PA3 The Art of Compression

Goals and Overview

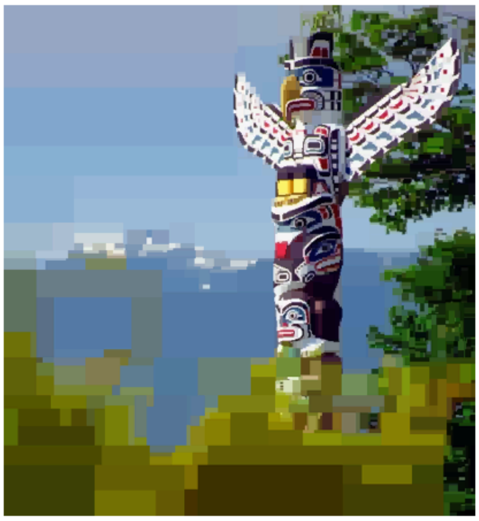
In this PA (Programming Assignment) you will:

- learn about algorithmic art
- learn about an algorithm for lossy image compression
- learn about space partitioning trees similar to KD Trees
- learn about clever mechanisms for speeding up statistical algorithms

Part 1: Inspiration and Background

The inspiration for this assignment came from an article about an artist whose work recreates classic portraits with a blocky effect. The exact algorithm he used is not given in the article, but we will create similar images using a strategy for image representation that underlies common lossy image compression algorithms.

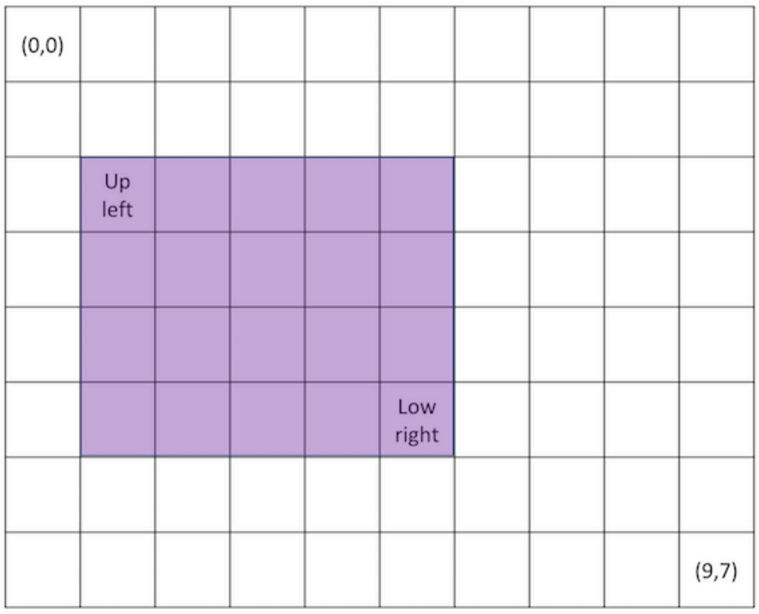
The two images below illustrate the result of this assignment. Note that the second image sacrifices color detail in rectangles that do not contain much color variability in the original image, but maintains detail by using smaller rectangles in areas of the original image containing lots of variability.



In specifying the algorithm, we make several conventional assumptions, illustrated below:

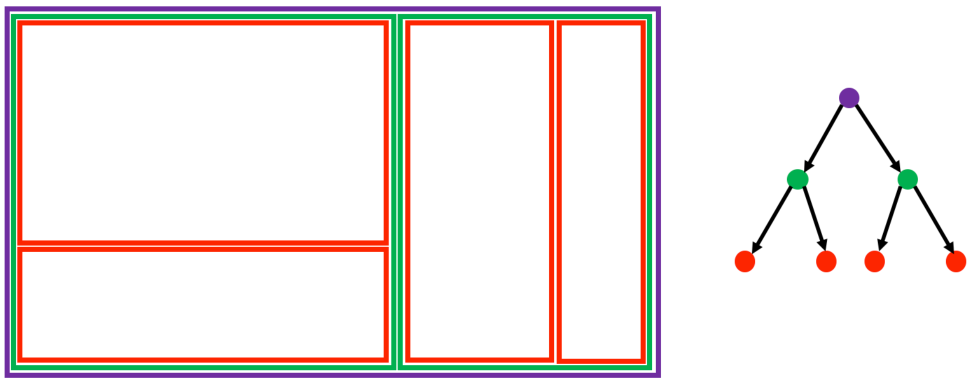
- The origin, position (0,0), is in the upper left corner of the image.
- Rectangles are specified by a pair of points at its upper left and lower right corners.
- The upper left corner of a rectangle is the one nearest to the origin.

