CS5234: Combinatorial and Graph Algorithms

MiniProject Ideas

Mini-Project Instructions and Ideas

1 Overview

The last three "problem sets" consist of a mini-project. (I refer to it as a mini-project in that it should involve more exploration than a standard problem set, but it is more constrained than a large project.)

The goal of the mini-project is to focus on a specific area and explore a specific problem in more depth. In general, your mini-project should involve algorithms related to this class, e.g., streaming algorithms, sublinear time algorithms, cache-efficient algorithms, or parallel/distributed algorithms. I expect most projects will have an implementation/experimental component, though theory/algorithm-oriented projects are certainly appreciated as well.

A good project will likely go beyond simply implementing one algorithm from this class. Either it will examine algorithms that we did not study in this class, or it will compare several algorithms in a more thorough fashion, or it will involve considering several different implementation optimizations and exploring which implementation techniques work best. (Notice that some algorithms from this class are more complicated than other, and some involve more optimization work to implement efficiently. This will be taken into account.) The goal of an implementation is to learn something about how the algorithms in this class actually work in practice, and any project that achieves that is headed in the right direction.

There are several different ways to approach your mini-project:

- *Problem driven*: Choose a problem that interests you. Look at how that problem can be solved using techniques from this class. You may consider existing algorithms, optimized versions of existing algorithms, or develop new algorithms. Either run experiments comparing different approaches, or prove new theorems showing that your approach is efficient. The goal should be to understand how your problem can be solved efficiently as the related data gets very large.

- *Data driven*: Choose an interesting data set (that is large). Use algorithms from this class to understand and analyze the data. See how efficiently you can drive interesting and useful conclusions.

- *Algorithm driven*: Find 3-4 different algorithms that solve the same problem using different techniques. Implement and compare these different approaches. See if you can optimize any of these algorithms to perform better. See if you can determine what aspects are important for a good implementation.

Regardless of which approach you take, it is a good idea to:

- *Think about the problem:* There should be some problem you are trying to solve, or something you are trying to better understand.

- *Use real data:* If you are doing an experimental project, try to use real data (rather than artifical, randomly generated data). See below for some suggestions of interesting data sets.
- *Implementation:* Translating an algorithm from theoretical pseudocode to a real program requires making several decisions. Think about (and discuss in your write-up) how to get the most efficient implementation.
- *Optimization:* Many of the algorithms discussed in this class can be optimized to run much faster in the common case. Think about how to get better performance for your problem. (Remember, we have focused mostly on asymptotic analysis, but if you can improve the running time by a factor of two, that is a huge improvement!)

2 Organization Details

You may work on the mini-project in teams of two. By Thursday October 20, I would like you to submit (via the form posted on the website): (i) the members of your team, (ii) the topic you have chosen, and (iii) the questions you are hoping to answer.

Your results should be submitted as a short self-contained report. The final submission should consist, roughly, of four sections:

1. Overview/background: Describe the problem and the general area. Summarize what is known. This section should be similar to the introduction and related work section of a research paper, providing the reader with the necessary background. This will likely require you to do some background reading. Depending on your mini-project, it might be relatively short, or might be 1-2 pages.
2. Algorithms and theory: Describe any algorithms that you will use or have devised. Give any proofs that you may need or that you develop.
3. Implementation and experiments: Describe any implementations. Present any data that you have collected and analyze it. (Your analysis of the experiments/data should explain the implications of what you have done.)
4. Conclusion: State any conclusions, and describe any hypotheses/conjectures that have arisen from your analysis/experiments. Give a few interesting questions that have come up during your exploration. Describe some possible further applications of these techniques.

Each section should read as a well-written stand-alone section, beginning with an introductory paragraph and continuing with a sequential development of subsections. The goal is that the report should be readable by someone who has not taken the class. Good writing is encouraged throughout.

Timeline:

Deadline	**Milestone**
Oct. 20	Project choice and team formation
Oct. 31	Interim report draft due
Nov. 10	Presentation (in class)
Nov. 11	Final report due

The interim report is designed to describe your progress so far. It should include the overview and/or background part of your final report, along with a discussion of ongoing progress/issues.

In the last week of class, your group will give a short presentation on your mini-project. Presentations will likely be limited to 10 minutes for each mini-project.

A note on proper citation: Be sure, throughout, to properly indicate what is already known and what is derived from existing sources. You are expected to properly cite where any algorithm or argument or proof derives from. If your algorithm is similar to (or derived from) an existing one, then you might explain that and cite it. If your proof was inspired by (or consists of a modified version of) an existing proof, then you must explain that and cite it.

