Kyle March

25 Oct 2017

Section 002

Project 03 – Network Routing – Write Up

1. **Code:**

//-----------------------------------------------------------------------------//

//-------------------------- Dijkstra's Algorithm -----------------------------//

//-----------------------------------------------------------------------------//

/\*\*

\* This Dijkstra's is implemented based off of the pseudocode found in the text.

\* Time complexity for the Binary Heap Priority queue is O(|V| \* |logV|)

\* because the implementation requires all the nodes to be visited but the

\* update methods require |log V| time. The array priority queue is O(|V|^2)

\* because it has to visit all the nodes on implementation and has to itterate

\* through the nodes in order to delete and update the queue. The space complexity

\* for both is O(|V|) because each queue stores all of the nodes generated

\*\*/

private List<int> dijkstrasAlgorithm(PriorityQueue queue, bool isArray)

{

queue.makeQueue(points.Count);

List<int> previous = new List<int>();

List<double> distances = new List<double>();

// Sets all distance values to "infinity"

// sets previous as 'undefined'

for(int i = 0; i < points.Count; i++)

{

previous.Add(-1);

distances.Add(double.MaxValue);

}

// distance to start node

distances[startNodeIndex] = 0;

// checks which priority queue will be used

if(isArray)

{

queue.insert(startNodeIndex, 0);

}

else

{

queue.insert(ref distances, startNodeIndex);

}

// main loop

// loops until there arent any nodes with a permanent distance to the end node

while(!queue.isEmpty())

{

int minIndex;

if(isArray)

{

minIndex = queue.deleteMin();

}

else

{

minIndex = queue.deleteMin(ref distances);

}

PointF u = points[minIndex];

// finds the best path amongst its neighbors, if it exists

foreach(int index in adjacencyList[minIndex])

{

PointF alt = points[index];

double altDistance = distances[minIndex] + distanceBetween(u, alt);

if(altDistance < distances[index])

{

previous[index] = minIndex;

distances[index] = altDistance;

if(isArray)

{

queue.decreaseKey(index, altDistance);

}

else

{

queue.decreaseKey(ref distances, index);

}

}

}

}

//queue.printQueue();

// gives the path of the nodes

return previous;

}

//--------------------------- Helper Method -----------------------------------//

private double distanceBetween(PointF u, PointF v)

{

double deltaXSqr = Math.Pow(v.X - u.X, 2);

double deltaYSqr = Math.Pow(v.Y - u.Y, 2);

double distance = Math.Sqrt(deltaXSqr + deltaYSqr);

return distance;

}

//-----------------------------------------------------------------------------//

//---------------------------- Draw Methods -----------------------------------//

//-----------------------------------------------------------------------------//

private PointF findMidPointOfPoints(int firstIndex, int secondIndex)

{

PointF midpoint = new PointF();

midpoint.X = (points[firstIndex].X + points[secondIndex].X) / 2;

midpoint.Y = (points[firstIndex].Y + points[secondIndex].Y) / 2;

return midpoint;

}

// draws the shortest path

private void drawPath(ref List<int> path)

{

int cur = stopNodeIndex;

int previousIndex = cur;

double totalPathCost = 0;

Console.Write("path nodes are: ");

foreach(int num in path)

{

Console.Write(num + ", ");

}

Console.WriteLine();

while(true)

{

cur = path[cur];

if (cur == -1) break;

Pen pen = new Pen(Color.Black, 1);

graphics.DrawLine(pen, points[cur], points[previousIndex]);

double distance = distanceBetween(points[cur], points[previousIndex]);

totalPathCost += distance;

PointF midPoint = findMidPointOfPoints(previousIndex, cur);

int distanceTruncated = (int)distance;

graphics.DrawString(distanceTruncated.ToString(), SystemFonts.DefaultFont, Brushes.Black, midPoint);

previousIndex = cur;

}

pathCostBox.Text = totalPathCost.ToString();

}

1. **Priority Queue Classes**

/\*\*

\* Abstract class that is implemented by HeapPriorityQueue and ArrayPriorityQueue

\*\*/

abstract class PriorityQueue

{

public PriorityQueue() { }

public abstract void makeQueue(int nodeNum);

public virtual int deleteMin() { return 0; }

public virtual int deleteMin(ref List<double> distances) { return 0; }

public virtual void decreaseKey(int index, double key) { }

public virtual void decreaseKey(ref List<double> distances, int index) { }

public virtual void insert(int index, double value) { }

public virtual void insert(ref List<double> distances, int index) { }

public abstract void printQueue();

public abstract bool isEmpty();

}

class HeapPriorityQueue : PriorityQueue

{

private int capacity;

private int count;

private int lastElement;

private int[] distancesHeap;

private int[] pointers;

public HeapPriorityQueue() { }

/\*\*

\* Is empty is time complexity of O(1) because it is a comparison. There is no space complexity.

