

Lab 3 Report

Part 1

```
public static List<int[,]> MatrixChainOrder(int[] pArray)
{
    int n = pArray.Length;

    int[,] mTable = new int[n, n];

    int[,] sTable = new int[n, n];

    for (int i = 0; i < n; i++)
    {
        mTable[i, i] = 0;
    }
    for (int l = 2; l < n; l++)
    {
        for (int i = 1; i < n - l + 1; i++)
        {
            int j = i + l - 1;

            mTable[i, j] = int.MaxValue;

            for (int k = i; k < j; k++)
            {
                int q = mTable[i, k] + mTable[k + 1, j] +
                    (pArray[i - 1] * pArray[k] * pArray[j]);

                if (q < mTable[i, j])
                {
                    mTable[i, j] = q;
                    sTable[i, j] = k;
                }
            }
        }
    }
    List<int[,]> result = new List<int[,]>();

    result.Add(mTable);

    result.Add(sTable);

    return result;
}
```

For this method, we take in an array of integers that represent the dimensions of a chain of matrices being multiplied together. First, we create two $n \times n$ arrays where n is the length of the given pArray. Then, after initializing the diagonals of the mTable to zeros, we go through a series of complex loops to determine the number of scalar multiplications needed to find the product of the matrices we are looking at by that point. Once we determine the minimum value of the options we have, we set the mTable of that spot to the minimum, and the sTable holds the corresponding k value. Once both tables have been populated, they are returned as elements of a list.

Part 2

```
public static List<int[,]> MemoizedMatrixChain(int[] pArray)
{
    int n = pArray.Length;

    int[,] mTable = new int[n, n];

    int[,] sTable = new int[n, n];

    for(int i = 0; i < n; i++)
    {
        for(int j = 0; j < n; j++)
        {
            mTable[i, j] = -1;
        }
    }
    LookupChain(mTable, sTable, pArray, 1, n - 1);

    List<int[,]> result = new List<int[,]>();

    result.Add(mTable);

    result.Add(sTable);

    return result;
}
```

This method is the first of two needed for the memorized version of the matrix chain algorithm. It takes in the same array of matrix dimensions as the previous version. Then it sets n to be the length of the array, and creates two $n \times n$ arrays where n is the length of the given pArray, and sets the mTable's entries to -1. Then it calls the lookupchain method. Once the lookupchain is done, the two tables are returned as elements of a list.

```

public static int LookupChain(int[,] mTable, int[,] sTable, int[] pArray, int i, int j)
{
    if(mTable[i, j] != -1)
    {
        return mTable[i, j];
    }
    if(i == j)
    {
        mTable[i, j] = 0;
    }
    else
    {
        for (int k = i; k < j; k++)
        {
            int q = LookupChain(mTable, sTable, pArray, i, k) +
                LookupChain(mTable, sTable, pArray, k + 1, j) +
                (pArray[i - 1] * pArray[k] * pArray[j]);
            if (q < mTable[i, j] || mTable[i, j] == -1)
            {
                mTable[i, j] = q;
                sTable[i, j] = k;
            }
        }
    }
    return mTable[i, j];
}

```

The base case for this method is if the value at the given index is not -1. If that is the case, the value at the index is returned. Otherwise, if we are on the diagonal, it is set to zero, and then we loop through recursively using solving each entry before using it to solve the next one.

Part 3

```

C:\Users\Gabriel\Desktop\WOU\05-2019_springTerm\CS361\CS361-S19-gibsgibs\labs\cs361...
Matrix Chain Order
M Table with p = { 30, 4, 8, 5, 10, 25, 15 }
+-----+
| 0 | 960 | 760 | 1560 | 4360 | 4660 |
+-----+
| 0 | 0 | 160 | 360 | 1360 | 2860 |
+-----+
| 0 | 0 | 0 | 400 | 2250 | 3725 |
+-----+
| 0 | 0 | 0 | 0 | 1250 | 3125 |
+-----+
| 0 | 0 | 0 | 0 | 0 | 3750 |
+-----+
| 0 | 0 | 0 | 0 | 0 | 0 |
+-----+

Matrix Chain Order
S Table with p = { 30, 4, 8, 5, 10, 25, 15 }
+-----+
| 0 | 1 | 1 | 1 | 1 | 1 |
+-----+
| 0 | 0 | 2 | 3 | 4 | 5 |
+-----+
| 0 | 0 | 0 | 3 | 3 | 3 |
+-----+
| 0 | 0 | 0 | 0 | 4 | 5 |
+-----+
| 0 | 0 | 0 | 0 | 0 | 5 |
+-----+
| 0 | 0 | 0 | 0 | 0 | 0 |
+-----+

Matrix Chain Order
Optimally Parenthesized Form
+-----+
| (A1((((A2A3)A4)A5)A6)) |
+-----+

