

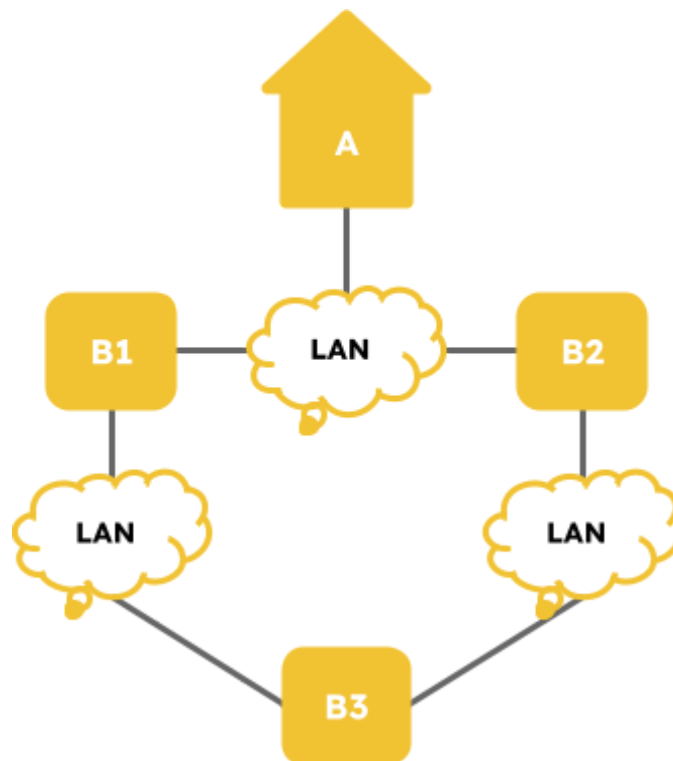
Homework 3: Bridging, Routing, and IP Forwarding

Due on Tuesday, November 12th, 2024 at 2:00 pm (Week 7)

Note: Question 5 is merely for exam practice; do not turn it in. We will also just pick **two** of the four mandatory questions to grade.

Question 1: Bridging and Loops

The Spanning Tree algorithm cannot prevent temporary loops (for example, if two separate LANs are connected by a bridge, and then someone plugs them together using a repeater). Eventually, the loop will be broken, and the right bridges will turn off. However, packets can circulate at very high speeds until the loop is broken.



Alyssa P. Hacker has thought of a way to improve the situation during temporary loops. Consider a temporary loop of bridges shown above and assume that bridges **B1**, **B2**, and **B3** all think they are ON. Suppose **A** sends a multicast packet. Both **B1** and **B2** pick up the packet, and the packet will circulate in two directions, both clockwise and anti-clockwise. This will also happen if the destination is unknown.

Questions

1. **Modifying Bridge Learning** (15 points)

Based on Alyssa's observation, how could you modify the bridge learning and forwarding to prevent multicast and unknown destination packets from circulating continuously in a temporary loop?

Bridge learning already defines a mapping between interfaces and source MAC addresses. To temporarily break loops, we can simply drop packets that arrive from an interface that's different than expected.

2. **Dropped Packets** (10 points)

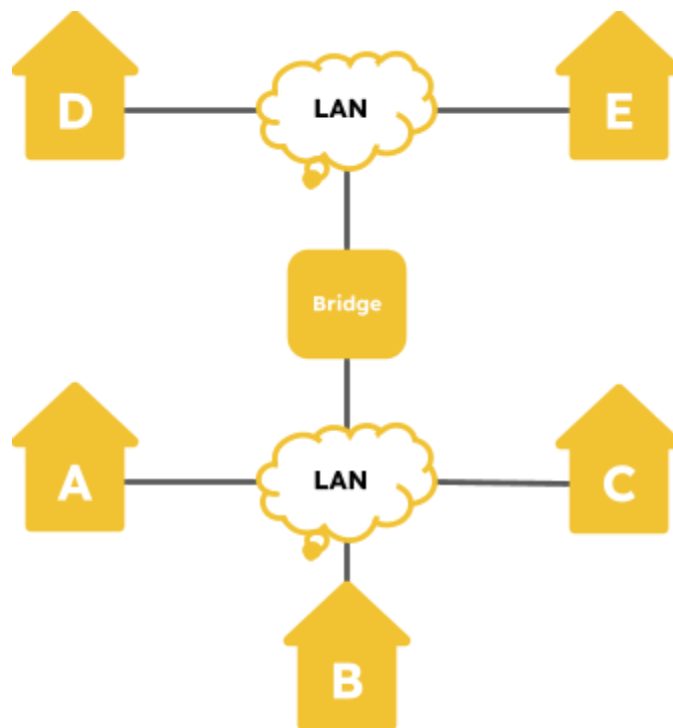
What goes wrong with your method if packets can be dropped?