3 Data Sources

There are a variety of great sources of data that you can use to test your ideas or motivate your miniproject. Here are a few places to look:

- Stanford Large Network Dataset Collection: `https://snap.stanford.edu/data/`
  The SNAP project has made available a variety of data sets, particularly focused on social networks. For example, you can find a LiveJournal dataset with almost five million nodes, or a Friendster dataset with 65 million nodes! You can also find web graphs, wikipedia data, Amazon reviews, movie reviews, etc.

- Other social networks: There are a variety of other sources for social networks as well! For example: http://law.di.unimi.it/datasets.php (has more social networks, Facebook, and internet data).

- Taxi data: there are a variety of sources for taxi data. For example, there is significant data on New York taxi trips (`http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml`, `http://www.andresmh.com/nyctaxitrips/`, `https://github.com/fivethirtyeight/uber-tlc-foil-response`). There is data on San Francisco taxi traces (`http://crawdad.org/epfl/mobility/20090224/`) which includes mobility as well. And you can even get real-time streams (updated every 30 seconds) of the location of available taxis in Singapore (`https://data.gov.sg/dataset/taxi-availability`)!

- Airline data: You can find a lot of data on airlines at http://www.transtats.bts.gov/. This includes information on airline departures and arrivals (and when they are on-time or late), e.g., https://apps.bts.gov/xml/ontimesummarystatistics/src/index.xml.

- Wikipedia data: You can view wikipedia as a graph, where entries are nodes and links between Wikipedia pages are edges. You can access Wikipedia directly (via its typical interface), or you can get a summary of the graph structure (https://dumps.wikimedia.org/), or you can use the dataset others have extracted (http://haselgrove.id.au/wikipedia.htm).

- Project Gutenberg: This "dataset" includes a very large number of books.

- GitHub data: https://www.githubarchive.org/

- Dynamic data: One interesting phenomenon to observe is how graphs change over time. There exists some datasets that include a temporal element: `http://projects.csail.mit.edu/dnd/`. This includes, for example, DBLP, Google+, web data, brain network/connectome data.
- Movies: There is data available from MovieLens, including movies and reviews (`http://grouplens.org/datasets/movielens/`). There is also IMDB (`https://datahub.io/dataset/imdb`).

I cannot vouch personally for all of these sources (let me know if they are not good, or if there are better that you recommend), but there is lots of data available.

4 Project Ideas

Below are some project ideas that we talked about last week. (Many of these are ideas that you came up with and that I am repeating here for others to hear.)

Network analysis:

- Start with a large social network graph (e.g., the Friendster dataset with 65 million nodes), and explore this dataset. Implement efficient sublinear time algorithms to approximate various properties of the graph (e.g., estimating diameter, shortest paths, spanners, matchings, etc.). Use some combination of algorithms from this class and algorithms that you find interesting that we have not covered. You could also look at the Wikipedia graph.
- Look at how network properties change over time. Use a dynamic data set and use fast algorithms to estimate rates of change of various properties.
- Triangle counting: there are several different algorithms for counting triangles. Some handle insertions of edges, some insertions and deletions. Some are designed for streams, while others are sublinear time. Implement and compare several different algorithms for counting triangles. Which are faster? Which are more accurate? How do they scale?
- Influence and centrality: An important property of a social network is how influence propagates (and the respective "centrality" of different nodes). These ideas capture how ideas (or diseases) spread through a network, and how we can predict/control/modify the spread through networks. Experiment with algorithms for determining the influence of users in a network.
- Importance: PageRank has become famous as the technique that Google uses to determine the importance of different webpages. There are also a variety of related techniques. Implement/experiment with Pagerank and related techniques to determine the importance of nodes in a network (e.g., Wikipedia or a social networks). How do you implement PageRank efficiently? Can you implement it in a cache-efficient manner? Can you approximate it in sublinear time?

Cache-efficient algorithms:

- Compare different cache-efficient graph algorithms. For example, implement Dijkstra's algorithm use a cache-efficient Priority Queue, a cache-oblivious priority queue, a B-tree, and a regular priority queue. Is there any difference in performance? If so, how big does the graph need to be for it to matter? Alternatively, you could look at BFS, or other graph problems. For the cache-aware versions, find the best values of B and M. Test your code on different machines: do they have different optimal sizes for B and M?