Image locations are typically specified as (x,y), where x is a horizontal offset, and y is a vertical offset.



The original image will be represented in memory as a binary tree whose nodes contain information about rectangles.

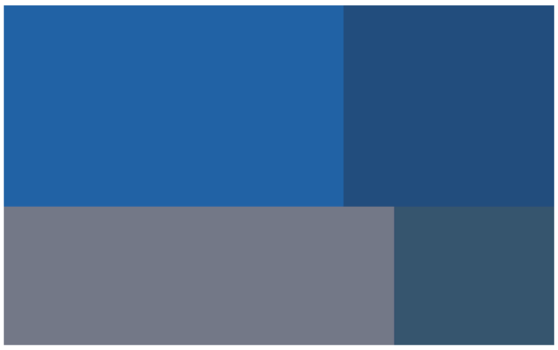
- Every node contains upper left and lower right points that specify a rectangle.
- Every node also contains a pixel that represents the average color over the rectangle in the original image.
- When a rectangle is split into two smaller rectangles, the parent node contains the whole rectangle, the left child contains the parent's upper left corner and a new lower right corner, and the right child contains a new upper left corner and the parent's lower right corner.
- Rectangles can be split either horizontally or vertically.



Splits are performed so as to minimize color variability across the resulting rectangles. The variability measurement we use is the sum, over all pixels, of the squared difference from the rectangle average. We compute these differences independently for the red, green, and blue, color channels, and then we sum them to get a total variability score. The formula for the squared difference is given on the left, and a computationally more convenient, but algebraically equivalent version is given on the right. (The area of rectangle *R* is given by $|R|$, and all sums are taken over $x \in R$

$$\sum (x-\bar{x})^2 = \sum x^2 - \frac{(\sum x)^2}{|R|}$$

To split a rectangle *R*, we choose the horizontal or vertical partition into R_1 and R_2 that minimizes the sum of the variability scores of R_1 and R_2. The first two splits of the image of Canada Place are illustrated in the second image below.



To achieve the artistic effect, we prune, or cut off, parts of the binary tree. Two parameters, percent and tolerance, are used to evaluate a subtree's suitability for pruning. To prune a node, we simply remove its left and right subtrees. A node is pruned if at least (greater than or equal to) percent of the leaves in its subtree are within (less than or equal to) tolerance of its average. Distances between colors are computed as the sum, over each color channel, of the pixel value differences, squared.

Part 2: Coding

Problem Specification

Specifications for each function you write are contained in the given code. The list of functions here should serve as a checklist for completing the exercise.

In `class stats`:

- `long getSum(char channel, pair<int,int> ul, pair<int,int> lr)`
- `long getSumSq(char channel, pair<int,int> ul, pair<int,int> lr)`

- `stats(PNG & im)`
- `long getScore(pair<int,int> ul, pair<int,int> lr)`
- `RGBAPixel getAvg(pair<int,int> ul, pair<int,int> lr)`
- `long rectArea(pair<int,int> ul, pair<int,int> lr)`

The `stats` class is used only for constructing the `twoDtree`. The choice of optimal split requires the computation of statistics over each candidate rectangle, and these are simply too expensive to compute in real time, especially given the number of rectangles in our recursively defined tree! Instead, we precompute support structures that allow the necessary statistics to be computed in constant time. We request a particular set of structures (cumulative sums, and cumulative sums of squares, for each color), but we leave it as a puzzle for you to figure out how to use them to get the info you need.

In `class twoDtree`:

- `void clear()`
- `void copy(const twoDtree & other)`
- `twoDtree(PNG & imIn)`
- `PNG render()`
- `void prune(double pct, int tol)`
- `Node * buildTree(stats & s,pair<int,int> ul, pair<int,int> lr)`

The twoDtree class is a simple binary tree class. We've defined the Node class for you, and a few other necessary memory management functions, but everything else is left to you. Since we will be grading both the .h and .cpp files, you are welcome to add helper functions to the class.

Getting the Given Code

The source code and test pictures are attached in the website. Please download it.

Your assignment will be judged by the correction and the organization of your report.

Good luck!