Kevin DeVocht
CS 312
Project 3
Sec 2

1.)See Attached Code in email. OnePath and AllPaths will be in the NetworkPath.cs file

2.)Priority Queue:

a.)Insert:

```
public void insert(int index, Node node, double distance, Node previous)
{
	index = index + 1;   O(1)
	counter++;   O(1)
	pointers[index] = counter;  O(log n)
	node.distance = distance;  O(1)
	node.previous = previous;  O(1)
	heap[counter] = node;   O(log n)
	bubbleUp(counter);  O(log n)
}

private void bubbleUp(int index)
{
	if (index == 1)  O(1)
	{
		return;
	}
	bool bubbling = true;  O(1)
	while (bubbling)
	{
		bubbling = false;  O(1)
		double heapIndex = pointers[index];   O(log n)
		if (heapIndex <= 1)  O(1)
		{
			break;
		}
		double parent = Math.Floor(heapIndex / 2);
		if (heap[(int)heapIndex].distance <
heap[(int)parent].distance)  O(log n)
		{
			exchange((int)heapIndex, (int)parent);  O(log n)
			bubbling = true;  O(1)
		}
	}
}

private void exchange(int heapIndex, int parent)
{
	Node tempNode = heap[heapIndex];  O(log n)
	heap[heapIndex] = heap[parent];  O(log n)
```

```
        heap[parent] = tempNode;  O(log n)
        pointers[heap[heapIndex].pointerIndex] = heapIndex;  O(log n)
        pointers[heap[parent].pointerIndex] = parent;  O(log n)
    }
```

Looking at the three functions that are needed to do an insert, You can see that the worst cost is $O(\log n)$ sometimes it might be more like $O(2\log n)$ but the Big-O is still $O(\log n)$. This is because you are just looking up values in arrays and setting values in arrays as well.

b.)Delete

```
    public Node delete()
    {
        Node minNode = heap[1];   O(log n)
        pointers[minNode.pointerIndex] = -1;   O(log n)
        if (counter != 1)   O(1)
        {
            heap[1] = heap[counter];   O(log n)
            pointers[heap[1].pointerIndex] = 1;   O(log n)
        }
        heap[counter] = null;   O(log n)
        counter--;   O(1)
        bubbleDown();
        return minNode;
    }

    private void bubbleDown()
    {
        int index = 1;   O(1)
        while (hasLeftChild(index))  O(log n)
        {
            int smaller = index * 2;   O(1)
            if (hasRightChild(index))   O(1)
            {
                if (heap[index * 2].distance > heap[(index * 2) +
1].distance)   O(log n)                    {
                    smaller = (index * 2) + 1;   O(1)
                }
            }
            if (heap[index].distance > heap[smaller].distance)   O(log n)
            {
                exchange(smaller, index);   O(log n)
            }
            else
            {
```

Kevin DeVocht
CS 312
Project 3
Sec 2

```
                        break;   O(1)
                    }
                    index = smaller;   O(1)
                }
            }
```

Again you can see that each function is $O(\log n)$, the while loop is only logn because you are only checking for the leftChild

c.)Decrease-Key

```
    public void decreaseKey(int index, double distance)
    {
        index = index + 1;  O(1)
        int heapIndex = pointers[index];   O(log n)
        if (distance < heap[heapIndex].distance || heap[heapIndex].distance
    == Double.PositiveInfinity)   O(log n)
        {
            heap[heapIndex].distance = distance;   O(log n)
        }
        bubbleUp(index);   O(log n)
    }
```

