EC 504 Project Suggestions

A major component of EC 504 is the final project, which is done in groups of up to 4-5. Below I list 5 possible projects, along with how they will be graded. Feel free to choose one of these or design your own project. If you design your own project, you should consult with me first.

A short (up to 2 page) project proposal will be due on gradescope on Thursday, October 17. This counts for 10 percent of the grade. A mid-term progress report will be due Thursday, November 14 (see below for details on what to include in this report). The final presentations will be held Dec 5 and 10; these consist of 10-15 minute (depending on time) presentations from each group discussing their project. A final report is due TBD (see below for details).

Project 1. Image Foreground/Background Segmentation using Network Flow

A basic problem in computer vision is how to segment objects in images. In class we will discuss segmenting the foreground from the background in an image. This problem is generally treated as a clustering problem, where the pixels are clustered into two groups (foreground or background).

In our discussion of applications of network flow (currently scheduled for Oct 31), we will discuss one of the canonical ways to perform this segmentation. Briefly, network flow can be used as a way to solve the segmentation problem efficiently in the case of two segments. Kleinberg and Tardos, Section 7.10, detail this construction (and we will go over it briefly in class as well).

In order to actually implement the algorithm, we require for every pixel a likelihood for the pixel to be assigned to the foreground or the background. Typically this is done by using an off-the-shelf clustering algorithm based the color (e.g., RGB values) of the pixels, such as k-means or Gaussian mixtures; existing code can be used for this step. One challenge is how to turn the output of a clustering algorithm into an appropriate likelihood score for use in segmentation. Another crucial piece of the algorithm is determining how to appropriately set the separation penalties. How these pieces of the algorithm are implemented will have a substantial effect on the resulting segmentations.

The minimum requirements (35%) for this project consists of a piece of code that takes as input an image, and computes a segmentation into two groups of pixels, along with an appropriate visualization of the two segments. The code should utilize network flow as a main technique for segmentation.

For speed purposes you may want to down-sample the input images so that the number of pixels is manageable. (Note however that too much down-sampling may result in poorer performance.)

Optional requirements (you must do at least 1 of these; this counts for 35%):

-Extend beyond two segments to k segments. Here, the network flow algorithm cannot be directly applied, but there are several techniques for extending to k segments. For example, see Section 13.6 of “Probabilistic Graphical Models” by Koller and Friedman for a discussion on non-binary segmentation.

-Improve the algorithm by enhancing the basic likelihood computation. There are many ways to try and improve a basic algorithm for computing likelihoods, such as using more than two clusters (but finding a way to use these to compute only two likelihood scores), trying to incorporate other features beyond color when doing the computation, or experimenting with other clustering algorithms.

-Develop an interactive application in which a user clicks on a piece of the image, and then (using your segmentation algorithm) an entire segment of that image is removed and a new image is created with the segment removed.

The code will be judged by the speed of segmentation as well as qualitative results on several (new) images that the instructor will use for testing the code. These results will be compared with other groups that complete this project. This part counts for 30% of the grade.

Project 2. Video Encoder

Design and implement an efficient encoder of images into video. Your encoder should be able to take an arbitrary number of JPEG files (of the same dimensions), encode them into one file, and play that file on demand.

Minimum Requirements (35%):

- Ability to encode up to 100 JPEG images (of the same dimensions) into one encoded file in under 5 minutes. The file size should never be more than the sum of the individual files.
- Ability to play back the images from the encoded file at least 10 images per second.
- Provide a command-line User Interface for encoding and viewing. Format should be:
    - encode [file1] [file2] ... [filen] --output [outputFile]: encode file1...filen, in order, into the given output file
    - view [outputFile]: display the images encoded in outputFile, at least 10 images per second

Extensions (35%):

Mix and match these features, as you wish. The maximum total score for this section will be 35% of the grade, but no credit will be given unless all the minimum requirements are met.

- Ability to store arbitrary binary files. [5%]
- Develop a Graphical User Interface that provides an intuitive method for entering and/or reorganizing files, progress bar for encoding, and controls for viewing the encoding. [10%]
- Provide three "interesting" real-time video effects during playback [10%]
- Allow user to adjust parameters on movie creation in order to trade disk space for quality [10%]
- Develop an Android client for viewer [15%]
- Make your encoder file MPEG-1 compliant [20%]

Evaluation (30%):

Your protocol will be compared to others in class (and, potentially, my own) on an (unknown) reasonably large testbed and ranked according to:

- Ratio of number of images to encoded file size.
- Speed of encoding/decoding.

Your grade on this feature will correspond to your rank.

Project 3. De-Duplicator

Design and implement an efficient data storage locker that utilizes deduplication. Your locker should be able to receive files and store them (for later retrieval) with a minimum storage by storing some common data blocks only once. For example, a first file containing "This is an example" might store all characters, but a second file containing "This is an example too" might store a pointer to the first file followed by " too".

Minimum Requirements (35%):

- Ability to store ten 10MB ASCII files, any two of which differ in at most 5 character edits (character edits are insertions of a character, deletion of a character, or modification of a character), using at most 20MB of storage.
- Ability to retrieve all files stored in the locker in any order and at any time.
- Your program should not require any live state (i.e. it should be possible to stop and restart the program, even on a different computer, with the storage locker contents in order to reproduce the stored files).
- Command-line User Interface that allows users to insert files into the locker and displays current storage usage. Format should be: store -file [file] -locker [locker location].