\*\*/

public override bool isEmpty()

{

return count == 0;

}

/\*\*

\* This is for debugging purposes and is of time complexity O(|V|) because it itterates through all

\* the nodes in the distanceHeap array. There is no space complexity.

\*\*/

public override void printQueue()

{

Console.Write("Contents of Heap are: ");

for (int i = 1; i < capacity; i++)

{

if (distancesHeap[i] != -1) Console.Write(distancesHeap[i] + ", ");

}

Console.WriteLine();

}

/\*\*

\* Here both time and space complexity is O(|V|). Time is such because it itterates through the

\* nodes to assign values in increasing number. Space is of the same order because there are

\* two arrays that are allocated to the size V.

\*\*/

public override void makeQueue(int nodeNum)

{

distancesHeap = new int[nodeNum + 1];

pointers = new int[nodeNum];

for (int i = 1; i < nodeNum + 1; i++)

{

distancesHeap[i] = i - 1;

pointers[i - 1] = i;

}

// distancesHeap[0] = -1;

capacity = nodeNum;

count = 0;

lastElement = capacity;

}

/\*\*

\* Time complexity here is O(|logV|) because we know the minimum node and then we just have to

\* bubble up and sort the tree to accomadate for deleting the root. The worse case scenario

\* would be the height of the tree which is |logV|. There is no space complexity because there

\* are no new space allocations

\*\*/

public override int deleteMin(ref List<double> distancesArray)

{

int minValue = distancesHeap[1];

count--;

if (lastElement == -1) return minValue;

distancesHeap[1] = distancesHeap[lastElement];

pointers[distancesHeap[1]] = 1;

lastElement--;

int whileIndex = 1;

while (whileIndex <= lastElement)

{

int leftChildIndex = whileIndex \* 2;

if (leftChildIndex > lastElement) break;

if (leftChildIndex + 1 <= lastElement && distancesArray[distancesHeap[leftChildIndex + 1]] < distancesArray[distancesHeap[leftChildIndex]])

{

leftChildIndex++;

}

if (distancesArray[distancesHeap[whileIndex]] > distancesArray[distancesHeap[leftChildIndex]])

{

int temp = distancesHeap[leftChildIndex];

distancesHeap[leftChildIndex] = distancesHeap[whileIndex];

distancesHeap[whileIndex] = temp;

pointers[distancesHeap[leftChildIndex]] = leftChildIndex;

pointers[distancesHeap[whileIndex]] = whileIndex;

}

whileIndex = leftChildIndex;

}

return minValue;

}

/\*\*

\* Time complexity is O(|logV|) because it finds the new distance to update, switches it

\* and then updates the tree like in delete min. This gives the worse case to be the height

\* of the tree, |logV|. There is no space complexity because there is no new space allocations.

\*\*/

public override void decreaseKey(ref List<double> distancesArray, int index)

{

int heapIndex = pointers[index];

count++;

int whileIndex = heapIndex;

while (whileIndex > 1 && distancesArray[distancesHeap[whileIndex / 2]] > distancesArray[distancesHeap[whileIndex]])

{

int temp = distancesHeap[whileIndex / 2];

distancesHeap[whileIndex / 2] = distancesHeap[whileIndex];

distancesHeap[whileIndex] = temp;

pointers[distancesHeap[whileIndex / 2]] = whileIndex / 2;

pointers[distancesHeap[whileIndex]] = whileIndex;

whileIndex = whileIndex / 2;

}

}

/\*\*

\* Insert has a time complexity of O(|logV|) because it inserts the given node

\* and updates the tree by bubbling up. |log V| is the height of the tree so

\* that would be the worse case scenario. There is no space complexity because

\* there are no new space alloations.

