CS 6150 Projects

September 13, 2011

1 Rules

Please form a group of at most three people, and choose ONE of the projects listed in the following sections, Here are some general guidelines.

Submitting your project It doesn't matter which language you choose to implement your code in, although choosing something extremely obscure will make it difficult to test. All that matters is that your code can be run on the CADE system, so that we can evaluate it. Specifically, this means that when you turn in your project, you'll be turning in three things.

1. The actual code in a subdirectory
2. A text file containing a single line command that can be used to run the code.
3. (a week later) your report on the project (see below)

I/O issues Your code will be expected to read its data from standard input and send any necessary output to standard out. Inputs will be provided in a format specific to each problem: see the problem statements for more details.

Machine Tricks The goal of these projects is to explore nontrivial algorithmic ideas for hard problems. While methods that exploit machine idiosyncracies are definitely useful, they can miss the point of focusing on algorithmic innovation (which scales much more effectively than machine speedup). Secondly, for each of the problems I've listed, a well designed implementation will actually be useful to many people out in the field, and so portability is key. To that end, you're allowed to use some basic multicore methods if they're provided through a clean API, but for the most part I'd like to see a focus on (sequential) algorithm design and implementation. If this is unclear, come and talk to me.

Timeline No matter which project you choose, this is what I expect:

By September 19 Tell me which project you're going to choose, and who your group is (no more than three people). You're welcome to pick a group name as well: we'll use it on leaderboards. Please send this information to me by email. The subject line should be ``Algorithms Project: <choice ID>'' and the content should be a list of names, CADE userids and email addresses, one triple per line. The ``choice ID'' is `1,2` or `3` for the three projects. We will assume that the first person on the list will be the one whose user ID is used for the `handin` submissions later (see below).

By September 30 An initial two-page writeup that outlines your initial implementation plan. This will also be a good checkpoint to make sure you have any testing apparatus you need (and the test cases that we'll be providing).

By November 1 An intermediate summary (4 pages) that outlines how far you've succeeded with the initial implementation plan, what results you have, and what new strategies you're planning to implement.

By December 1 A finalized implementation. After this day, we will freeze code submission. You will now start writing your final report

By December 7 Final report on your project. See below for more details.

With the exception of the original email, all submissions will be made via `handin`. Please choose a designated person and list them first on the list of participants. We will use this person's CADE userid as the identifier for the project.

Complying with deadlines is important. Doing so will help keep you on track with the project. As an additional incentive, 10% of the project points will be allocated for timely submissions. If you make all deadlines on time, you will be awarded full points. If not, each deadline you miss will lose you 1/5 of the points. There are no extensions, and no late policy for these intermediate deadlines.

Report Your final project report must contain the following elements:

- A summary of the problem with whatever motivation you can find for studying it.
- A detailed accounting of prior art on the problem, with a focus on the main algorithmic ideas and tools used.
- A detailed description of your implementation: what algorithm(s) did you use, and how did you combine things ? What is innovative (if at all) about what you did ?
- Experiments that demonstrate the efficacy of your method. In all cases, you'll be expected to compare to at least one other approach that isn't just a strawman. Your experiments should make a clear argument for the efficacy of your implementation, by comparing to prior work and by indicating the absolute runtime-size-of-input tradeoffs you're able to achieve.

I expect that your final project report will be in the neighborhood of 10 pages of 11pt,single spaced, single column text (including plots and figures, but excluding bibliography). The project will be graded on the following elements:

- Code performance (as described in each project)
- Scholarship. This can be broken down into
  - Clarity of writing
  - Ideas used in the code
  - Experimental design
  - Exposition of prior and related work.

The two main elements are roughly equally weighted, so I'd encourage you to maintain a working document as you go along.

1. Treewidth

Treewidth is a graph-theoretic concept that is used to generalize the idea of being ``treelike''. It is an integer function of a graph G (usually denoted tw(G)) such that for any tree T, tw(T) = 1, and as the treewidth increases, the graph becomes less and less like a tree.

To define treewidth, we will first define a *tree decomposition*.