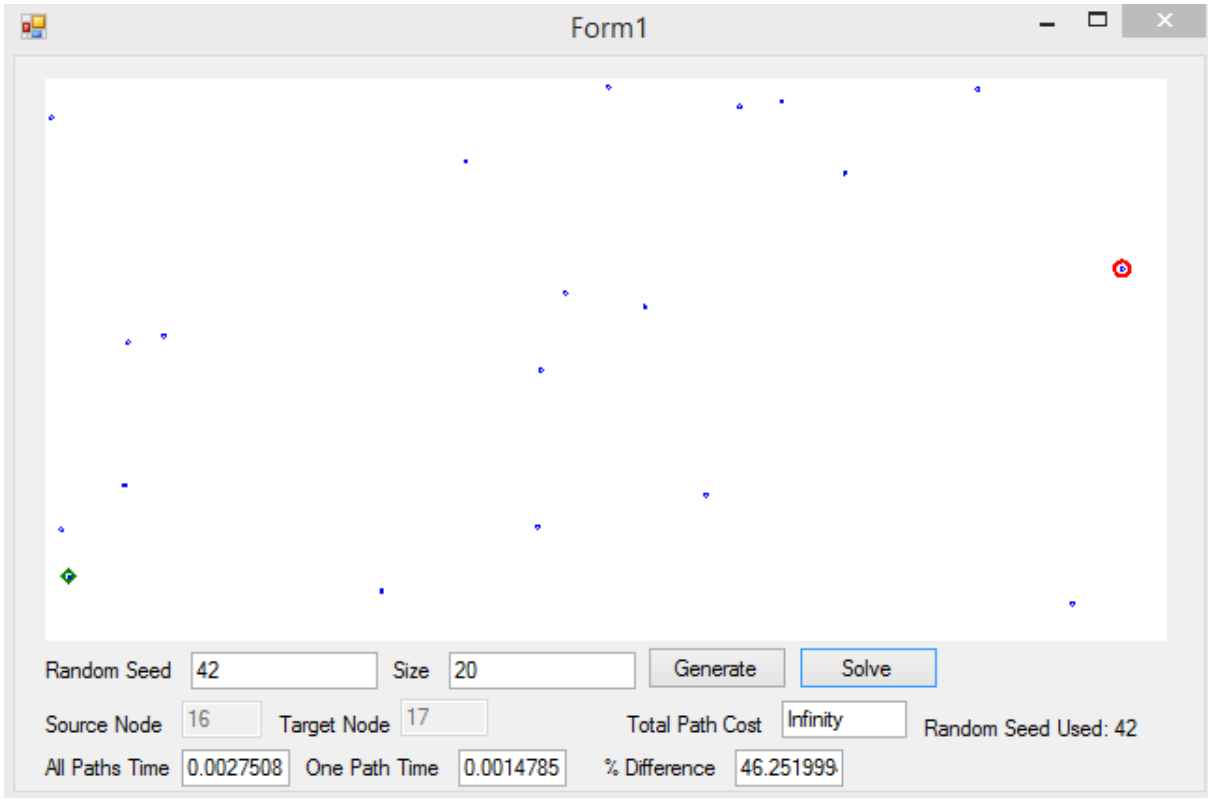
And lastly, Decrease-Key is also only $O(\log n)$

3.)We have just shown that the priority queue runs in $O(\log n)$. We can't do any better than that, we also know that for the All Paths algorithm, we need to check each node which has to be $O(n)$

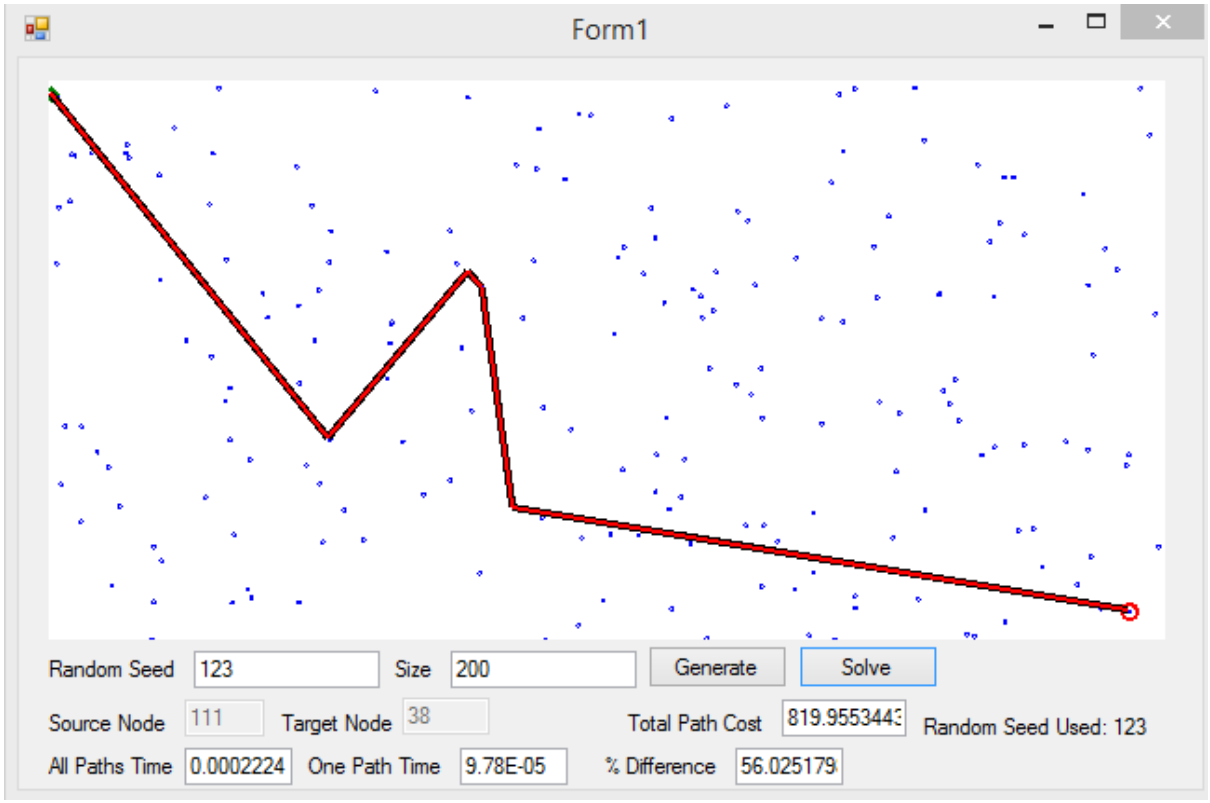
So for each node it takes $O(\log n)$ So the overall time complexity for AllPaths is $O(n\log n)$ For One path, We only put one node on the queue to start and go until the best path has been found. So worst case is $O(n\log n)$ if you have to check every node so your overall is still $O(n\log n)$.

