Image-Processing Projects for an Algorithms Course

Eugene Fink **Michael Heath**

eugene@csee.usf.edu heath@csee.usf.edu

Computer Science and Engineering

University of South Florida

Tampa, Florida 33620

Abstract

Courses on algorithm analysis often include little programming, and do not emphasize application of advanced techniques to practical problems. On the other hand, students usually prefer hands-on learning, and may lack motivation to study theory.

We augmented an algorithms course with a series of programming tasks, which involved application of the course material to image processing. These tasks motivated the students, and led to better understanding and retention of theoretical material. They also enabled the students to learn the basics of representing and manipulating images, along with the algorithm theory.

Keywords: Algorithm theory, image processing, computer vision, undergraduate education.

1 Introduction

Software development often involves advanced data structures, and requires knowledge of algorithm theory. The ability to choose and implement appropriate algorithms is a crucial skill for a software engineer [7].

Undergraduate students learn relevant theory through courses on data structures and algorithms; however, they often have difficulty applying the learned theory in actual programming. Some students initially perceive algorithms as impractical material, and the right motivation has proved an essential part of theoretical courses. Even graduate students are sometimes reluctant to utilize algorithm theory in their research, and need close guidance in their implementation of advanced techniques.

This problem may be partially due to the traditional methods of teaching the algorithm analysis. A survey of university curriculums reveals that advanced algorithms courses usually include little programming, and do not emphasize the role of theoretical techniques in software development. An integration of an algorithms course with programming tasks may help the students to understand theory and develop skills for its application.

To test this strategy, we included programming projects into a senior-level algorithms course at the University of South Florida. The projects were centered on image processing, and the students applied the learned theory to developing efficient image-analysis programs. The use of this common theme allowed the students to acquire basic experience with digital images, along with a deeper knowledge of algorithms.

This use of programming tasks was a continuation of work by Sarkar and Goldgof, who integrated image analysis into a sophomore-level course on data structures [5]. They emphasized the growing need for images in software development, and suggested that basics of image manipulation should become a part of the core curriculum. Their integrated course proved a success: the students not only learned the foundations of image analysis, but also deepened their understanding of data structures.

The described programming projects are also related to recent work by Stevenson, who used computational geometry and image analysis in an algorithms course [6]. His projects included convex-hull computation, triangulation, and convolution; they emphasized algorithm design strategies and advanced programming techniques, such as object-oriented programming, graphical user interfaces, and multiple threads.

On the other hand, we focussed on the implementation of specific algorithms and data structures, empirical analysis of their performance, and its comparison with the theoretical time complexity. We outline the course (Section 2), describe programming projects (Sections 3 and 4), and conclude with a discussion of the observed results (Section 5). The reader may find a more detailed description of the course and all related materials at `www.csee.usf.edu/~eugene/algs/`.

2 Course overview

The undergraduate algorithms course at the University of South Florida is a fifteen-week course that covers standard foundations of algorithm analysis and design; the class size is about forty students. The course includes a series of theoretical homeworks, two midterm exams, and a final. It usually involved little or no programming, and did not provide a strong tie between theoretical concepts and specific applications.

The students take a data-structures course before the algorithms class. This prerequisite course includes basic structures and their implementation in C; the students learn to work with arrays, linked lists, stacks, queues, hash tables, trees, recursion, and simple sorting. The algorithms course is a continuation of this material, which includes not only standard algorithms and design techniques, but also advanced data structures, such as priority queues and disjoint sets.

Many students feel that this material is harder than other courses, and that the underlying theoretical tools are counterintuitive. They often have difficulty with the concepts of time and space complexity, as well as with properties of advanced data structures. The lack of intuitive understanding not only makes the course harder, but also leads to retaining less material after the exam.

Traditionally, the course did not emphasize the use of multiple algorithms in a single application, and students did not acquire skills for integrating several techniques. For example, some students complained that the use of disjoint-set operations in graph algorithms was confusing, because it required integration of two different data structures.

We added image-processing projects with the goal to help the students acquire hands-on experience with algorithms. Most students had no prior experience with images; since Sarkar and Goldgof had not taught data structures in the previous year, the students had not taken their integrated course on data structures and image analysis.