Let's say that **A** sends a packet to the upper LAN, but for whatever reason, it's dropped on the way to **B2**. **B1** will only send **A**-originating packets from the **right** interface. **B2** will now receive the packet from the **lower** interface through **B3**, so it will only send **A**-originating packets from the **lower** interface. This creates a loop.

Question 2: IP Broadcast Storms, Bridges vs. Routers

(Adapted from Perlman's book.) A broadcast storm is an event that causes a flurry of messages. One implementation that caused broadcast storms was the Berkeley UNIX endnode IP implementation. In this implementation, an endnode attempts to forward a packet that it mysteriously receives with a network layer (IP) address that is different from itself. This is what you would do if you found a neighbor's letter wrongly placed in your mailbox. However, this seemingly helpful policy can cause problems.

Consider the figure below which shows 2 LANs connected by a bridge, with several IP endnodes on each LAN. There are no IP routers. All IP endnodes are configured with the same mask and so can tell that they have the same net number/prefix. Suppose IP endnode A is incorrectly configured and incorrectly thinks its data link address is all 1's. The data link address of all 1's is the broadcast address: any packet sent to such an address is received by all stations on a LAN (it is the ultimate multicast address!).



Questions

1. **Address Resolution Protocol** (10 points)

What happens when another IP endnode D decides to send a packet to IP endnode A? Assume that D initially does not have A's data link address in its cache, and so must do the ARP protocol. Give the sequence of events.

1. D broadcasts an ARP request for A's IP address.

2. A responds with the faulty broadcast address.
3. D then sends the data packet (IP addressed to A) to everyone in the network (the bridge forwards this packet since it has the destination as the faulty broadcast address).
4. All the hosts running the Berkeley IP endnode implementation attempt to forward this packet to A. They will all send an ARP request for A.
5. A responds to all these requests with the faulty broadcast address.
6. All the hosts will then attempt to forward the data packet to A, but with the faulty broadcast address.
7. All these hosts will receive these packets and attempt to forward the data packet to A. (For each iteration k , there will be T^k (the total number of endnodes in the network of Ethernets) packets sent.)

2. **IP Router** (15 points)

Suppose the bridge is replaced by an IP router. (Of course, the masks at the nodes must be changed so that there are now two masks, one for each LAN. Note a mask is just a bitmap of 1's in the most significant bits that tells you how long the prefix for that subnet is) The problem does not disappear, but it does get a little better. Explain as precisely as you can the improvement using two parameters: T the total number of endnodes in the network of Ethernets, and M the maximum number of endnodes in a single Ethernet.

In 2.1, we know that the number of packets sent is T^k . If we add an IP router, it will request the MAC address of A itself. This limits the storm to the lower subnet, or M^k .

Question 3: IP Forwarding, Longest Matching Prefix using Tries

To prepare for the Arista interview, you give ChatGPT the prompt:

“Can you write C code to do longest matching prefix of a 32-bit destination IP address in an Ethernet packet with an IP routing header which contains the destination IP address at the right offset. Do longest matching prefix on a database of prefixes called ForwardingTable implemented as a binary trie. The result should return the output interface and IP address of the next hop.”

ChatGPT's reply: Here's a basic implementation of a longest prefix match using a binary trie data structure in C. This code will parse the destination IP address from the Ethernet packet, traverse the trie to find the longest matching prefix, and return the output interface and next hop IP address.

In this example, I assume:

1. Each trie node has children for **0** and **1** bits.
2. Each leaf node contains the next hop information, including the output interface and next hop IP.
3. The trie is built ahead of time and passed to the longest prefix match function.

Here's the code:

```
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

// Define a structure for the trie node
typedef struct TrieNode {
    struct TrieNode *children[2]; // 0 and 1 children
    int is_end_of_prefix;        // Q2: Why is this field needed?
    uint32_t next_hop_ip;        // IP address of the next hop
    int output_interface;        // Output interface
} TrieNode;

// Initialize a new trie node
TrieNode* create_node() {
    TrieNode *node = (TrieNode*)malloc(sizeof(TrieNode));
    node->children[0] = node->children[1] = NULL;
    node->is_end_of_prefix = 0;
    node->next_hop_ip = 0;
    node->output_interface = -1;
```

```

    return node;
}

// Insert a prefix into the trie
void insert_prefix(TrieNode *root, uint32_t prefix, int prefix_length, uint32_t
next_hop_ip, int output_interface) {
    TrieNode *node = root;
    for (int i = 31; i >= 32 - prefix_length; i--) {
        int bit = (prefix >> i) & 1;
        if (!node->children[bit]) {
            node->children[bit] = create_node();
        }
        node = node->children[bit];
    }
    node->is_end_of_prefix = 1; Q3: Why do this?
    node->next_hop_ip = next_hop_ip;
    node->output_interface = output_interface;
}

// Perform the longest prefix match
int longest_prefix_match(TrieNode *root, uint32_t dest_ip, uint32_t *next_hop_ip,
int *output_interface) {
    TrieNode *node = root;
    TrieNode *last_matching_node = NULL;
    for (int i = 31; i >= 0; i--) {
        int bit = (dest_ip >> i) & 1;
        if (!node->children[bit]) break;
        node = node->children[bit];
        if (node->is_end_of_prefix) {
            last_matching_node = node;
        }
    }
    if (last_matching_node) { // Q4: Why do this?
        *next_hop_ip = last_matching_node->next_hop_ip;
        *output_interface = last_matching_node->output_interface;
        return 1; // Match found
    }
    return 0; // No match found
}

// Sample Ethernet packet structure
typedef struct {

