CS5234: Combinatorial and Graph Algorithms

MiniProject Instructions

Mini-Project Instructions

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., sublinear time algorithms, streaming algorithms, cache-efficient algorithms, or parallel/distributed algorithms. Projects can have an implementation/experimental component, though theory/algorithm-oriented projects are certainly appreciated as well. In general, your project projects will have different aspects:

- *Algorithms*: Develop and analyze new algorithms. This part should include intuition, detailed explanation, and careful analysis.
- *Experiments*: Run experiments to determine how algorithms work on real-world data. This part should illustrate how your algorithms work on different types of data sets.
- *Explanation*: Draw conclusions as to how we should best use these algorithms. What should we learn from your work?

Some projects will focus more on one of these aspects than the others (e.g., will focus more on algorithm design and analysis and less on experiments, or vice versa). However, most projects will contain at least a little bit of all three. For example, you might take an existing algorithm, explain well how it works, modify/optimize it so that it performs better, and run some experiments exploring how well it actually works.

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 think about 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!)

At the end of this document, I will propose four specific problems that you might choose to work on. Feel free to suggest your own topic instead.

2 Organization Details

You may work on the mini-project in teams of two. By Thursday October 25, 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. One week later, by November 1nd, you should submit a one-page summary of what you are planning to do and what you have learned so far. On November 8, you should submit a progress report that contains preliminary results. On November 15th in class, you will give a short presentation on your mini-project. (Slides for the presentation must be submitted by the end of November 14th.) The final report is due November 18th.

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. 5	Project choice and team formation
Nov. 1	One page plan and progress report
Nov. 8	Interim report
Nov. 15	Presentation (in class)
Nov. 18	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.

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.

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.

4 Project Ideas

4.1 Clustering

A key technique in data analysis is clustering: take a large collection of (often, high-dimensional) data points and group them into a set of clusters where all the data in a cluster is "close together." There are typically two metrics we care about in clustering algorithms: (i) accuracy (i.e., how good are the clusters?) and (ii) speed. Your goal is to develop the fastest, most accurate clustering algorithm possible. That requires balancing the trade-off between performance and speed.

In this project, focus on sublinear time approximation algorithms for clustering. That is, your algorithm will look at only a small component of the dataset (e.g., using sampling), and from this derive clusters for the entire dataset. Your tasks here are as follows:

1. Notice that there are many different clustering problems, e.g., k-means, k-median, community detection, graph partitioning, etc. There are also a variety of metrics to determine how good a clustering is. Your first task is to identify a few candidate data sets, along with the clustering goals that are appropriate for that data set. (For example, if you are looking at a social network, community detection may be the right notion. A geographic dataset or network data set may be best examined via a graph partitioning strategy. Etc.)

2. Explore existing algorithms for solving the clustering problem, but sublinear time approximation algorithms and standard ("baseline") algorithms. Choose a few candidate algorithms.

3. Implement the algorithms in question and run some preliminary experiments.

4. Improve the existing existing algorithms. This might involve:

   - Developing a better algorithm.
   - Combining two existing algorithms to yield better performance (e.g., using one algorithm in some cases and a different one in other cases).
   - Applying a new technique to improve the performance. (For example, you might try using Johnson-Lindenstrauss dimensionality reduction techniques to further speed up the algorithm, if the data is high-dimensional.)
   - Optimizing the implementation so that it runs faster (e.g., taking advantage of caching techniques or parallelism).
   - Taking advantage of special structure in your problem (e.g., the graph is dense, planar, has a heavy-tail distribution on degree, etc.).

5. Analyze your improved algorithm and show that it is correct.

6. Run a final set of experiments comparing the baseline algorithm, the original algorithm, and the improved algorithm. Be sure to examine the trade-off between running time and quality of the clustering. (Does your algorithm have any tunable parameters that allow you to achieve different points on the trade-off space?) Ideally, you would test your algorithm on different types of dataset to better understand which ones your algorithm is more suitable for. It may be useful (in addition) to generate synthetic datasets that allow you to carefully explore which parameters of the dataset are most important for your algorithm.