We used the textbook by Cormen *et al.* [1] and covered the topics listed in Figure 1. The students had to complete not only programming projects, but also nine paper-and-pencil homeworks, which asked them to simulate algorithms by hand, design new algorithms using pseudocode, and analyze time complexity (see example questions in Figure 2). The homeworks covered all topics of the course, whereas the projects utilized only a subset of the material (see the italicized topics in Figure 1). The homeworks were worth 25% of the final grade, and the programming tasks amounted

Undergraduate Algorithms Course

- Mathematics review, asymptotic notation, and recurrences
- Sorting: *Insertion sort,* merge sort, heap sort, *quick sort, counting sort,* and radix sort
- Data structures: *Priority queues,* binary search trees, and *disjoint sets*
- Graphs: *Representation, search,* topological sort, spanning trees, and shortest path
- Dynamic programming and greedy algorithms, including *Huffman codes*
- Introduction to NP-completeness

Figure 1: Main topics of the algorithms course; the italics show the theoretical material underlying the image-processing projects (see the list of projects in Figure 3).

to 15%; the remaining points were divided between the two midterms and final exam.

3 Programming projects

The course included the four projects listed in Figure 3. The first project familiarized the students with image encoding, whereas the other three involved implementation and testing of efficient algorithms. For each project, the students had to integrate several theoretical concepts, from different textbook chapters.

The empirical comparison of different algorithms was a significant part of these tasks. For example, the median-filtering project required the students to plot the running time of three sorting algorithms, and analyze their relative performance for different array sizes. As another example, the students had to compare two different ways of implementing disjoint sets in the connected-component algorithm. Thus, they observed close relationship between theoretical time complexity and empirical

- Show the operation of the counting sort on the array $\langle 6,3,4,5,6,4,3,8,4,1,2\rangle$
- Write an efficient procedure that prints out all nodes of a binary search tree whose keys are between two given values, *min* and *max*
- Give a nonrecursive version of depth-first search; the time complexity should be the same as the complexity of the recursive version

Figure 2: Sample questions from the paper-and-pencil homeworks.

Representation of images: Image encoding and basic operations with arrays.

Median filtering: Insertion sort, quick sort, and counting sort.

Connected components: Graph representation, search, and disjoint-set operations.

Compression of images: Priority queues and Huffman's compression algorithm.

Figure 3: Programming projects and the underlying theoretical concepts.

efficiency.

We used images in the Portable Gray Map (PGM) format, which allowed simple low-level operations for loading images and accessing pixel values. The students could view PGM images on both Unix and PC machines, by converting them to either PCX or GIF, and then using a standard image viewer, such as `xv`. We provided the images given in Figure 4 for testing the students' programs. The first two pictures are NASA astronomical images, the third is a mammogram, and the others are from the image repository of the Vision Lab at the University of South Florida.