```

```

uint8_t destination_mac[6];
uint8_t source_mac[6];
uint16_t ethertype;
uint8_t payload[1500];
} EthernetPacket;

// Extract destination IP from an IP header within the Ethernet payload
uint32_t extract_destination_ip(EthernetPacket *packet) {
    return ((uint32_t)packet->payload[16] << 24) |
           ((uint32_t)packet->payload[17] << 16) |
           ((uint32_t)packet->payload[18] << 8) |
           ((uint32_t)packet->payload[19]);
} // Q5: Explain why the code uses the constants 16, 17, 18, 19. Explain shifts
and ORs.

int main() {
    TrieNode *root = create_node();

    // Example: Insert prefixes into the trie
    insert_prefix(root, 0xC0A80000, 16, 0xC0A80001, 1);
    // 192.168.0.0/16 -> next hop 192.168.0.1, interface 1
    insert_prefix(root, 0xC0A80100, 24, 0xC0A80101, 2);
    // 192.168.1.0/24 -> next hop 192.168.1.1, interface 2

    // Example Ethernet packet with destination IP 192.168.1.5
    EthernetPacket packet;
    memset(&packet, 0, sizeof(EthernetPacket));
    packet.payload[16] = 192;
    packet.payload[17] = 168;
    packet.payload[18] = 1;
    packet.payload[19] = 5;

    uint32_t dest_ip = extract_destination_ip(&packet);
    uint32_t next_hop_ip;
    int output_interface;

    if (longest_prefix_match(root, dest_ip, &next_hop_ip, &output_interface)) {
        printf("Match found:\n");
        printf("Next hop IP: %u.%u.%u.%u\n",
               (next_hop_ip >> 24) & 0xFF,
               (next_hop_ip >> 16) & 0xFF,
               (next_hop_ip >> 8) & 0xFF,

```

```

    next_hop_ip & 0xFF);
    printf("Output interface: %d\n", output_interface);
} else {
    printf("No matching prefix found.\n");
}
// Free trie memory (not shown here)
return 0;
}

```

Questions

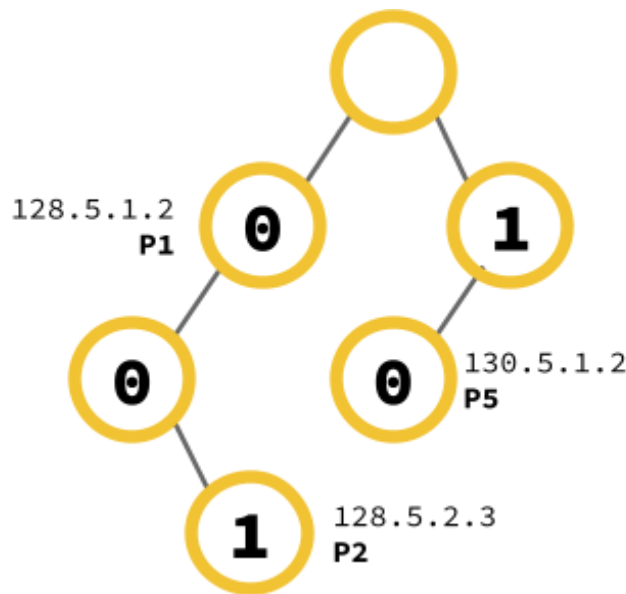
1. Trie Structure (10 points)

Draw an example of the trie data structure after you insert three prefixes:

0* whose next hop is **128.5.1.2** and interface is **P1**,

001* whose next hop is **128.5.2.3** and interface is **P2**, and

10* whose next hop is **130.5.1.2** and interface is **P5**.



2. End of Prefix (3 points)

What is the point of `is_end_of_prefix` (see comment)? This seems useless at first because the loops for insertion and search terminate when they see a null pointer. Drawing the picture for part 1 will help as it will for any code you get from ChatGPT.

