**CSE341T/CSE549T**

# Homework 4

*Assigned: 10/27/2014* *Due Date: 11/17/2014*

# 1 Introduction

You will begin this assignment by doing a simple analysis of a couple of algorithms, and apply maximal independent set algorithm you learned in class. In addition, you will implement the Floyd-Warshall Algorithm for solving the All Pairs Shortest Paths problem.

## Submission

This assignment requires that you submit both code and written answers.

Your written answers must be printed out and submitted in class on the day the assignment is due. The coding portion of the assignment is in your `svn` repository. Perform an `svn update` in the directory where you have checked out your repository and you will see the directory `hw4`. Inside this directory, you will see the following files in it:

- `StudentSolution.cpp`: **the only file you should modify for this assignment**
- `Apsp.cpp`: where the main All Pairs Shortest Path code lives. This is the code that calls your 4 methods.
- `cilktime.h`, `Timer.h`, `Timer.cpp`: files used to time and compare the speed of your 3 algorithms
- `Makefile`, `ApspSupport.cpp`: code you should be glad we wrote for you
- `input1`, `input2`, `input3`: sample input files of increasing sizes. the first line of the file indicates the size of the matrix. For example, if the first line is 5, a correctly formatted input file will have 5 *more* lines, each with 5 values in it, representing a 5x5 matrix. Remember, the edges are directed, so the matrix is not symmetric. Also, the values on the diagonal should all be 0, because a node has distance 0 to itself.

Your assignment is to complete the 4 methods in `StudentSolution.cpp`. This is the only file you are allowed to modify. This is the only file we will look at when we grade your assignment. Your code must compile cleanly. If you make changes to any other files, they will be discarded before we compile, run, and grade your homework. Make sure that the three methods maintain their original names and signatures. Your code will be automatically graded; if your method has the wrong name or signature, you will not receive credit for your work.

**Problem 4-1. All Pairs Shortest Paths** [50 points]

The All Pairs Shortest Paths (APSP) problem asks one to find the shortest path between all nodes in a weighted graph with directed edges and no negative cycles. In class, we learned two dynamic programming algorithms for computing the APSP problem, one using repeated matrix multiplication and the Floyd Warshall Algorithm. In this homework, you will implement both these algorithms. The functions you need to write are write in `StudentSolution.cpp` are the following:

- `void matrixMultSequential(vector<vector<weight_t>> &matrixA, vector<vector<weight_t>> &matrixB, vector<vector<weight_t>> &result)`
- `void matrixMultParallel(vector<vector<weight_t>> &matrixA, vector<vector<weight_t>> &matrixB, vector<vector<weight_t>> &result)`
- `bool apspMMParallel(vector<vector<weight_t>> &weights, vector<vector<weight_t>> &distances)`
- `bool floydWarshallParallel(vector<vector<weight_t>> &weights, vector<vector<weight_t>> &distances)`

where `weight_t` is type of the edge weight, defined to be an `int` in this case.

They have the following arguments:

- `matrixA, matrixB`: the matrices to multiply
- `result`: the result of multiplying the two matrices
- `weights`: matrix of edge weights stored as a vector of vectors. The value of entry `(2,4)` indicates the weight of the edge from node 2 to node 4. If there is no edge from vertex 2 to 4, then the cell contains inf.
- `distances`: matrix of distances between all pairs, stored as a vector of vectors. The value of entry `(2,4)` should indicate the distance of the shortest path from node 2 to node 4. If there is no path from 2 to 4, then the cell should contain inf (the maximum value one can represent with an `int`).

Finally, the APSP functions return `false` when a negative weight cycle is discovered.

**Compiling your code**. Compile your code by typing "`make Asps`" into the command line.

**Running your code**. Your code can be run with many possible arguments. Here are just a few examples:

| | |
|---|---|
| `./Asps -help` | shows all possible arguments for customizing your run |
| `./Asps -testMM 0` | runs on a randomly generated graph; will *not* explicitly test the matrix multiply functionality. |
| `./Asps -size 10 -sparsity .3` | runs on a randomly generated graph of 10 nodes with sparsity 0.3 |
| `./Asps -f input1` | runs with the input file `input1` |
| `./Asps -s 35 -size 1000` | runs on a randomly generated graph of 1000 nodes, generating weights using seed 35 |

**Testing Your Code.** Be sure to test your code for correctness with a variety of graphs to convince yourself that your implementations are correct.

Three input files of integers `input1-3` are provided for you in the `hw04` directory. They have sizes 5, 5, and 15, respectively.

Feel free to create new input files, or randomly generate sequences of different lengths. We will run your student solution against our reference solution and the solutions must match in order to get full credit. Also, the parallelism reported by Cilkview should make sense.