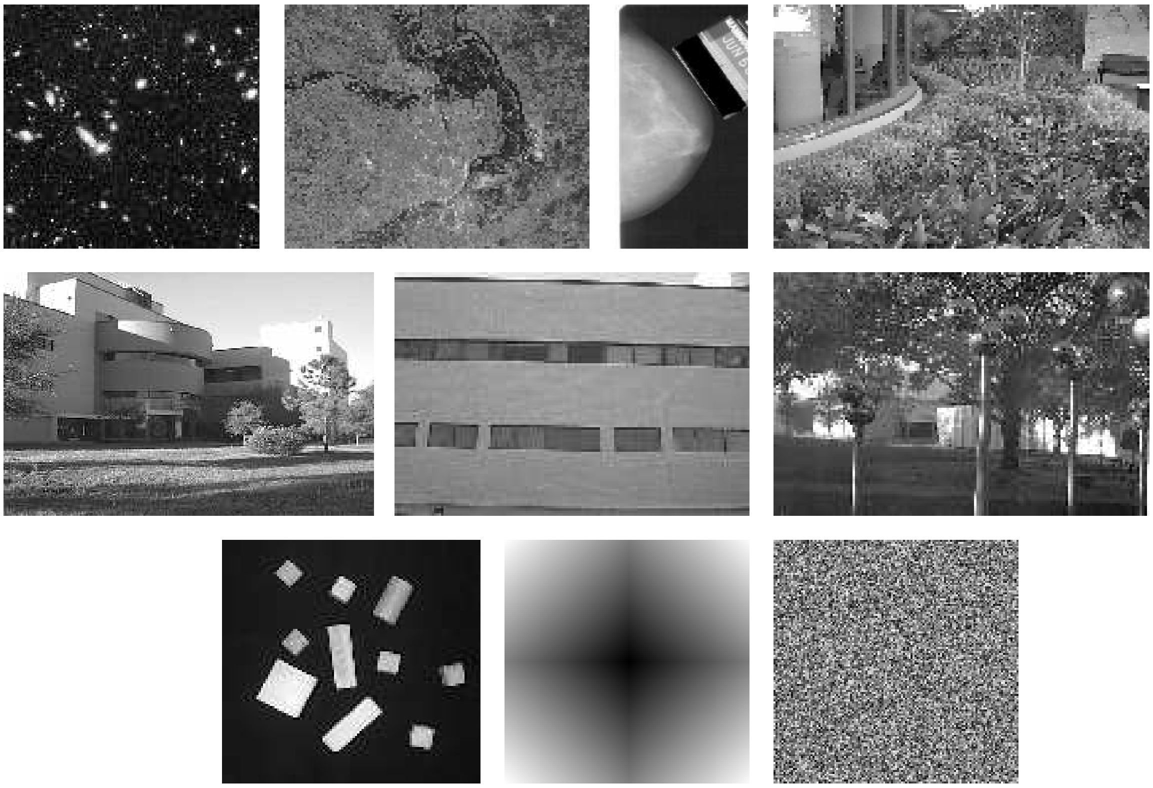

Figure 4: Images for the programming projects.

The students had to program in C or C++; most students selected C, because they learned it in the data-structures course. For each project, we provided code that supported relevant low-level operations, and the students had to integrate their implementation with the pre-packaged procedures. This code included pre-defined data structures, functions for loading PGM images, and tools for debugging and running the project implementation. It helped the students to avoid PGM-specific details and concentrate on the algorithms. In addition, it led to the standardization of output, and allowed the use of scripts for testing the submitted programs.

The students had to complete each project in three weeks. We encouraged them to work in groups of two or three, but also allowed individual work; since the project size did not depend on the group size, most students chose to work in groups of three.

They had to e-mail their source code and then give a ten-minute demo; in addition, the teaching assistant used a script to run the submitted code on several test files.

The students also submitted reports with performance graphs and answers to related questions; they included empirical results and compared the observed running time with the expected asymptotic behavior. In addition, they had to write a summary of their experience with the project and point out the hardest part of the work. The project grade depended on the correct functionality of the implementation (40%), code readability and good interface (30%), and quality of the report (30%).

4 Specific tasks

We now describe each programming project and the related theoretical material. We started with a simple well-defined project, and then gradually increased complexity of underlying algorithms and software design. Toward the end, the students had freedom to construct their own data structures.

Project 1: Representation of images

Since most students had no previous experience with images, the goal of the first task was to familiarize them with image encoding. They had to understand the PGM format and write their own basic functions for loading images into the memory and saving them to the disk. In addition, they implemented a procedure for finding the brightest pixel, darkest pixel, and mean gray level of all pixels in a given image.

The students tested their code on several images, measured the running time of each function, and determined the dependency of the time on the image size. In particular, they compared the time of disk operations with the time of linear search

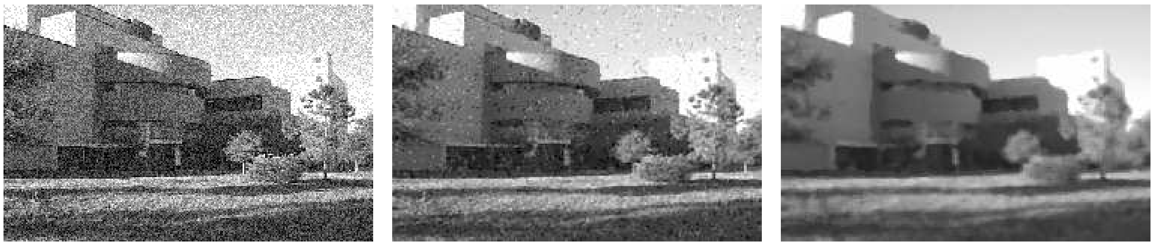

Figure 5: Images in the median-filtering project: The initial noisy image (left), and the results of filtering with a 3×3 window (middle) and 7×7 window (right).

in memory, and verified that the search time was proportional to the array size. All students successfully completed this task and received the full grade; two submissions earned extra credit for an exceptionally good interface.

The project helped the students to understand the representation of images in computer memory, and to learn tools for viewing and printing out images. Many students reported that learning PGM was harder than the implementation part, and some of them had difficulty with the concept of representing an image as a collection of numbers. Sarkar and Goldgof [5] observed a similar conceptual barrier in their data-structures course.

Project 2: Median filtering

The next project required the students to apply sorting algorithms to median filtering, which is a technique for smoothing a noisy image. For each pixel in the image, the filtering procedure considers a rectangular window centered on that pixel, computes the median of pixel values in the window, and replaces the original pixel with the median value (see Figure 5).