The end of a prefix isn't always once you terminate your trie search. For example, what if your IP address starts with three **0s**? **000*** isn't a valid prefix, so you have to keep track of which prefixes are valid.

3. Resetting End of Prefix (3 points)

Again, when you insert a prefix the code sets `is_end_of_prefix` to 1 (see comment). Why is that needed and why does it not need to be reset?

This is needed since we need to indicate which nodes in our trie have next hops. Note that we don't ever need to reset `is_end_of_prefix` since we don't have a mechanism to remove prefixes. Also, when adding longer prefixes with a shared sub prefix, we need to be able to distinguish between them.

4. Last Matching Node (3 points)

At the end of search (see comment), the code retrieves the next hop from `last_matching_node`. Why not pick it up directly as the code walks the trie?

We don't want to greedily take the first end of prefix. Since we're looking for the longest prefix match, we'll want to find the last matching node.

5. IP Addresses (3 points)

The code that retrieves the Dest IP address (see comment) has some interesting constants including 16, 17, 18, and 19. It also does some shifting and ORing. Explain why.

16, 17, 18, and 19 represent the bytes in the Ethernet payload that correspond to the destination IP address. The shifting and ORing is to convert from network order (big endian) to host order (little endian on most machines).

6. Missing Code (3 points)

Explain what code may be missing besides deallocating memory.

Routers need a default address to forward to. This code as written does not do such a thing. Other answers may include:

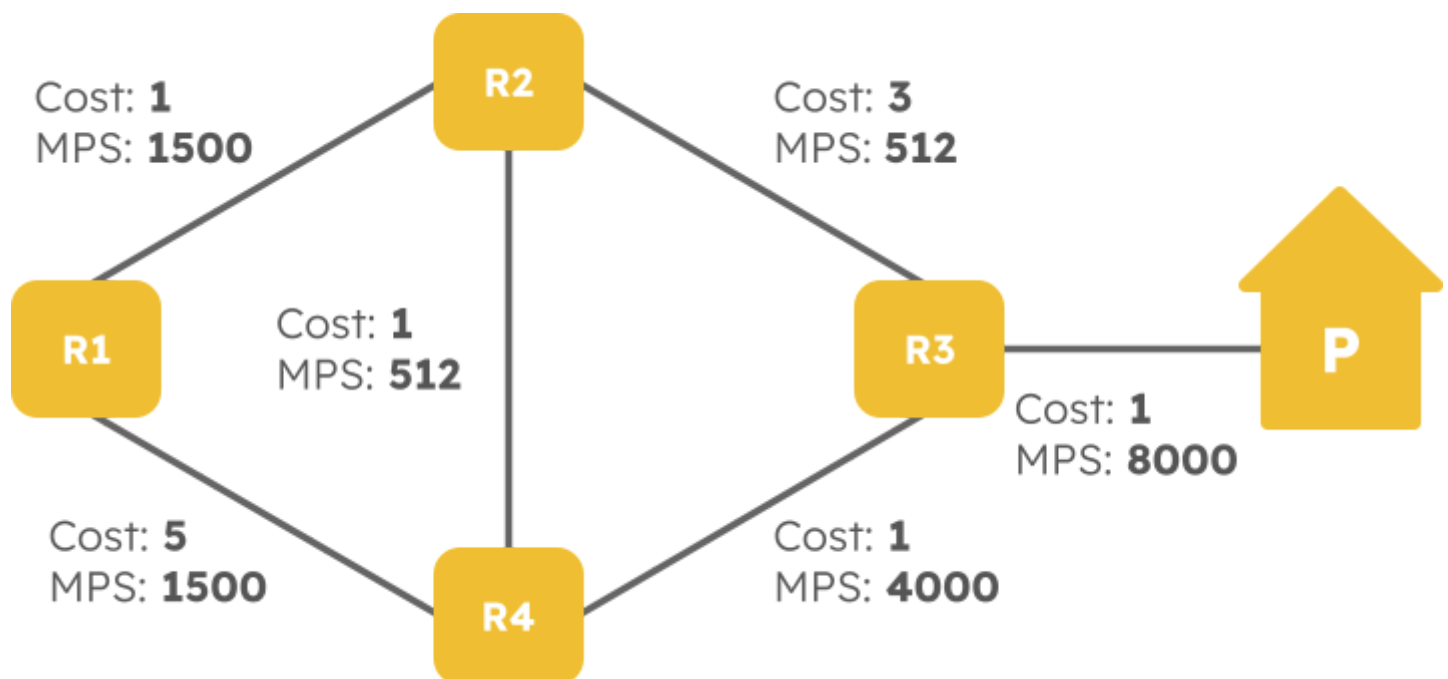
- Input sanitation/format checking
- Removing prefixes
- Modifying prefixes (changing next hops)

Really, anything reasonable receives credit.