\*\*/

public override void insert(ref List<double> distances, int pointerIndex)

{

count++;

int whileIndex = pointers[pointerIndex];

while (whileIndex > 1 && distances[distancesHeap[whileIndex / 2]] > distances[distancesHeap[whileIndex]])

{

int temp = distancesHeap[whileIndex / 2];

distancesHeap[whileIndex /2] = distancesHeap[whileIndex];

distancesHeap[whileIndex] = temp;

pointers[distancesHeap[whileIndex / 2]] = whileIndex / 2;

pointers[distancesHeap[whileIndex]] = whileIndex;

whileIndex = whileIndex / 2;

}

}

}

class ArrayPriorityQueue : PriorityQueue

{

private double[] queue;

private int count;

public ArrayPriorityQueue() { }

/\*\*

\* Time Complexity is O(1). There is no space complexity.

\* \*/

public override bool isEmpty()

{

return count == 0;

}

/\*\*

\* makeQueue time and space complexity is O(|V|) because it itterates through all the nodes.

\* it also makes a new array the same size of the number of nodes.

\* @Param the number of nodes generated

\* \*/

public override void makeQueue(int nodeNum)

{

queue = new double[nodeNum];

count = nodeNum;

for(int i = 0; i < nodeNum; i++)

{

queue[i] = double.MaxValue;

}

}

/\*\*

\* Print queue method for debugging

\* If it needs to be included because the instructions say so it is of time complexity O(N)

\* where N is the length of the queue

\*\*/

public override void printQueue()

{

Console.WriteLine("Queue: ");

for(int i = 0; i < queue.Length; i++)

{

Console.Write(i + ": " + queue[i] + ", ");

}

Console.WriteLine();

}

/\*\*

\* Time Complexity is O(1) because it just puts the key in the queue at the given index.

\* There is no space complexity.

\*\*/

public override void decreaseKey(int index, double key)

{

queue[index] = key;

}

/\*\*

\* The time complexity is O(|V|) because it itterates through all the nodes in the system.

\* There is no space complexity because there is no new arrays made.

\*\*/

public override int deleteMin()

{

double minValue = double.MaxValue;

int minIndex = 0;

for(int i = 0; i < queue.Count(); i++)

{

if(queue[i] < minValue)

{

minValue = queue[i];

minIndex = i;

}

}

count--;

queue[minIndex] = double.MaxValue;

return minIndex;

}

/\*\*

\* The time complexity of the insert method is O(1) because it directly inserts into the

\* array at the given index. There is no space complexity because there are no new

\* allocations.

\*\*/

public override void insert(int index, double value)

{

queue[index] = value;

count++;

}

}

1. **Time and Space Complexity**

/\*\*

\* This Dijkstra's is implemented based off of the pseudocode found in the text.

\* Time complexity for the Binary Heap Priority queue is O(|V| \* |logV|)

\* because the implementation requires all the nodes to be visited but the

\* update methods require |log V| time. The array priority queue is O(|V|^2)

\* because it has to visit all the nodes on implementation and has to itterate

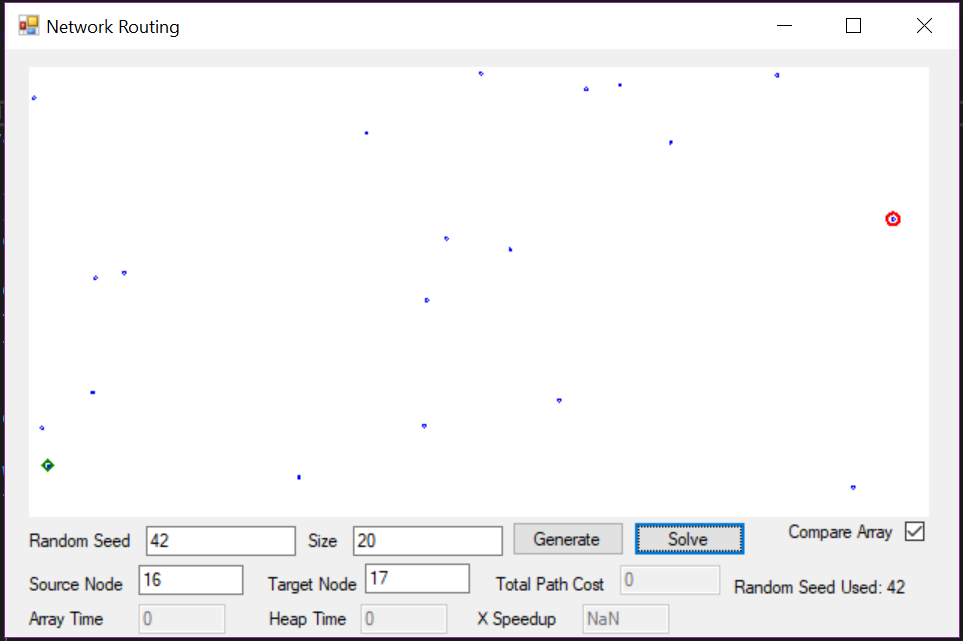
\* through the nodes in order to delete and update the queue. The space complexity