4.2 Cache-efficient distance computation

One of the benchmark problems for graphs is finding shortest paths: given a graph, find the shortest path between some node u and some node v (or, alternatively, find the shortest paths between all pairs of nodes). Your goal in this problem set is to develop the fastest algorithm you can for finding shortest paths. Specifically, your goal is to take advantage of caching performance to do better than traditional algorithms.

There are three specific problems to consider in this project:

- Given an unweighted graph, find the shortest path from some node u to some node v. This is, effectively, a BFS problem. A first simple experiment would compare classical BFS to a cache-efficient BFS algorithm.
- Given a weighted graph, find the shortest path from some node u to some node v. The classical solution to this is Dijkstra's algorithm. Using techniques from class, you could use a cache-efficient priority queue and (potentially) see better performance.
- Given a weighted graph, find the shortest path between all pairs of nodes. The classical version of this is Floyd-Warshall. You can likely see significant improvements in speed by using a cache-efficient matrix multiplication algorithm to replace the inner loop of Floyd-Warshall.

Your tasks in this project may be enumerated as follows:

1. A first goal might be to compare the traditional algorithms to those we have seen in this class. Find a good (big) data set, implement the algorithms in question, and compare the performance.
2. At that point, the real task at hand is to try to do better than the algorithms we saw in class. This might involve using more advanced algorithms, developing new techniques, optimizing further, etc. You might accomplish this by:
   - Developing better algorithms.
   - Optimizing your existing implementation.
   - Combining ideas from multiple existing algorithms to form one better algorithm.
   - Apply similar techniques to a harder problem (e.g., generalize from unweighted matching to weighted matching).
   - Taking advantage of special structure in your problem (e.g., the graph is dense, planar, has a heavy-tail distribution on degree, etc.).
   - Consider the benefits of approximation, i.e., can you make the algorithms a lot faster if you are willing to tolerate a 10% error?
3. Prove that your new algorithms are correct.
4. Run a final set of experiments comparing the baseline algorithm(s), the class algorithm(s), and your new algorithm(s). Be sure to examine any trade-offs that arise. Ideally, you would test your algorithm on different types of dataset to better understand which ones your algorithm is more suitable for. (For example, some algorithms may perform better on planar graphs, while others might perform better on dense graphs.)

In general, the baseline algorithms here are pretty efficient. BFS is very fast! Dijkstra's algorithm is fairly fast! To see any real improvement, you will need a very large dataset that cannot be stored in memory, i.e., that will cause multiple disk reads in order to complete the computation. While you may want to test your algorithms first on smaller data sets, you should be sure to run some of your experiments on very, very large data sets (where running times are in the range of hours, not seconds). And while your main goal is to develop an algorithm that runs as fast as possible, you may want to measure the cache misses caused by your algorithm as well. (You can often find this information by looking at the hardware performance counters.)

4.3 Map-Reduce Algorithms for Matching, Coloring, and Densest Subgraph

In this project, you explore one of the canonical graph problems in the Map-Reduce model.

- *Matching:* We have already talked in class about the problem of finding a maximal matching. In this case, the challenge is to implement such an algorithm in the Map-Reduce model. How do you deal with weighted edges?
- *Coloring:* The problem of coloring is to assign a color to each node in the graph so that no two neighbors share a color. In some ways, this is a generalization of the MIS problem that we considered in class (and can be solved using similar techniques).
- *Densest subgraph:* The problem of finding a dense subgraph is to find a collection of nodes that have a lot of edges connecting them. (Such a dense subgraph often represents a community in a social network.)

For many of these problems, there exist good Map-Reduce algorithms. For some variants of these problems, you will have to develop your own algorithms by generalizing techniques we have studied in class.

For implementing Map-Reduce algorithms, the standard framework is Hadoop. An interesting alternative is Pregel/Giraph, which is specifically designed for graphs. One interesting experiment might be comparing implementations in Hadoop and Giraph.