Question 4: Modifying Routing to Avoid Fragmentation

We learned in class that in order to avoid fragmentation, IP tries various packet sizes until it finds one that works. Alternatively, we could modify the routing protocol to compute the minimum of the maximum packet sizes of all links on the best route to each destination.

In distance vector routing, a router **R** computes its own distance **Distance(P, R)** to a destination prefix **P** using the distances sent by its neighbors as follows: **Distance(P, R) = Minimum across all neighbors N of Distance(P, N) + Distance(R, N)**. We want to see how to modify this protocol to also compute the minimum max packet size on the shortest distance route to **P**.



Questions

1. **Shortest Path** (5 points)

What is the shortest path between **R1** and **P**? What is the largest packet size that is guaranteed to get through without fragmentation on this path?

R1 -(1)-> **R2** -(1)-> **R4** -(1)-> **R3** -(1)-> **P** (4 units in total)

The largest packet size is **512** units.

2. **Minimum Max Packet Size** (5 points)

Assume each router receives the additional variables **Distance(P, N)** and **MinMaxPacketSize(P, N)** from each neighbor **N**. Write an equation to compute these two variables from the corresponding variables of all a router's neighbors.

$$\text{Distance}(\mathbf{R}, \mathbf{P}) = \min \{ \text{Distance}(\mathbf{R}, \mathbf{N}) + \text{Distance}(\mathbf{N}, \mathbf{P}) \mid \mathbf{N} \text{ in all neighbors of } \mathbf{R} \}$$

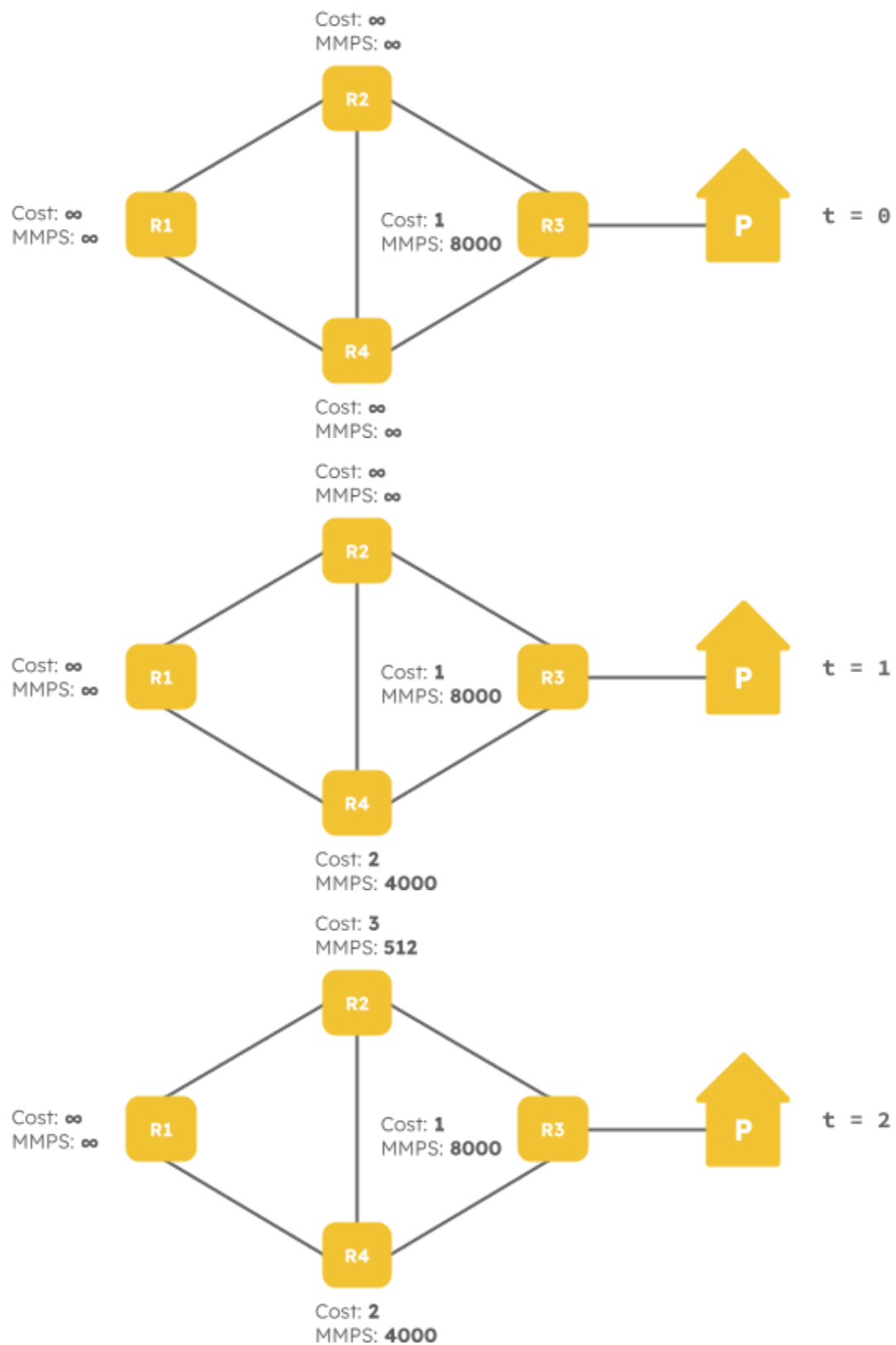
$$\text{MinMaxPacketSize}(\mathbf{R}, \mathbf{P}) = \max \{ \min(\text{MaxPacketSize}(\mathbf{R}, \mathbf{N}), \text{MinMaxPacketSize}(\mathbf{N}, \mathbf{P})) \mid \mathbf{N} \text{ in all neighbors of } \mathbf{R} \text{ such that } \mathbf{N} \text{ is the minimum distance to } \mathbf{P} \}$$