Extensions (35%):

Mix and match these features, as you wish. The maximum total score for this section will be 35% of the grade, but no credit will be given unless all the minimum requirements are met.

- Ability to store arbitrary binary files. [5%]
- Develop a Graphical User Interface that storage progress (and file allocation) in real time. [10%]
- Develop networked access to your locker [10%]
- Ability to store directories of files as one entity. [15%]
- Develop an Android client for your locker. [15%]
- Implement file deletion from your locker [20%]

Evaluation (30%):

Your protocol will be compared to others in class (and, potentially, my own) on an (unknown) reasonably large testbed and ranked according to:

- Ratio of number of files to storage utilized.
- Speed of insertion/extraction from the locker.

Your grade on this feature will correspond to your rank.

Project 4. An Application of Space-Efficient vEB Trees

In class we discussed several advanced data structures, including Fibonacci Heaps and van Emde Boas trees. While vEB trees have beautiful theoretical properties, they also have limitations: i) they are difficult to implement, ii) the standard implementation requires O(u) space, which is prohibitive, iii) it is not clear how well they do in practice.

The minimum requirements (60%) are to implement the space-efficient van Emde Boas tree (see, e.g., problem 20-1) and *find an interesting application* of this structure. You should also compare using the vEB tree to a more basic structure, and determine if the effort of coding the vEB tree is worth it.

Extensions (40%): you should make some enhancements over the basic requirements. Possibilities include implementing other data structures for comparison, or finding a second application. Also, depending on the application you choose, there may be many possible extensions you can do on the basic application (e.g., create a front-end for your application, etc.).

Project 5. Hashing and Trees for Fast Approximate Nearest Neighbor Search

One of the standard problems in information retrieval is the nearest-neighbor problem. Given an object and a database, quickly retrieve the most similar object to the query from the database. A naive implementation would require one to linearly scan through the entire database, which is typically too costly for online applications.

Two common approaches that have been suggested for this problem are locality-sensitive hashing and kd-trees. Both may be viewed as data structures / algorithms for finding an approximate nearest neighbor to a query.

In this project, you will explore and implement these two techniques, and test them on a data set (for example, the 80 million tiny images dataset from MIT, where nearest neighbors can be computed directly using L2 distance over the computed gist features of these images; or a zip code locator based on latitude and longitude information). Compare the performance of these approaches in terms of speed and accuracy (i.e., how often do these methods retrieve one of the top few actual nearest neighbors).

Optional extensions:

-Explore extensions beyond basic LSH and kd-trees (there are many), and find an extension that yields better performance.

-Develop an interactive application that allows one to search quickly through a large database of files (e.g., images).

-Suggest and test your own extension to one of these algorithms.

Proposals

Header

This should contain:

i. The name of your project and a brief description (or link to the description).
ii. The list of active group members.
iii. The *final* feature list that you plan to implement (from the project description). Any deviations from this feature list will need a written explanation in the final report.

References

An updated list of references that you are using for your project. As before, references should be specific, complete and high-quality (i.e. from reputable, well-reviewed sources).

Schedule

A complete schedule, including:

i. A timeline of what has been completed already, and when it has been completed.
ii. A timeline of what has yet to be completed, and as detailed a plan as possible for when it will be completed.

iii. A detailed statement of what has been done so far by every person in the group, and what has yet to be done. Recall that every member must have a substantial technical contribution to the project.

Mid-Term Report

In addition to the above, for the mid-term report, include the following:

Prototype
Your submission must include a runnable prototype of your project. Obviously, a significant technical component of the project might not yet be ready, but I would like to see that you have a working framework for the final project, ideally with trivial (or simple) components. Your submission must include:

- All code *and* clear instructions needed to compile and run your code.
- A clear and specific written description of what your prototype does, and its current limitations.

Challenges
A descriptive list of the problems that you are currently tackling. Most important here is a description of challenges that you feel may challenge your ability to complete the project or some features in your final list.

Final Submissions

I. **Documentation**
    - description of the problem
    - relevant references and background materials
    - high-level description of the implementation, with particular emphasis on the design decisions related to data structures and algorithms
    - a list of the features from the project proposal that have been implemented
        - for each feature implemented, provide a description of how it was implemented with an emphasis on data structures and algorithms used

II. **Code**: complete, working code for your implementation
    - clear and terse instructions how an average student in class can compile and run your code from scratch

III. **Supporting files**
    - all libraries needed for your project to run, and instructions for how to freely (and legally!) acquire them
    - examples of how to use your project

        - testing patterns
            - unit tests
            - system tests

IV. **Work breakdown**

- a statement detailing what each member of the group contributed to the project, signed by all members of the group.

Keep in mind that:

- Severe penalties will be assessed for any code that I cannot run myself from scratch. I will only invest what I consider a reasonable amount of effort to run your code, so you need to invest some effort to make sure that your descriptions are clear. Feel free to test your code on someone who is not familiar with your work.
- Little to no credit will be given to any code that is not working or which I can cause to substantially misbehave. You are better off not submitting features that are buggy (or specifying the bugs away).