- Compare different cache-efficient sorting algorithms. How does external MergeSort compare to Buffer-tree Sort? And how do they compare to FunnelSort (which is cache oblivious). How do they compare to traditional QuickSort? How big does the data need to be? For the cache-aware versions, find the best values of B and M. Test your code on different machines: do they have different optimal sizes for B and M?

- Are streaming algorithms inherently cache-efficient? Use a streaming algorithm to find a spanning tree (or an MST) of a graph by scanning the graph on disk. Analyze (theoretically) the cost of the streaming algorithm in terms of cache cost (i.e., block transfers). Implement the streaming algorithm, and implement a cache-efficient algorithm for finding a spanning tree (e.g., using cache-efficient BFS). How does the cache-efficient algorithm compare to the streaming algorithm (both in theory and in practice)? How does it compare to a classical algorithm (that is not cache-aware) for finding a spanning tree? Hopefully, we can use our knowledge of streaming algorithms to build cache-efficient algorithms!

- Examine how to implement sublinear time or streaming algorithms in a cache-efficient manner. If you are using a streaming algorithm to find a spanning tree, what is the cost in terms of block transfers for maintaining the sketch and for reconstructing the spanning tree? (Give theoretical bounds.) Can you optimize that, improving the caching performance (particularly for the graph reconstruction at the end)?

Sublinear time algorithms:

- There is a list of open problems for sublinear time algorithms here: `http://sublinear.info/index.php?title=Open_Problems:By_Number`. Experimenting with any of these problems would be interesting (even if you do not find a complete solution).

Streaming algorithms:

- Often in streaming algorithm, you want to look at time windows, e.g., you want to know the top k most common values in the last 10 minutes, in the last hour, in the last day, and in the last month. (Or you might want to know something about the number of connected components of the graph, if you only include the most recent 100 edges, 1000 edges, 10,000 edges, etc.) Investigate these types of streaming algorithms, and test them on various data (e.g., taxi data—see the next item).

- Build a streaming algorithm that takes data from the real-time taxicab location stream and analyzes it in some interesting manner. For example, if you assume that taxis within 100 meters of each other are connected, then what does the graph of taxis look like and how does it change over time? How does the number of free taxis change over time? How does the taxi density change over time? In this context, the windowed streaming algorithms may be intesting.

Other problems:

- Clustering algorithms: Often, given a set of points, we want to cluster them into groups. Can you maintain clusters (or cluster heads) in a data stream? For example, can you divide the taxis into clusters based on their location? (Do these clusters partition Singapore into an interesting set of regions?) How efficiently can you maintain these clusters over time?
- Clustering, again: Investigate how to implement clustering algorithms in a cache-efficient manner.
- Matching algorithms: Another common problem is matching, where you want to pair up items in your dataset. For example, imagine you want to match taxis with passengers. Or perhaps you want to pair up people with partners in a social network (like Facebook). Can you implement a sublinear-time algorithm for estimating the size of matchings? What if you want to match items in a stream? Can you implement the matching algorithms in a cache-efficient manner?
- Dimension reduction: Often, you can interpret a dataset as consisting of high-dimensional data. (A simple example: consider a text as a vector where each index of the vector is associated with a single word in the English language, and the value in the vector is the number of times that word appears.) We often want a way to represent this high-dimensional data in terms of a smaller number of dimensions, while preserving the distances and angles between points. The canonical technique here is based on the JohnsonLindenstrauss which involves using a random projection into a lower dimensional space. Implement this dimension reduction technique and experiment with how well it works on different data sets. Perhaps run standard clustering algorithm on the lower dimensional data as an example of how much easier it is to work with low-dimensional data!

Distributed/Parallel algorithms:

- Implement and experiment with graph algorithms in a MapReduce framework (e.g., using Hadoop or Pregel). How fast can you solve classic graph problems? (To complete this project, you may need to find for yourself a network of computers to run your experiments on, e.g., using Amazon Web Services.)
- Can you use the sketching algorithm from class to build an efficient spanning tree in a distributed network? Can you use it to build an MST? (Talk to me for ideas.) Simulate such an algorithm, and see how well it works. Compare it (both in theory and in your simulation) to more classical distributed algorithms for finding a spanning tree.
- Can you use the streaming algorithms from class to solve aggregation/data analysis problems in sensor networks? (You might look at Synoptic Algorithms as one example.) Simulate such algorithms and see how well they work.

- Recently, there has been a lot of work on implementing efficient algorithms on GPUs. Choose one of the graph algorithms from this class and implement it on a GPU. Run experiments to understand the performance improvements. (To complete this project, you may need to find for youself a machine with a GPU.)