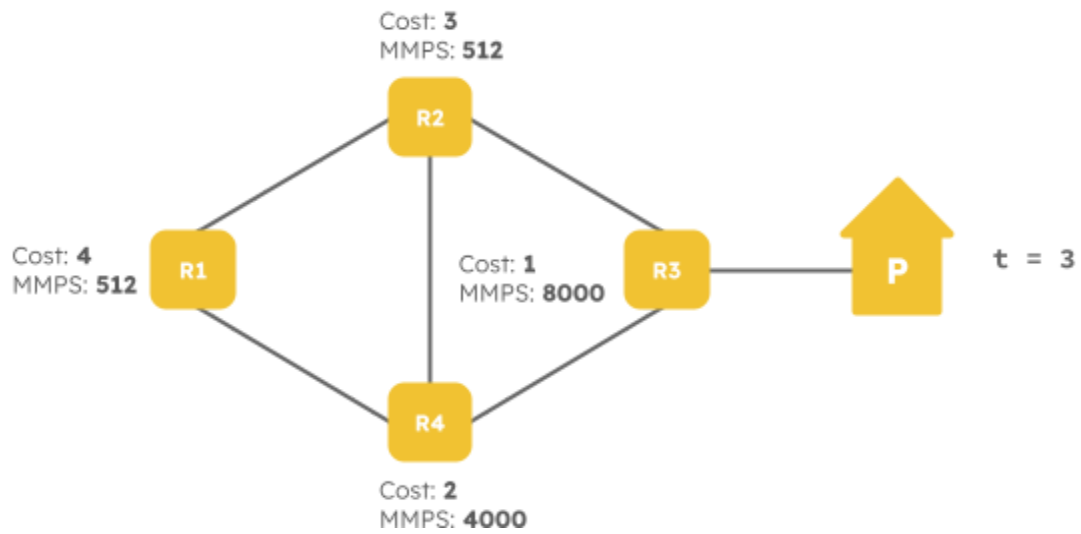
(If your second equation is a bit simpler since it doesn't consider neighbors with the same cost, that's okay).

3. Packet Size Propagation (10 points)

Assume we are calculating these estimates only for distances to **P**. Assume that at **t = 0**, router **R3** has **Distance (P, R3) = 1** and **MinMaxPacketSize (P, R3) = 8000** and all other routers have the distance and min packet size to **P** set to a default of **infinity**. Assume that at **t = 0**, each router sends an update to each neighbor. Draw several pictures of the same topology with the changing estimates of each router for its two variables based on the equation you wrote down until all estimates stop changing. After how much time do all the estimates converge (i.e. do not change any more)?

The answers below assume that each propagation happens in link cost units. Your solution may slightly differ (and that's okay) if you let each propagation happen in one unit of time.

