

CSE 830: Fall 2009 Project

Due: Tuesday, November 24th 2009, 10:20am

The real reason we study algorithms is to find fast ways for solving problems. Analysis gives us a way of seeing how well we are doing, but it washes away questions about constants, difficulty of implementation, and performance in practice. For this assignment, you will be given an algorithmic problem requiring some form of combinatorial search, and your goal is to design and implement as fast a solution as possible. This is intended to be a competition - may the fastest program win! Modest prizes will be given for the fastest and most clever entries.

The Problem:

Each campus of MegaCorp has a collection of buildings; some campuses are small while others are sprawling. Many of these buildings have their electrical systems directly connected to each other. MegaCorp wants to be prepared in the case of a blackout, so they plan to buy generators for some of the buildings. You are given a graph on which each vertex represents a building, and each edge connects a pair of buildings that are wired together. You are hired to write an algorithm that will find the minimum set of buildings to put generators in such that every building either has a generator or else directly shares wiring with another building containing a generator. Since each generator is expensive, it is important to find the exact minimum number of generators to be bought to meet these criteria.

The company represents this problem as a graph where the n buildings are the vertices, and the m pairs of buildings that share wiring are the edges. Your goal is to find the minimum set of vertices such that each vertex is either in the set or directly connected by a single edge to a vertex in the set.

This problem is NP-complete, meaning that it is exceedingly unlikely that you will be able to find an algorithm with polynomial worst-case running time (you will prove it NP-Complete on homework #5). However, since the goal of the problem is to find a subset of vertices in the graph, a branching program that iterates through all 2^n possible subsets of vertices and tests whether all of the vertices not in the set are directly connected to one in the set is an easy $O(n m 2^n)$ algorithm. But the goal of this assignment is to find as practically good an algorithm as possible.

Input and Output:

There are a variety of data files available in the directory `~cse830/Public/project/`. Each graph is in a format such that the first line gives the number of vertices (n) and the second line is the number of edges (m). The next m lines each contain a pair of numbers on them indicating the two vertices that are connected by each edge. Vertices are numbered from 0 to $n-1$. Please keep an eye on this directory because I will add extra sample files based on questions that I receive.

In a combinatorically explosive problem such as this, adding one to the problem size can double the running time, so test on the smaller files first. Do not be afraid to create your own test files to help debug your program.

Your program must take a single argument of the filename containing the graph to solve. You may assume that this file contains a simple graph. Your program must output the number of vertices in the solution set, and list the specific subset of vertices the solution includes. Don't worry if your solution is not unique as long as it is correct and takes the minimum number of possible vertices.

Implementation:

You will be graded on how fast and clever your program is, not on style. Incorrect programs will receive no credit. A program will be deemed incorrect if it does not find a subset corresponding to a minimum legal set of vertices for multiple examples.

You may use any programming language if you have a preference, but if you don't please use C++ for uniformity/efficiency (and my ability to help you debug!). The programs must be able to run under UNIX on the CSE cluster. I will be testing all of the programs on the same machine, using the same examples for uniformity, probably on `adriatic.cse.msu.edu`.

Writing efficient programs is an iterative process. Build your first solution so you can throw it away, and start it early enough to go through several iterations, especially if you feel your programming background is weak. Use a simple backtracking approach first; don't get fancy until you have something that works. It is possible to get a program working in about 100 lines.

Do not forget to use the code optimizer on your compiler when you make the final run, to get a free 50% or so speedup. The system profiler tool, `gprof`, will help you tune your program.

Grading:

You will be graded on how fast and clever your program is, not on style. Incorrect programs will receive no credit. A program will be deemed incorrect if it does not find a subset corresponding to a minimum legal set of vertices for multiple examples. As a rough guide, if you can consistently handle graphs with ~18 vertices, you'll get a 70, ~25 vertices will get you an 80, ~35 will get you a 90, and ~45 will get you 100. If you can handle graphs with 55 vertices, you'll get an extra credit, and if you make a significant effort and get to 70 vertices, I'll drop your lowest homework grade. A (compiled) example of such a program can be found in `~cse830/Public/project/graph.dom`

Other details:

Everyone *must* do this question individually. Just this once, you are not allowed to work with your partners. The idea is to think about the problem from scratch and on your own. If you do not completely understand the problem definition, you don't have the slightest chance of producing a working program. Don't be afraid to ask for clarification or explanation!

You are to turn in a listing of your program, along with a brief description of your algorithm and any interesting optimizations, sample runs, and the time it takes on sample data files. Make sure to include the largest test files your program could handle in one minute or less of wall clock time.

What kind of approaches might you consider? Instead of testing every subset of vertices, you should be able to develop a backtracking algorithm that prunes partial solutions (i.e., when you have excluded a vertex and all its neighbors) and bounding (i.e., when there is no way to finish building a smaller than the best one found so far). Can you order the vertices so that the search is likely to proceed more quickly? Starting off with a good approximate solution may help achieve faster cutoffs. There is plenty of room for cleverness in selecting data structures to minimize the time needed to test if a given collection of vertices forms solution.

Good luck!

