CS141, Fall 2021, Assignment #5 **Due: 11:59pm, Tue 11/30, 2021**
Name: Rishab Dudhia **Student ID: 862141444**

1 Walking in UCR (25pts)

Questions:

(1) When using a binary heap as the priority queue for Dijkstra's Algorithm, the worst case running time for the algorithm would be $O((n+m)\log(n))$ when there are n vertices and m edges. If the graph is dense, the m value will dominate having a factor of m operations of cost $\log(n)$ in the while loop to clear the priority queue. If the graph is sparse, the number of operations to clear the priority queue will be a factor of n instead of m (since there will be more vertices than edges) while the cost of the operations will still be $\log(n)$. The worst case running time for Bellman-Ford's Algorithm will be $O(mn)$. We can see this because each edge can be checked at most n times since each iteration of the while loop checks the edges of the current vertex then settles the current vertex and moves to the next vertex. Therefore, after all vertices have been checked, each edge could have been checked at most n times and therefore there are a maximum of $O(nm)$ checks for this algorithm.

(2) Dijkstra's Algorithm uses a greedy approach because the use of the priority queue implemented as a binary minimum heap allows us to choose the current vertex in the while loop as the root of the heap (vertex in queue with lowest tentative distance). Here, we are making the greedy choice of choosing to settle the vertex with the lowest tentative distance and then solve the subproblem without the current vertex by similarly choosing the vertex with the lowest tentative distance. Bellman-Ford's algorithm does not have a greedy approach since it iteratively checks each and every vertex with its neighbors and just remembers past checked edges which does not contain a greedy decision.

(3) Dijkstra's Algorithm:

Step	selected vertex	tentative distance								#of relaxations
		1	2	3	4	5	6	7	8	
Initial	-	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	0	$+\infty$	
Round 1	7	1	$+\infty$	$+\infty$	$+\infty$	$+\infty$	17	0	$+\infty$	2
Round 2	1	1	4	$+\infty$	$+\infty$	10	15	0	$+\infty$	3
Round 3	2	1	4	8	$+\infty$	9	15	0	$+\infty$	2
Round 4	3	1	4	8	10	9	15	0	$+\infty$	1
Round 5	5	1	4	8	10	9	15	0	$+\infty$	1
Round 6	4	1	4	8	10	9	15	0	17	2
Round 7	6	1	4	8	10	9	15	0	17	1

Note that even a relaxation is not successful, you should count it in the last column.

(4) There are m relaxations done by Dijkstra's Algorithm on a graph with n vertices and m edges. We know this because each edge will be relaxed exactly once. The current vertex in the while loop will have its neighbors checked (via the edges connecting to the current vertex) and then the current vertex will be marked as settled meaning that the vertex will not be checked again, by way the edges connecting to that vertex will also not be used to settle any other nodes. Once the algorithm finishes, all edges will be used exactly once and will also connect to settled vertices meaning that through $n-1$ rounds, there will be m relaxations done by Dijkstra's algorithm on a graph with n vertices and m edges. This will result in the shortest path from the starting vertex to all vertices in the graph.

(5) Bellman Ford's Algorithm uses more relaxations. We know this because the number of relaxations that will occur will be the number of edges times one less than the number of vertices in the graph,

$m(n-1)$ relaxations. We know this because each edge will get checked in the double for loop checking the neighbors in the $n-1$ hops from the starting vertex in which each iteration is trying to relax the current tentative distance with the current amount of hops (determined by the iteration number of the outer for loop). Therefore, a check, or relaxation, will occur m times per iteration for $n-1$ iterations in the algorithm. This is significantly more than Dijkstra's algorithm which uses only m relaxations.

2 Parallel Matrix Multiplication (25pts)

In our lecture 3, we learned a divide-and-conquer algorithm for matrix multiplication. Our task here is to parallelize this algorithm.