```

Part 4

```

C:\Users\Gabriel\Desktop\WOU\05-2019_springTerm\CS361\CS361-S19-gibsgibs\labs\cs361...
Memoized Matrix Chain
M Table with p = { 30, 4, 8, 5, 10, 25, 15 }
+-----+
| 0    | 960 | 760 | 1560 | 4360 | 4660 |
+-----+
| -1   | 0    | 160 | 360  | 1360 | 2860 |
+-----+
| -1   | -1   | 0    | 400  | 2250 | 3725 |
+-----+
| -1   | -1   | -1   | 0    | 1250 | 3125 |
+-----+
| -1   | -1   | -1   | -1   | 0    | 3750 |
+-----+
| -1   | -1   | -1   | -1   | -1   | 0    |
+-----+

Memoized Matrix Chain
S Table with p = { 30, 4, 8, 5, 10, 25, 15 }
+-----+
| 0 | 1 | 1 | 1 | 1 | 1 |
+-----+
| 0 | 0 | 2 | 3 | 4 | 5 |
+-----+
| 0 | 0 | 0 | 3 | 3 | 3 |
+-----+
| 0 | 0 | 0 | 0 | 4 | 5 |
+-----+
| 0 | 0 | 0 | 0 | 0 | 5 |
+-----+
| 0 | 0 | 0 | 0 | 0 | 0 |
+-----+

Memoized Matrix Chain
Optimally Parenthesized Form
+-----+
| (A1((((A2A3)A4)A5)A6)) |
+-----+

```

Part 5

```

public static void BreadthFirstSearch_al(AdjacencyList adjacencyList, List<Vertex> vertices, Vertex startingVertex)
{
    for(int i = 0; i < vertices.Count; i++)
    {
        vertices.ElementAt(i).SetColor("WHITE");
        vertices.ElementAt(i).SetDistance(int.MaxValue);
        vertices.ElementAt(i).SetParent(null);
    }
    startingVertex.SetColor("GRAY");
    startingVertex.SetDistance(0);
    startingVertex.SetParent(null);

    Queue<Vertex> queue = new Queue<Vertex>();
    queue.Enqueue(startingVertex);

    while(queue.Count() != 0)
    {
        Vertex u = queue.Dequeue();

        int location = Convert.ToInt32(u.GetLabel()[0]) - 65;

        foreach (Vertex v in adjacencyList.adjacencyList[location])
        {
            if(v.GetColor() == "WHITE")
            {
                v.SetColor("GRAY");
                v.SetDistance(u.GetDistance() + 1);
                v.SetParent(u);
                queue.Enqueue(v);
            }
        }
        u.SetColor("BLACK");
    }
}

```

This method performs a breadth first search on an adjacency representation of a graph. It does so by looping through the list of vertices and setting the color to white, the distance to the maximum int value, and the parent vertex to null. Then it sets the starting vertex's color to white and its distance to 0. Then it enqueues it into a queue and then as long as the queue is not empty, it loops through and sets each non visited vertex's color to gray, its distance to the previous vertex's distance plus 1, and its parent to the previous vertex. Once all adjacent vertices to a vertex are checked, the vertex is colored black. Then the cycle continues until all the vertices are black.