Definition 1.1. *Let $G = (V, E)$ be an undirected graph on $|V| = n$ vertices and $|E| = m$ edges. A* tree decomposition *of G is a pair (X, T) where $X = X_1, \ldots X_n$ is a collection of subsets of V and T is a tree whose nodes are the X_i, satisfying the following properties:*

- *The union of all X_i is V (each vertex of V occurs in at least one tree node)*
- *Each edge $\{a, b\} \in E$ occurs in some X_i (all edges are modelled)*
- *If both X_i and X_j contain a vertex $v \in V$, then all nodes on the (unique) path between X_i and X_j in T must also contain v. Equivalently, if X_i, X_j, X_k are nodes such that X_j lies on the path from X_i to X_k, then $X_i \cap X_k \subseteq X_j$.*

The *width* of a tree decomposition (X, T) of G is one less than the maximum size of an X_i. In other words, the width of (X, T) is $\max_i |X_i| - 1$. The *treewidth* of a graph G is the minimum width of a tree decomposition of G.

The wikipedia article on treewidth[8] provides the basic definitions and some examples. For a more in-depth review of the notion of treewidth, see Bodlaender[2].

Goal. Computing a tree decomposition of a graph is a very important problem. As we've seen in class, there are many problems that are NP-hard on general graphs but are easy to solve on trees. A simple extension of the dynamic programming framework for trees works for graphs of bounded treewidth (because you can essentially run a dynamic program on the tree T) and typically takes time that's polynomial in n, but is exponential in the treewidth. One classic example of this is exact inference in graphical models, where tree decompositions are used to make the inference problem more tractable.

However, computing the treewidth of a graph is NP-hard. So unless you plan to prove $\mathsf{P} = \mathsf{NP}$, you're likely to need an algorithm that takes exponential time in the worst case. It's not entirely obvious what even a brute force algorithm would do, because you have to both enumerate over all trees, and then enumerate over ways of assigning vertex subsets to trees.

Problem 1.1. *Design an algorithm to compute the treewidth of a graph, and implement it.*

Please see Appendix A for details on the input file format your program can expect. As output, you will produce both the treewidth, and a tree decomposition that certifies it. Specifically, your output should consist of:

1. a single line with the treewidth computed, followed by
2. A tree (represented in the graph format described in Appendix A), followed by
3. A sequence of lines of the form ``$v x_1 x_2 x_3 \ldots$'', where v is the index of a tree vertex and the x_i are indices of graph vertices

For example, suppose G was the tree $\{a, b\}, \{a, c\}, \{b, d\}, \{b, e\}$. Then in the tree decomposition there is one node for each edge of G. The resulting output file will look like

```
# treewidth
1
# tree decomposition
## number of nodes (number of edges is always one less)
```

```
4 3
# adjacency list of tree
1 2 3
2 1 4
3 1
4 2
# graph nodes assigned to tree nodes
1 a b
2 b d
3 a c
4 b e
```

Evaluation. The goal is to design an exact algorithm, and so a high value will be placed on correctness. In order to check correctness of your approach, we will test your method on large graphs of known treewidth (we will provide you with a generator to test your own code). In addition, we will look at performance. As a baseline, we will use one prior effort (also a masters project) to compute the treewidth of a graph, located at `http://www.treewidth.com`.

We expect that with an appropriate choice of programming language and the right heuristics, you should be able to come close or match the performance of this method. If you're consistently able to beat it by an order of magnitude, you are guaranteed an A grade on the project. More importantly, if you're able to beat it significantly, you'll have enough material to publish a paper on the topic.

Correctly computing the treewidth with plausible but not comparable performance will guarantee at least a B grade and maybe higher, depending on the actual performance. Being at the top of the leaderboard relative to other students will earn you bonus points that will only increase your grade. Note that beating the performance of this code is not necessary to get an A, if you perform well relative to other students.

2. Primality Testing

Primes are important. At the very least, you need them for RSA-style cryptography, More importantly, you need them for all kinds of applications of hashing (as we shall see when we talk about hashing in class). It's trivial to verify that a number n is NOT prime: produce a factor of it. It's a little trickier to verify that a number IS prime: this was the famous result of Vaughan Pratt in 1975[6] showing that a short (polynomial-size) certificate exists to convince you that a number is prime (i.e PRIMES is in NP).

But what about algorithms to determine whether a number is prime (i.e is PRIMES in P) ? This is probably one of the oldest problems in mathematics: indeed, one of the first algorithms for checking primality is the Sieve of Eratosthenes, dating back to 240 BC. But it took till 2002 for this question to be answered, when Agrawal, Kayal and Saxena[1] showed a polynomial time deterministic algorithm for determining whether a given number is prime.