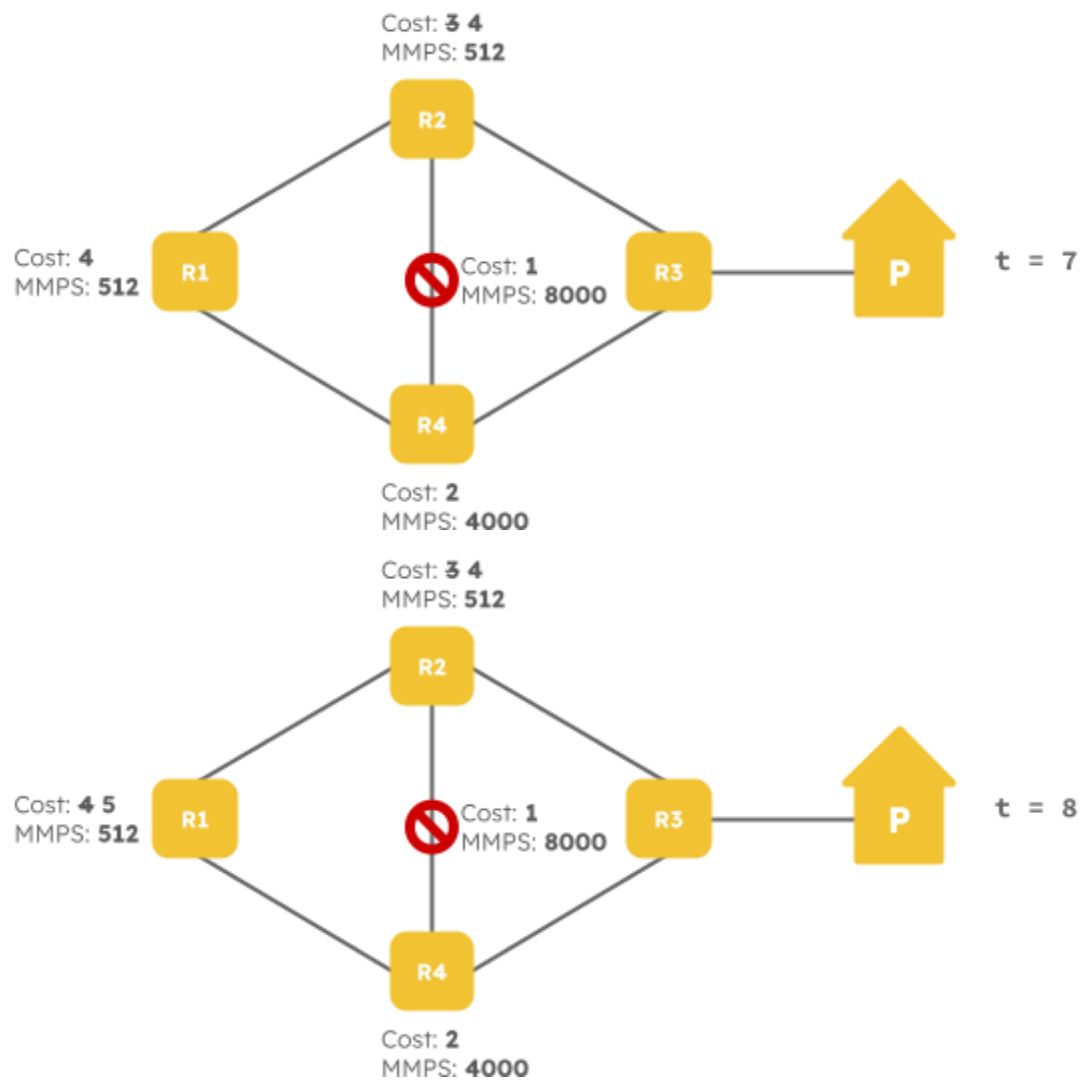




Another update occurs at $t = 6$, but it doesn't change **R1**'s calculations.

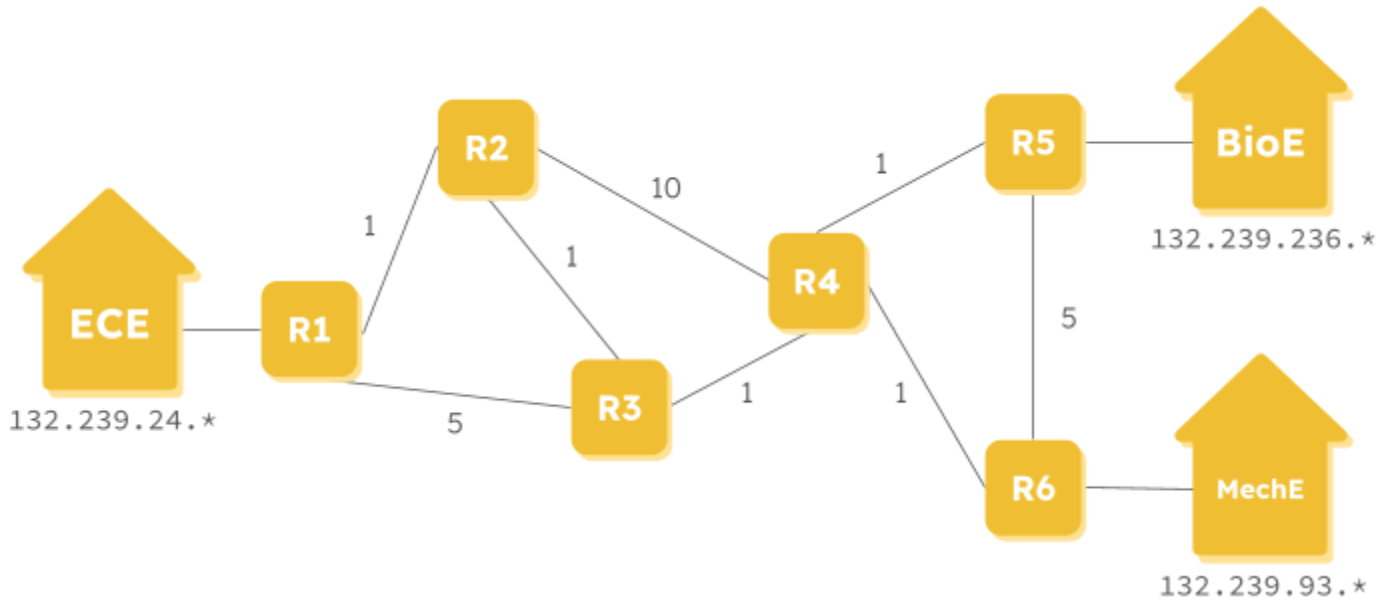
4. **Crash** (5 points)

