Here are some candidate projects for Option#3

(hands-on implementation/prototyping of certain graph/network algorithms)

Topic1: Isomorphism checking on tree graphs

As input, your code should accept graphs of up to 1000 vertices. Graph representation should be a standard one, such as incidence matrix or adjacency matrix. It cannot be a tree-based representations, such as a BST or a heap. The goal is to take two (undirected) graphs at a time, determine if both graphs are trees or not, and if they are, test if the two trees are isomorphic. (I have in mind the regular graph isomorphism here, not rooted tree or ordered tree isomorphism.) So, given G1 and G2 as input, the output should be: either no isomorphism testing is done, because G1 or G2 (or both) are not trees, or else yes, both G1 and G2 are trees, and yes they are / no they are not isomorphic. (In other words, if at least one of them is not a tree, you don't need to run the isomorphism test).

So, your solution should have two parts: 1) given a standard matrix representation of the two graphs, you first need to determine if both of them are trees or not; and 2) once you have determined that both graphs are trees, implement an efficient isomorphism test. Note, there are tree isomorphism algorithms that are very efficient (linear time in the # of vertices); this does not hold for general undirected graphs, but it is the case for trees. You should test your initial solution on small graphs, such as of size n=10, where you can readily verify by hand whether the graphs are trees and whether the two trees are isomorphic. But, your implementation should run efficiently, and in particular take at most a few seconds (on a standard laptop) even for n=1000.

Topic2: Distributed Graph Partitioning into (Small) Cliques

This project is based on implementing a variant of a local algorithm for partitioning a graph into non-overlapping (maximal) cliques, based on my own work in grad school 10+ years back. (One benefit of this project is, if you do a good job it will readily be extensible into a research paper with me -- however, insofar as CS580 class is concerned, this is just one of several candidate topics and will be treated exactly the same way as all others from the course grade standpoint.)

This algorithm has an awkward acronym -- MCDCF, and links to the original papers about it are listed as Topic3 in Option1 (plus you can find more easily w/ Google Scholar). I do not expect you to have a distributed implementation; but the basic mechanism of identifying cliques locally by comparing lists of neighbors among the vertices has to be followed. I am very open to exploring different tie-breaking mechanisms when a group of vertices has not reached an agreement / identified a clique (to make sense of this, you will have to read at least one of my papers), as well as exploring other alternatives in the details of the mechanism, as long as the core idea of how cliques are found follows MCDCF. Obviously, if someone picks this for his or her project, I can explain MCDCF algorithm in detail since, well, I was the one who originally designed it back in 2004.

So, the input here is an undirected graph (say, of size up to 1000), and the output is the list of tuples or sets of vertices, so that i) each tuple of nodes corresponds to a clique in the original graph, ii) no two tuples have the same element (= each vertex belongs to exactly one tuple), and iii) the union of all tuples covers the entire vertex set of the orig. graph (i.e., each vertex is in some tuple/clique). Notice that, for a typical graph, it is to be expected that there will be a lot of 'trivial' cliques (such as, single edges corresponding to 2-tuples of adjacent vertices, and even single nodes corresponding to trivial "1-cliques").

Topic3: A heuristic algorithm for Traveling Salesman Problem (TSP)

Implement Algorithm 6.4.2. in the Gross & Yellen. (See exercise 6.4.20 in the textbook.)

INPUT: a complete weighted graph with n vertices, where n is between 4 and 20 (reject sizes less than 4 or bigger than 20). Weighted graph is given as an nxn adjacency matrix. Assume all weights a[i,j] for i != j to be positive integers (return 'Error: graph is not complete' if any adjacency matrix entries off the main diagonal are zero or negative).

OUTPUT: a closed Hamiltonian path (i.e., cycle that returns to the starting vertex) and its total cost (a positive integer equaling the sum of the weights of all edges in the Hamiltonian path that your implementation has found).

Test your implementation on graphs in exercises 6.4.2 -- 6.4.5 in the book. For graphs 6.4.2. and 6.4.3, find the optimal cost of a Hamiltonian cycle by exhaustively trying out all possibilities. Did your implementation of Algo. 6.4.2 find the actual optimal cycles for these two graphs? If not, how close is the solution found by Algo 6.4.2. as compared with the optimal cost? Is the found value within the multiplicative factor of 2 from the optimal cost? If it is not, does that violate Theorem 6.4.4. in the book (and why / why not?)

Topic4: The N chess queens problem

The n-queens problem is concerned with the placement of N (chess) queens on an NxN chessboard such that no two queens can attack each other. Two queens attack each other if they are in the same row, column, or diagonal of the chessboard. Your task is to design a program that finds solutions to the N queens problem using *backtracking* strategy, for the specified range of values of N.

INPUT: A positive integer N, 4 <= N <= 20 (if one inputs an integer <4 or >20, your program should warn that value is out of range for which the N queen problem is solved; and of course, any non-integer input value should be rejected with warning that the input has to be an integer in specified range)

OUTPUT: Solutions to the N-queens problem. When the # of solutions (for the given N) is <=10, all solutions should be found. For those values of N where there are more than 10 solutions, the algorithm should stop after having found 10 solutions, and output those 10 solutions. A solution is the list of positions of N non-attacking queens on the NxN chessboard. Solutions can be listed in the output either graphically, or using extended chess notation (so, letters a,b,c... to enumerate NxN board's rows, and integers 1,2,3 to enumerate the board's columns). For instance, for N=4, solution in Fig. 3.17 in Joyner et al. can be represented as the tuple or array [a2, b4, c1, d3].

See problem 3.19 (and read Chapter 3) in the supplementary reading for this course (eBook by D. Joyner et al.)