The students computed the median by sorting the pixel values in the window and then picking the middle value. We provided functions for loading an image from the

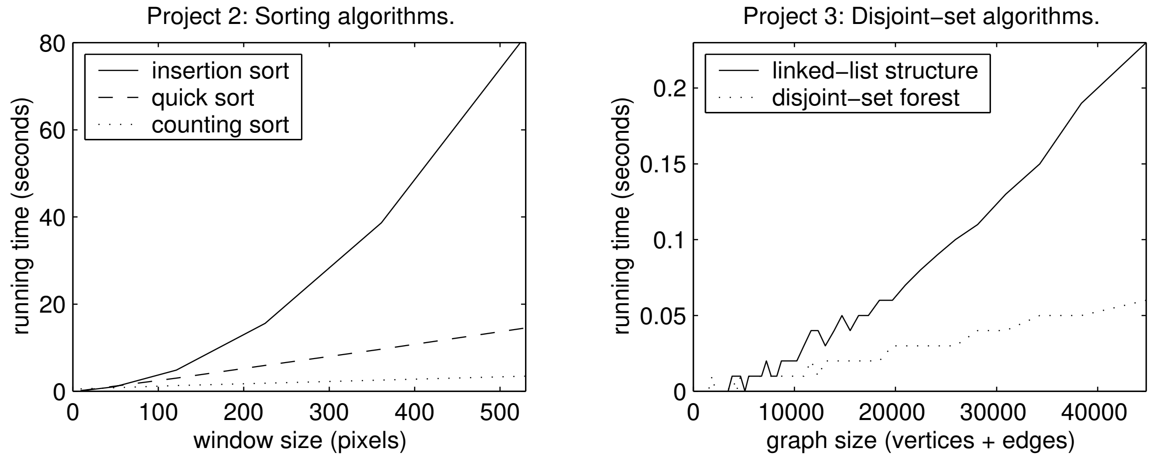

Figure 6: Experiments with sorting algorithms (left) and disjoint-set structures (right).

disk, identifying the window for a given pixel, and converting this window into a linear array. The students had to implement sorting algorithms and integrate them with the pre-packaged code; they wrote and tested three algorithms: insertion sort, quick sort, and counting sort.

We asked the students to plot the performance of each algorithm for different window sizes, and compare the resulting curves with the theoretical time complexity (see an example graph in Figure 6, left). This experiment demonstrated the importance of time complexity for large arrays.

The students also had to determine the "constant factor" of each sorting procedure, hidden in the asymptotic notation. In addition, they compared the sorting techniques for small arrays, which showed the practical significance of the "constant factor." They observed that the insertion sort is the fastest technique for 3×3 and 5×5 windows, despite its poor asymptotic behavior. All students submitted correct code; however, three groups lost part of the credit for insufficient written analysis of their results.

Project 3: Connected components

Next, the students wrote a procedure for identifying white objects on dark background, and applied it to count the number of stars in an astronomical image. They converted an image into an undirected graph, and utilized an algorithm for identifying connected components, described in Section 22.1 of the textbook by Cormen *et al.* [1]. The algorithm included basic graph operations along with disjoint-set structures.

The task involved implementation of the connected-component algorithm and three data structures: an adjacency-lists graph, a linked-list representation of disjoint sets, and a disjoint-set forest. We provided a function for converting an image into a graph, and the students used it in their implementation.

They first tested the algorithm with linked-list sets, and then ran the same algorithm with disjoint-set forests. They compared the efficiency of these two structures and plotted the respective running times (see Figure 6, right). In addition, the students again compared the observed running time with the theoretical complexity and determined the constant factor. The project convinced them that the second representation is more efficient, and that it readily scales to very large sets.

Most students reported that integration of a graph structure with disjoint sets was the hardest part of the project, as they had little experience with combining multiple structures. They had to figure out that the same object served both as a vertex of the graph and an element of the disjoint-set structure. Almost all students successfully completed the project; ten groups out of seventeen received the full grade, and six groups lost part of the credit for their written reports. Only one group was unable to complete implementation.

Project 4: Compression of images

The purpose of the last task was to familiarize students with greedy algorithms. They had to implement a compression program based on Huffman codes (see Section 17.3 of Cormen *et al.* [1]), which served as an example of greedy design, and apply it to several image files.