Questions:

1. We learned a Divide and Conquer Matrix Multiplication algorithm with the recurrence of

$$T(N) = 8 \cdot T(\frac{N}{2}) + \Theta(N^2)$$

In which we can set the problem up as such:

```
C11|C12          A11|A12        B11|B12
---+---    =     ---+---   x    ---+---
C21|C22          A21|A22        B21|B22
```

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

As we can see from above, to calculate the four C matrix values, we must, first, calculate the 8 values (terms in above equations) of the A and B submatrices. Therefore, in the recurrence of the algorithm shown above, there are 8 calls to the function, each creating N^2 space. From master theorem, we can say that the function has time complexity $\Theta(N^3)$ since $f(n)$ in our recurrence is of complexity $\Theta(N^2)$ which is equal to $O(N^{(\log_b a)-\epsilon})$ where $b = 2$, $a = 8$, and $\epsilon > 0$ (we choose $\epsilon = 1$). We know that for a function to be $\Theta(N^2)$, it must also be $O(N^2)$, so the Master Theorem can be applied.

2. To parallelize the algorithm, I will use the fork/join method learned in class. We know that when one value of C is calculated, the result will not affect the calculations of the other values of C. Therefore, we can parallelize the calculations of each value of C. Then, C_{11}, C_{12}, C_{21}, and C_{22} can all be calculated from the parallelly calculated submatrix values, as long as multiple processors are available. The pseudocode for the algorithm would look as:

```
matrix multMat(matrix A, matrix B) {
    create new empty matrix called C (size of A)
    if (A has 1 row) {
        C11 = A*B \\since only one element A and B are ints
    }
    else {
        fork:
            n1 = multMat(A11,B11)
            n2 = multMat(A12,B21)
            n3 = multMat(A11,B12)
```

```
            n4 = multMat(A12,B22)
            n5 = multMat(A21,B11)
            n6 = multMat(A22,B21)
            n7 = multMat(A21,B12)
            n8 = multMat(A22,B22)
        join
            C11 = n1 + n2
            C12 = n3 + n4
            C21 = n5 + n6
            C22 = n7 + n8
    }
    return C
}
```

Here we are creating a matrix C which will be returned and has the size of input matrix A (assuming same size as B). As long as there is more than one element in A, we fork the eight calculations to create up to 8 threads to parallelly calculate their respective values. Then the values are put into the new matrix created at the beginning of the function and the matrix is returned.

3. Recurrence for work for my algorithm:

$$W(n) = 8 \cdot W(\frac{n}{2}) + \Theta(n^2)$$

We get this recurrence for the work because the call to the function creates eight threads with each thread being passed matrices half the size of the matrices in the original thread. The definition of work means the time taken when only one processor is available, so the 8 created threads would have to run sequentially giving us this recurrence.
Recurrence for span for my algorithm:

$$S(n) = \Theta(n^2) + \max(S(\frac{n}{2}), S(\frac{n}{2}), ..., S(\frac{n}{2}), S(\frac{n}{2}))$$

$$S(n) = \Theta(n^2) + S(\frac{n}{2})$$

We know this is the recurrence for the span because the longest dependency chain will follow a path where in each thread along the path, the size of the matrix is cut in half.

4. Using the Master Theorem, we are able to solve the recurrences as follows (which case used of the master theorem):
Solved Work recurrence (1: $\epsilon = 1$):

$$W(n) = \Theta(n^3)$$

Solved Span recurrence (2: $k = 1$):

$$S(n) = \Theta(\log(n))$$

5. Strassen's Algorithm can definitely have the same idea applied to it. The algorithm would look almost identical to the pseudocode for part two, except in the fork/join segment, there would only be seven values calculated instead of 8. Instead of creating 8 threads, parallelizing Strassen's algorithm will therefore only net in 7 threads. The span will remain the same as the 8 way divide and conquer method since unfolding (recurring to the base case) the submatrices must take $\log(n)$ time. The work will decrease slightly from the 8 way divide and conquer method since the recurrence for work for Strassen's algorithm would be

$$W(n) = 7 \cdot W(\frac{n}{2}) + \Theta(n^2)$$

This would simplify to Strassen's algorithm having a total work of $W(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.8074})$.

3 (Bonus, 5pts + 2candy) Happy Days!