**(a) Serial Matrix Multiply Implementation** [5 points]

Implement a serial matrix multiply function called `matrixMultSequential()`. In this problem, you should use the operators that are suitable for your APSP implementation. That is, instead of the actual matrix multiply operators (sum, multiply) use the ones you would use as a subroutine for APSP (max, sum). *Watch out for integer overflow!* Since we use maximum possible integer value (inf) to represent nodes that don't contain path between each other, when you add two inf values together, you will overflow the weight and get a negative value in return.

This function is used mainly for debugging purpose. The driver code contain testing code that test your parallel implementation of the matrix multiplication against your serial implementation. This is only to help you debug. For implementing APSP using matrix multiplication, you should call the your parallel matrix multiply implementation, not the serial one.

**(b) Parallel Matrix Multiply Implementation** [10 points]

Implement a parallel matrix multiply function called `matrixMultParallel()`. The driver code contains testing code to ensure that this version returns the same answer as your serial implementation.

**(c) Implement All-Pair's Shortest Paths Using Matrix Multiplication** [10 points]

Use your matrix multiplication algorithm to implement the all-pairs shortest paths algorithm using matrix multiplication we learned in class. State the work and span of your algorithm in the writeup.

**(d) Floyd Warshall Implementation** [20 points]

Implement the Floyd Warshall Algorithm we learned in class. It should be a parallel implementation. Briefly describe the algorithm and state the work and span in your writeup.

**(e) Measure the work, span, and parallelism** [5 points]

Include in your writeup the values that you see for work, span, and parallelism (not the burdened one) reported by `cilkview` under the "Cilk Parallel Region(s) Statistics" section for all three parallel functions above. The driver code you received contains calls to `cilkview` to measure the work and span for each of these regions. You have to comment out calls for one region at a time and rerun to see the reporting for the given region. If you comment out all calls to `cilkview`, `cilkview` will report the work and span averaged across all three regions, which is not what you want. Report the values for a moderately sized input.

**Problem 4-2. Graph Coloring** [10 points]

Let a graph $G = (V, E)$ be given. If $C$ is a set of colors, a *coloring function* for $G$, $c_G : V \to C$, maps every vertex to a color such that adjacent vertices do not have the same color. That is to say, for all $uv \in E$, $c_G(u) \neq c_G(v)$. A graph is said to be $k$*-colorable* if there exists a coloring function for that graph whose range has size at most $k$. For example, every graph is trivially $|V|$-colorable by mapping each vertex to its own color; a star graph is 2-colorable by picking one color for the fringe vertices and another for the central vertex. The graph coloring problem is to produce a coloring function that uses the smallest possible number of colors. Producing a minimum coloring, i.e. one that uses the fewest possible number of colors, is known to be **NP**-complete in general, so we'll be looking for an approximation.

Let $\Delta$ denote the maximum degree in a graph. Develop an algorithm that reduces coloring to MIS to produce a $(\Delta + 1)$-coloring. Prove that your algorithm will in fact generate a $\Delta + 1$-coloring. Also analyze the running time.

**Problem 4-3. Cycle Graphs** [25 points]

In class, we saw a graph contraction algorithm for the cycle graphs. It flips a coin for each edge and selects an edge to contract if it gets a heads and its two neighbors both come up tails. In class, we analyzed the variant when the probability of heads is $0.5$. Now you will analyze some variants of that algorithm. For both of these, on an $n$-vertex cycle $C_n$, compute the probability that an edge $e$ is contracted. In expectation, how many edges are contracted in terms of $n$? Show your work.

**(a)** [10 points]

We use the same algorithm as in class, but a coin comes up heads with probability $p$ (instead of $0.5$). What is the probability of contraction of each edge? What is the value of $p$ to get the maximum probability of contraction?

**(b)** [15 points]

Now, we'll analyze a slight variant of the algorithm: the modified algorithm chooses a random number $x_e \in_R [0, 1]$ for each edge $e$ independently—and it selects an edge to contract if that edge gets a larger number than those of its two neighbors.

**Problem 4-4. Currency Exchange** [15 points]

**(a)** [10 points]

You are planning to go to Japan, and want to buy japanese yen. However, you want to get the maximum number of yen for your dollar. Ah! Your knowledge of algorithms is finally going to come in handy. More formally, you want to solve the following problem: Given the a set currencies, a set of exchange rates between them, and a source currency $s$, find for each other currency $v$ the best sequence of exchanges to get from $s$ to $v$.

**(b)** [5 points]

Now that you are familiar with the currency market, you decide to get rich by trying your hand at *arbitrage*. You want to see if there is a way to game the market so that you start with $x$ amount of a currency $s$, and after a series of exchanges, you end up with $y$ amount of $s$ where $y > x$. Again, given a set of currencies and a set of exchange rates, give an algorithm to find out if there is a way to commit arbitrage.