The students wrote a procedure that determined the frequency of pixel values in an image and used the resulting frequency table to construct a tree of Huffman codes. We provided a function that used the resulting codes to compress and uncompress images, as well as a transformation function for reducing the image entropy, which improved the compression rate.

The students had to implement not only the greedy-choice algorithm, but also two related structures: a tree of Huffman codes and a priority queue of pixel-value frequencies. Again, most students wrote that integration of two different structures was the hardest part.

After implementing Huffman's algorithm, the students measured its running time and compression rate for a collection of images, and compared it with the `gzip` procedure, which uses the Lempel-Ziv algorithm. They also had to explain why the implemented greedy procedure gives globally optimal results.

The students reported that this task was harder than the other three projects. It required more work on design and integration of data structures, and the resulting code was almost twice larger than the previous projects. Ten groups submitted working code, five groups did not fully debug their code and received partial credit, and two groups were unable to design the necessary structures.

5 Results and discussion

We have described a series of programming tasks, which allowed us to teach basics of image processing along with algorithm analysis. Since visual data are now an integral part of many computer applications, we expect that image manipulation will become an important skill for software engineers [2]. The projects did not distract from the theoretical material, and we covered the same amount of material as in the previous years. This conclusion was consistent with the results of Sarkar and Goldgof [5], and Stevenson [6], who also reported that image-analysis projects did not interfere with the course material.

Most students liked the hands-on style of learning and enjoyed their experience with visual data. They rated the course as 4.32 out of 5.00, which was higher than the departmental mean of 3.94. This evaluation was surprisingly high for the algorithms course, which had usually been rated below the mean.

After the students completed the third programming task, we conducted a short survey, which also confirmed the effectiveness of the projects. Specifically, 80% of the students replied that the projects helped them to understand the material, 88% confirmed that they liked working with visual data, and 68% indicated that they wanted to get a fourth programming task.

The projects provided convincing evidence of the practical applicability of algorithms, thus motivating the students. The hands-on experience helped the students to grasp complex concepts, and the exams revealed an improvement in their understanding and retention of the material. In particular, the empirical comparison of running times helped them to understand the notions of time complexity and scalability.

The students practiced working in groups and integrating their implementation with pre-packaged code, which are crucial skills in software development. Furthermore, they learned to combine several algorithms and data structures for designing an efficient application.

We believe that the described projects can be readily added to existing data-structures and algorithms courses, with little time overhead. The use of these projects does not require expertise in image analysis, and other teachers can readily re-use our course materials. The reader may find more ideas for image-processing tasks in the article by Sarkar and Goldgof [5], which provides a larger set of programming projects, and in Samet's textbook on spatial data structures [4].

Course materials

The course materials are available at `www.csee.usf.edu/~eugene/algs/`. They include the syllabus, theoretical homeworks, and programming tasks, along with all related handouts, PGM images, and C code. The sample solutions for the homeworks and solution code for programming projects are available on request.

Acknowledgments

We are grateful to the faculty members of the Vision Lab at the University of South Florida, Dmitry Goldgof, Sudeep Sarkar, and Kevin Bowyer, who helped to select image-processing tasks and integrate them into the course. Dmitry Goldgof and Sudeep Sarkar provided valuable comments on the contents of the article. We also appreciate the help of Josh Johnson, the teaching assistant for the algorithms course.

The test images for the course included NASA photographs by a space-borne radar and by the Hubble telescope. The work on the materials for the programming projects was partially sponsored by the National Science Foundation grant No. DUE-9980832.

References

[1] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

[2] B. B. Maxwell. "Teaching computer vision to computer scientists: Issues and a comparative textbook review," *International Journal of Pattern Recognition and Artificial Intelligence,* 12(8), 1998, pp. 1035–1051.

[3] R. Murphy, "Teaching image computation in an upper level elective on robotics," *International Journal of Pattern Recognition and Artificial Intelligence*, 12(8), 1998, pp. 1081–1093.

[4] H. Samet, *The Design and Analysis of Spatial Data Structures,* Addison-Wesley, Reading, MA, 1990.

[5] S. Sarkar and D. Goldgof, Integrating image computation in undergraduate level data-structure education, *International Journal of Pattern Recognition and Artificial Intelligence,* 12(8), 1998, pp. 1071–1080.

[6] D. E. Stevenson, "Computational geometry and image processing applications for an undergraduate algorithms course," in *Proceedings of the Workshop on Undergraduate Education and Computer Vision,* 2000.

[7] R. Wodaski, *C Programming Proverbs and Quick Reference,* Sams Publishing, Carmel, IN, 1992.