Prior to the AKS result, primality testing algorithms did exist. There were a number of randomized procedures (that could be made deterministic if the Extended Riemann Hypothesis is true). These were Monte Carlo algorithms, in that they ran in polynomial time but had a small probability of error. There was also a Las Vegas algorithm that runs in *expected* polynomial time but is correct on all inputs. Again, Wikipedia has a good summary of the state of the art[7], and the introduction to the AKS paper has even more information.

Problem 1.2. *Implement an algorithm that can test for primality as quickly as possible.*

Your input will be a single number in ASCII format. Your output will be the value 0 or 1 (false or true). You will also produce a *certificate of primality*[6]. This certificate is a list of numbers, each on one line after outputting the answer. If in fact the number is not prime, then you will output two factors for the number, each on one line.

Evaluation. There are many algorithms out for testing primality, and the problem is complex, so I don't expect that you'd come up with a new method (although if you do, instant A!). What I'm suspecting will work best is a creative combination of methods, built off of a careful study of the methods themselves and understanding (through reading the literature as well as experimentation) which algorithms work better under what situations. I'm particularly curious to see if the AKS algorithm can be implemented efficiently.

There is a 2003 report by Crandall and Papadopoulos on implementations of AKS-like methods[3]. While this paper is now 8 years old, it does provide a benchmark for what performance you should be able to achieve. Crandall and Pomerance also have a book that reviews much of the prime number literature circa 2005[4]. As with the previous project suggestion, relative performance will play an important role in judging how well you do.

3. Vertex Cover

Definition 1.2. *Let $G = (V, E)$ be an undirected graph. A* vertex cover $S \subseteq V$ *is a set of vertices incident on every edge in E. Specifically, for all $e = \{u, v\} \in E, \{u, v\} \cap S \neq \emptyset$.*

VERTEX COVER is the problem of finding a minimum cardinality vertex cover of G. This problem is one of the original problems[1] shown to be NP-hard in Karp's breakthrough paper[5]. It is fairly easy to get a vertex cover that's at most twice as large as the true answer (pick an edge, take its endpoints, delete them from the graph along with their incident edges, and repeat). VERTEX COVER can be formulated as an integer linear program, and can also be solved in time $2^k n^{O(1)}$, where k is the size of the optimal cover.

Problem 1.3. *Implement an algorithm that solves VERTEX COVER exactly as efficiently as possible.*

You will assume that the input is presented as described in Appendix A. Your output will be the size of the vvertex cover, followed by a list of vertex IDs, one on each line.

Evaluation. As with the other problems, you will be evaluated on both the correctness and efficiency of your solution. VERTEX COVER is one of the more tractable NP-hard problems, so be creative ! There are a few heuristics suggested in the wikipedia entry, and you can often exploit special structure in the graph to do better. For example, if you had a treewidth decomposition, you could implement a dynamic program for VERTEX COVER that would run in time polynomial in n and exponential in the treewidth. In general, it's like that one of two approaches will yield the best results: either frame the problem as an integer linear program and call a solver, or use any of the parametrized methods that you'll find with some digging around.

We will provide you with test cases to evaluate your algorithm on: this is how we will check correctness of your implementation.

A Graph format

An undirected graph $G = (V, E)$ is represented in the following manner:

```
# Number of vertices and edges
n m
# An adjacency list consisting of a vertex ID and the list of vertices its adjacent to
v v1 v2 v3
u v4 v5 v6
# .. and so on
```

Comments are preceded by # and can be ignored. For example, consider the complete graph on four vertices a, b, c, d. It will be represented as

```
4 6
a b c d
b a c d
c a b d
d a b c
```

[1]It was number five, actually.

References

[1] Agrawal, M., Kayal, N., and Saxena, N. Primes is in p. *Annals of Mathematics* (2004), 781--793.

[2] Bodlaender, H. A tourist guide through treewidth. *Technical report RUU-CS 92* (1993).

[3] Crandall, R., and Papadopoulos, J. On the implementation of aks-class primality tests. `http://images.apple.com/acg/pdf/aks3.pdf`.

[4] Crandall, R., and Pomerance, C. *Prime numbers: a computational perspective*. Springer Verlag, 2005.

[5] Karp, R. Reducibility among combinational problems. *Complexity of Computer Computations* (1972), 85--104.

[6] Pratt, V. Every prime has a succinct certificate. *SIAM Journal on Computing 4* (1975), 214.

[7] Wikipedia. Primality test. `http://en.wikipedia.org/wiki/Primality_test`.

[8] Wikipedia. Tree decomposition. `http://en.wikipedia.org/wiki/Tree_decomposition`.