano Avallone. "A new hybrid traffic engineering routing algorithm for bandwidth guaranteed traffic." In *Asian Internet Engineering Conference*, pp. 159-171. Springer, Berlin, Heidelberg, 2006.

Banglore, Kavitha, and Whalley Dr Lois Hawkes. "A Minimum Interference Hybrid Algorithm for MPLS Networks." (2002).

Kar, Koushik, Murali Kodialam, and T. V. Lakshman. "MPLS traffic engineering using enhanced minimum interference routing: An approach based on lexicographic max-flow." In *2000 Eighth International Workshop on Quality of Service. IWQoS 2000 (Cat. No. 00EX400)*, pp. 105-114. IEEE, 2000.