Plan. Your tasks in this project may be enumerated as follows:

1. A first goal might be to implement standard parallel solutions and see how well they work on different data sets. (Be sure to compare the performance against simple sequential algorithms: if the parallel performance is worse than sequential performance, then it is not very useful!)
2. Next, try to do better than these existing algorithms. You might accomplish this by:
    - Developing better algorithms.
    - Optimizing your existing implementation.
    - Combining ideas from multiple existing algorithms to form one better algorithm.
    - Apply similar techniques to a harder problem (e.g., generalize from unweighted matching to weighted matching).
    - Taking advantage of special structure in your problem (e.g., the graph is dense, planar, has a heavy-tail distribution on degree, etc.).
3. Prove that your new algorithms are correct.
4. Run a final set of experiments comparing the baseline sequential algorithm(s), the standard algorithm(s), and your new algorithm(s). Be sure to examine any trade-offs that arise. Ideally, you would test your algorithm on different types of dataset to better understand which ones your algorithm is more suitable for. (For example, some algorithms may perform better on planar graphs, while others might perform better on dense graphs.)

4.4 Streaming Algorithms and Cache Efficiency

One of the key advantages of streaming algorithms is that they are very cache-efficient, i.e., they iterate through the data only once (or a small number of times). Can you use existing streaming algorithms to develop faster cache-efficient algorithms? For example:

- We saw in class how we can use sketching techniques to determine the graph properties of a stream. Can we use these techniques to develop faster sequential (cache-efficient) algorithms? Beware, the sketching techniques for graphs are not immensely efficient to start with—perhaps by using multiple passes through the graph (instead of just one) we can develop a more efficient variant?
- Can we use streaming algorithms for weighted matching to device cache-efficient algorithms for finding a weighted matching? Can we use multiple passes through the data to make the algorithm more efficient?
- There are streaming algorithms (based on composable coresets) for matching and vertex cover that depend on the stream being presented in a random order. In an external memory model, we can first randomly permute the edges of the graph and then apply the existing algorithm. Does this yield a more efficient external memory algorithm?
- And there are many more streaming algorithms, some requiring multiple passes, some requiring randomized data, that might be interesting to study in the external memory model.

In this case your task is as follows:

- Choose a streaming algorithm and convert it to a cache-efficient algorithm. (Be sure to focus on making the "in-memory" part of the algorithm cache-efficient.)
- Implement three algorithms to solve your problem: a classical solution (that is not cache-efficient), the best cache-efficient algorithm you can (for the external memory model), and the best cache-efficient algorithm that you can develop by transforming an existing streaming algorithm. Compare the three solutions, running some preliminary experiments.
- See if you can improve/optimize the streaming algorithm, for example, by using multiple passes through the data, randomizing the order of the edges in the graph, or otherwise improving the solution.
- Prove that your converted algorithm is correct and analyze its performance.
- Run final experiments comparing the optimized algorithm to the three existing algorithms.

4.5 Other ideas

You may propose a mini-project on any topic related to this class, ideally following a similar structure to those described above. Below are listed some topics that you might want to explore.

General topics for further exploration: These are topics that are closely related to material in this class, but that we did not have time to explore further. Each of these topics could have been a class (or two)! If you want pointers to source material, let me know.

- PAC-learning (e.g., how to use sublinear time sampling algorithms to learn faster).
- Compressive sensing (e.g., how to use ideas like sketches to reconstruct data from sparse measurements).
- Johnson-Lindenstrauss dimension reduction (e.g., how to develop faster algorithms for higher dimensional data). This has applications to many types of algorithms.
- Graph sparsification (e.g., how to take a graph and reduce it to a sparse one that maintains many of the same good properties). There is a lot of interesting work on sampling to achieve graph sparsification, and it has been extended to the streaming model as well.
- Distributed sketches (e.g., how to use the sketching techniques in a distributed system to reduce communication complexity).

Network analysis: There is a lot of work on using techniques from this class to analyze social networks and other sorts of large-scale networks.

- 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: For search trees, there is a huge amount of evidence that caching matters. B-trees are basically faster than anything else around in large part due to their caching performance. For other problems, the question of efficiency is more complicated and there are trade-offs.

- 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)?

- Explore cache-oblivious algorithms, implementing them and comparing them to their cache-efficient counterparts.

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 yourself a machine with a GPU.)