Now assume the link to **R2** to **R4** crashes at time $t = 7$. Draw similar pictures for the time it takes to converge after the crash.



Question 5: Link State Routing and Crashes (*Optional, Exam Practice Only*)

Consider the topology shown below. This is the same topology we used in the slides to show the count up phenomenon for distance vector when both the links from **R4** to **R2**, and the link from **R4** to **R3** went down. We want to see why the count up phenomenon *does not happen with link state*.



Questions

1. **Normal Operation** (5 points)

Draw the link state packets sent by **R4**, **R5** and **R6** before any link crashes.

R4's LSP		R5's LSP		R6's LSP	
R2	10	R4	1	R4	1
R3	1	R6	5	R5	5
R5	1	BioE		MechE	
R6	1				

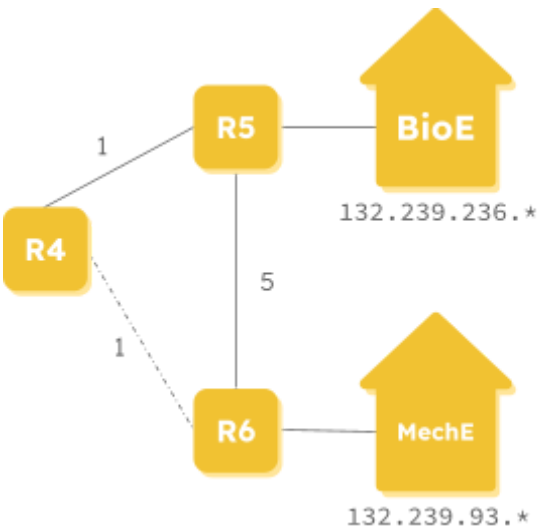
2. **Double Crash** (5 points)

Draw the link state packet sent by **R4** after the links from **R4** to **R2**, and from **R4** to **R3** crash (the links from **R5** and **R6** stay the same).

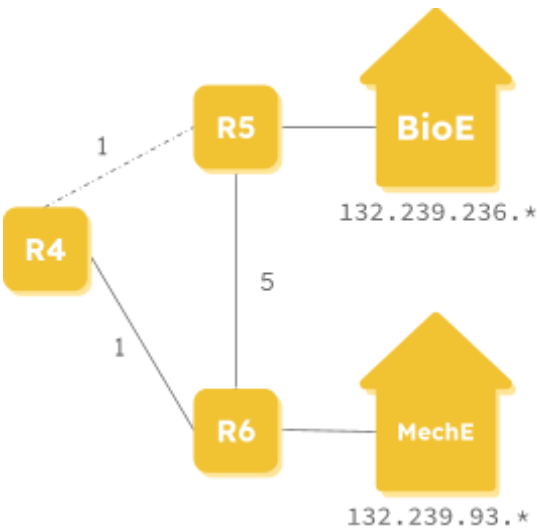
R4's LSP	
R5	1
R6	1

3. **Dijkstra Tree** (15 points)
- Assume this new LSP from **R4** gets to **R5** and **R6** in a few msec. Show the Dijkstra tree at **R5** and **R6**, and explain why after **R5** finishes its Dijkstra calculations (after a few msec), it will conclude that **ECE** is unreachable even though it still has link state packets from **R1**, **R2**, and **R3**.

Tree at R5:



Tree at R6:



These routers won't see any new link state updates from the other routers, concluding that they're unreachable; the only link states that are updated are from **R4** and **R6** (or **R4** and **R5** for **R6**).