\* for both is O(|V|) because each queue stores all of the nodes generated

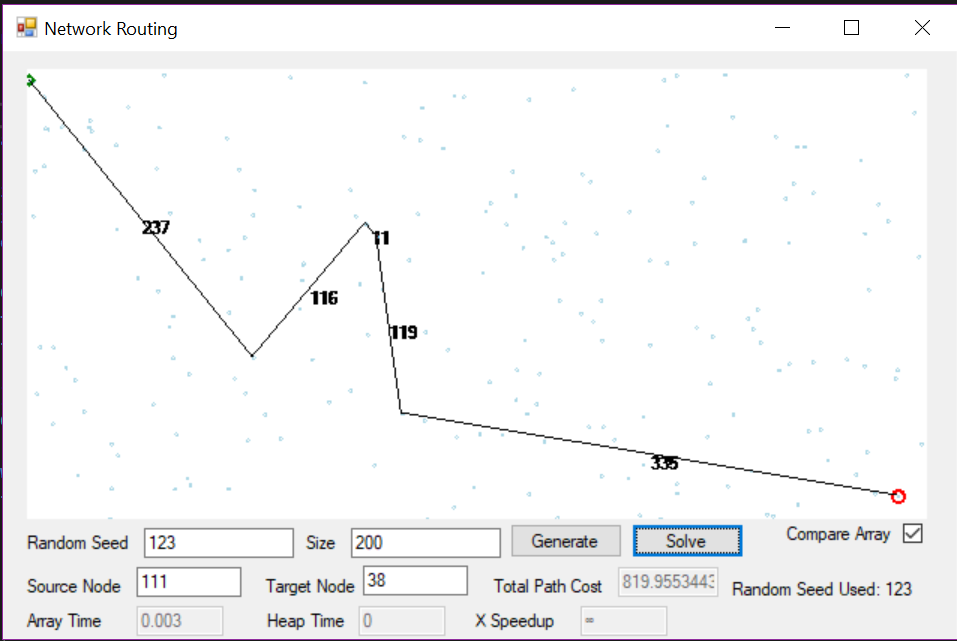
\*\*/

1. **Screenshots**

Random seed 42 – Size 20



Random seed 123 – Size 200



1. **Empirical Analysis**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array |  |  |  |  |  |  |
| Nodes | 1 | 2 | 3 | 4 | 5 | Avg |
| 100 | 0.001 | 0 | 0 | 0 | 0 | 0.0002 |
| 1000 | 0.056 | 0.058 | 0.054 | 0.055 | 0.064 | 0.0574 |
| 10000 | 4.367 | 4.348 | 4.328 | 4.521 | 4.478 | 4.4084 |
| 100000 | 99707.85 | 103278.3 | 127755.8 | 174552.8 | 161706.2 | 133400.19 |
| 1000000 | 9.77E+39 | 1.29E+40 | 1.29E+40 | 8.62E+41 | 4.68E+41 | 2.73E+41 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Heap |  |  |  |  |  |  |
| Nodes | 1 | 2 | 3 | 4 | 5 | Avg |
| 100 | 0.001 | 0 | 0 | 0 | 0 | 0.0002 |
| 1000 | 0.001 | 0.001 | 0.001 | 0.001 | 0.001 | 0.001 |
| 10000 | 0.015 | 0.012 | 0.013 | 0.015 | 0.015 | 0.014 |
| 100000 | 0.179 | 0.184 | 0.177 | 0.174 | 0.192 | 0.1812 |
| 1000000 | 4.606 | 4.487 | 4.58 | 4.553 | 4.73 | 4.5912 |

|  |  |  |
| --- | --- | --- |
| Number Nodes | **Avg Time Array** | **Avg Time Heap** |
| 100 | 0.0002 | 0.0002 |
| 1000 | 0.0574 | 0.001 |
| 10000 | 4.4084 | 0.014 |
| 100000 | 133400.19 | 0.1812 |
| 1000000 | 2.73E+41 | 4.5912 |

The results look like they are both square functions, but that’s not true looking at the time on the y axis. The array method would have taken about a lifetime to finish where as the heap implementation finished it a lot quicker. I was surprised at how dramatic the difference was between the two. It is really eye opening because the array queue is really simple to implement but not practical for scaling.