Kevin DeVocht
CS 312
Project 3
Sec 2

4.)



Kevin DeVocht
CS 312
Project 3
Sec 2



5.)

Size	Seed	All Path	One Path	% Diff
100	10	0.0028118	0.0015153	46.10925386
	11	0.0001536	0.0000962	37.36979167
	12	0.0001421	0.0000815	42.64602393
	13	0.0002116	0.0001122	46.97542533
	14	0.0001307	0.0000872	33.28232594
			Average	41.27656414

Size	Seed	All Path	One Path	% Diff
1000	15	0.0008065	0.0005466	32.22566646
	16	0.0007947	0.0001633	79.4513653
	17	0.0007223	0.0002876	60.18274955
	18	0.0008071	0.0004612	42.85714286
	19	0.0008267	0.0005602	32.23660336
			Average	49.39070551

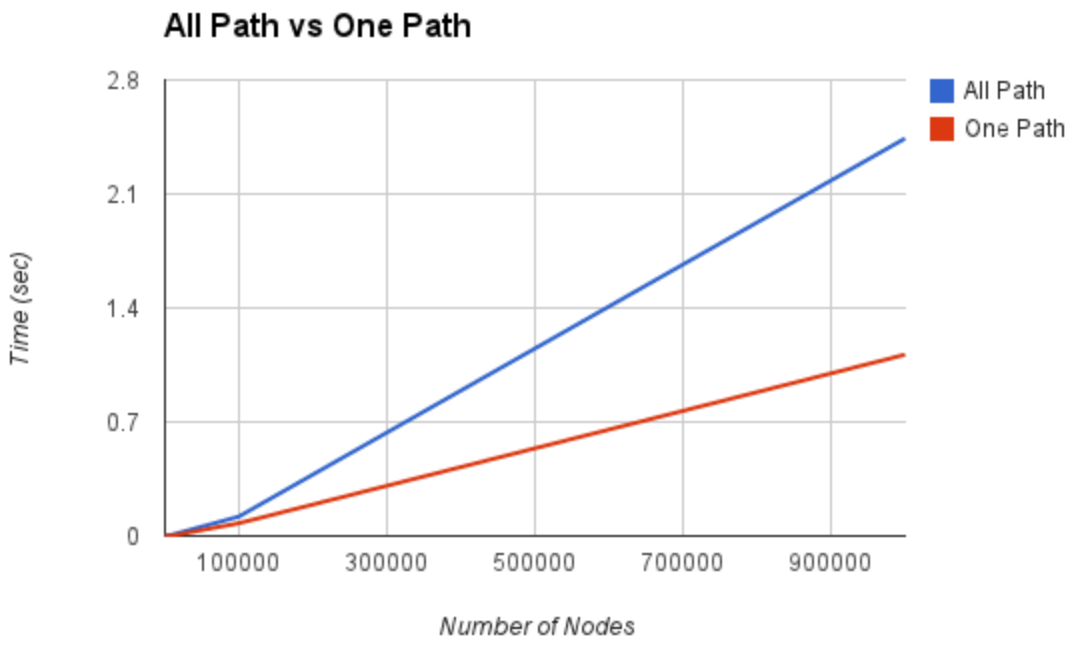
Kevin DeVocht
CS 312
Project 3
Sec 2

Size	Seed	All Path	One Path	% Diff
10000	19	0.0078075	0.0049284	36.87608069
	20	0.0080101	0.0006815	91.49199136
	21	0.0079488	0.0050289	36.73384662
	22	0.0078241	0.0016937	78.35278179
	23	0.0083587	0.0009913	88.14050032
			Average	66.31904016

Size	Seed	All Path	One Path	% Diff
100000	24	0.1198823	0.0795337	33.65684509
	25	0.1189553	0.0603246	49.28800987
	26	0.1181943	0.0515182	56.41228046
	27	0.1172902	0.1150837	1.881231339
	28	0.1223714	0.0843348	31.08291643
			Average	34.46425664

Size	Seed	All Path	One Path	% Diff
1000000	28	2.5466272	0.3910259	84.64534189
	29	2.4202346	1.8678721	22.82268421
	30	2.4751606	0.3113164	87.42237574
	31	2.3770033	1.1234273	52.73766343
	32	2.3848755	1.8719885	21.50581865
			Average	53.82677678

Kevin DeVocht
CS 312
Project 3
Sec 2



Just like we talked about in section 3, One Path is bounded by All Path but especially as the number of nodes get larger, it seems to perform faster because it is not required to visit each node like All Path has to. Looking at the percentages it looks like like One Path is about 50% faster so I would guess that the constant factor is 0.5