Part 6

```

public static int[,] BreadthFirstSearch_am(int[,] adjacencyMatrix)
{
    int m = adjacencyMatrix.GetLength(0);
    int n = adjacencyMatrix.GetLength(1);

    int[,] result = new int[m, n];

    PriorityQueue<int> queue = new PriorityQueue<int>();
    PriorityQueue<int> vertices = new PriorityQueue<int>();

    int i = 0;

    queue.Enqueue(i);

    for(int j = 1; j < m; j++)
    {
        if(adjacencyMatrix[i, j] == 1)
        {
            result[i, j] = 1;
            result[j, i] = 1;
            queue.Enqueue(j);
            vertices.Enqueue(j);
        }
        vertices.Enqueue(i);
    }
    while(queue.Count() != 0)
    {
        i = queue.Dequeue();

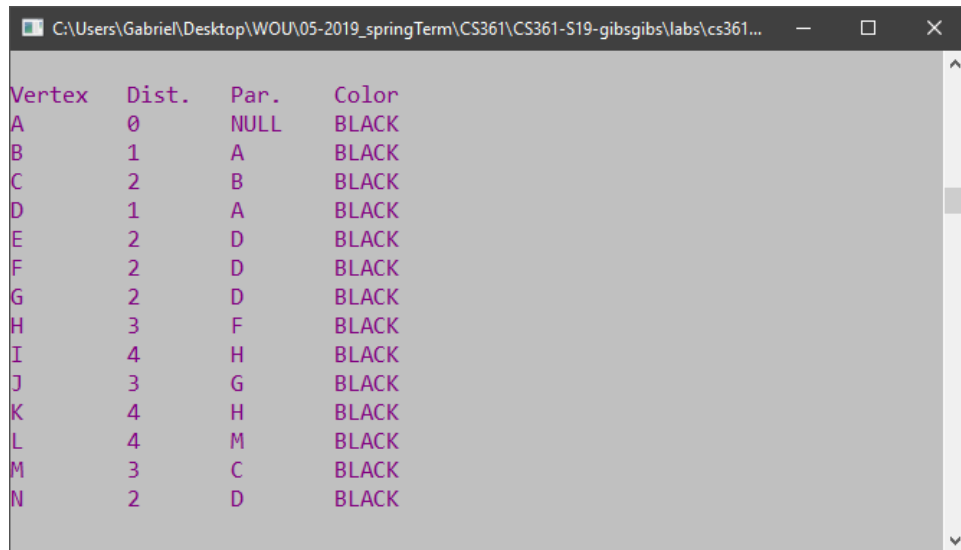
        for(int k = 0; k < m; k++)
        {
            if(adjacencyMatrix[i, k] == 1 && !vertices.Contains(k))
            {
                result[i, k] = 1;
                result[k, i] = 1;
                queue.Enqueue(k);
                vertices.Enqueue(k);
            }
        }
    }
    return result;
}

```

This method is very similar to the previous except instead of an adjacency list, this method takes in a matrix representation of a graph. It just checks if two vertices are incident on each other, and if they are then the resulting matrix has a 1 placed at the same spot, and the j index is enqueued into both the queue and the vertices queue. Then, as long as the queue is not empty, we dequeue the value in queue and loop through checking if the adjacency matrix at that dequeued value is 1 and if the vertices queue


does not contain k. If that is the case, the result matrix gains a new entry and the specified location, and the k value is enqueued into both queues. The result is then returned.

Part 7



Vertex	Dist.	Par.	Color
A	0	NULL	BLACK
B	1	A	BLACK
C	2	B	BLACK
D	1	A	BLACK
E	2	D	BLACK
F	2	D	BLACK
G	2	D	BLACK
H	3	F	BLACK
I	4	H	BLACK
J	3	G	BLACK
K	4	H	BLACK
L	4	M	BLACK
M	3	C	BLACK
N	2	D	BLACK

Part 8



Adjacency Matrix

0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	1	0
0	0	0	1	1	1	0	0	0	0	0	0	0	1
0	0	1	0	1	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0	1	0	0	0	0	0
0	0	0	0	1	1	0	1	0	1	0	0	0	0
0	0	0	0	0	0	1	0	1	0	0	0	0	0
0	0	0	0	0	1	0	1	0	1	0	0	1	0
0	0	0	0	0	0	1	0	1	0	1	0	0	0
0	0	0	0	0	0	0	0	0	1	0	1	0	0
0	1	0	0	0	0	0	0	0	1	0	0	1	0
0	1	1	0	0	0	0	0	1	0	0	1	0	0

C:\Users\Gabriel\Desktop\WOU\05-2019_springTerm\CS361\CS361-S19-gibsgibs\labs\cs361...

Breadth First Search

0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	1	1
0	0	0	1	1	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1	0	0	0	0	0
0	0	0	0	1	0	0	1	0	1	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0

References

I worked with Jacob Malmstadt on this project.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). Cambridge, MA: The MIT Press.