Moosaic: Photomosaic Creation Using K-D Trees

Chase Davis, Gabriela Merz, Dino Rodriguez, Samuel Stone

Demo Video: tinyurl.com/cs51moosaic2015

Demo Images: See *report/* directory, where we have included some of our sample Moosaics!

Overview

We implemented a program that creates a photomosaic of images that recreates a larger base image. When this program is run, with the base bitmap image and a folder of small tile bitmap images as arguments, it generates a new bitmap file with the small tiled images arranged in such a way that their colors correspond to the best matching regions of the larger images. K-D trees are the data structure at the heart of our project, with the nearest neighbor algorithm (which runs in O(logn)) being our primary algorithm.

Instructions

See code/README.

Planning

Below are our annotated final and draft specifications:

Draft Specification:
Moosaic

Brief Overview

In our final project, Moosaic, we plan to create an application that will take in a user's image file and create a photographic mosaic of the user image using a designated image database. This will be composed of hundreds or thousands (it is very hard to find a ton of bitmap images, so that number will likely be closer to hundreds than to thousands) of very small images that act as the pixels of the new user mosaic. Our primary goal for the project is to create an efficient algorithm that implements k-d trees to create the mosaics, by comparing the color of the pixels from the user image to the average color

of each of the potential pictures. Thankfully, we completed this primary goal! Our final program is actually able to do this quickly and efficiently! Our first secondary goal includes being able to compress this output file, as without any compression, it will likely be a huge file because of the many smaller images that compose the larger mosaic. (We realized this feature was probably a waste of time) Another secondary goal is to add additional helpful functionality, like ensuring that the user can use different file types (BMP, TIFF, PNG, JPEG) and also specify that the mosaic output should be a specific file type. (We underestimated how difficult this would be in Camlimages) And our third secondary goal would be implementing a "Where's Waldo" type game, where the Mosaic contains a small cow that will be hidden randomly somewhere that it blends in quite well (based on RGB analysis) and then we will export another image with the cow circled so users can effectively make a game out of it. (We also realized this feature was probably a waste of time)

Feature List

1. [Core] Allow user to input a BMP file to be processed into a mosaic. Done
2. [Core] Matches a BMP with a large library of images to create a mosaic, using k-d trees, and outputs this file Done
3. [Extension] Compresses the output if it is large, sacrificing some image quality but creating a more useable output file Extraneous
4. [Extension] Allows different file types for input and output Extraneous
5. [Extension] Color-grades each individual sub-photo to more closely match the original cell of the source image Too hard in Camlimages
6. [Extension] Where's Waldo style puzzle with a hidden cow in the mosaic Extraneous

Technical Specification

1: Modularization
Our modularization plan was actually quite similar to what we actually wound up implementing, although Camlimages took care of some of the tasks that we planned to do ourselves.

Module 1:
Takes in a photo from a user (core: just a BMP, extension: several filetypes) and saves and stores it. Basically have this – this was simpler than we thought because of camlimages

Module 2:
Divides the photo up into a grid, likely with each square of the grid being several pixels large Definitely have this!

Module 3:
Loops through every cell in the grid and calculate its average color, and then store each cell and the color vector corresponding to it in a kd-tree (hopefully in O (n log n) time). Yes, the core of our algorithm!

Module 4:
Traverses the KD tree in the most efficient way we could find (O(n log n) time, but we may have a way to do it in linear time) and create a new image created by matching the color vectors at each spot in our kd-tree and choosing images from our database to create the mosaic file. Nearest neighbor – we didn't know how hard this module would be – it wound up being probably ~50% of our code, but definitely have this!

Module 5:
Exports the mosaic image as a BMP (core) or a user-chosen filetype (extension). Definitely have this!

The only problems we can foresee with this modularization is that the implementation of two different data structures, like an array and a KD tree, could complicate our program unnecessarily, so we may ultimately attempt to find a way to use only a k-d data structure throughout the program

2: Functions Same as above, basically have these

Module 1:
- import: String filename -> file
  Likely the user will run the program with the filename in the command line and this function will find the file and ensure that it exists. This function should also warn the user if their input file could not be found.

Module 2:
- divide: BMP file -> array of BMP files
  This function will split our image into much smaller tiles so we can have a set of them to work with rather than one huge tile

Module 3:
- find_average_color: BMP files -> average color vector
  For each tile of our large image, this function should make an average color vector of the small tile
- insert: BMP files -> KD tree
  Inserts the color vector and image location into a KD tree that we can later use to create the mosaic

Module 4:
- traverse: KD tree -> BMP file
  Traverses the tree and creates a BMP file of mosaic images using a helper function
- find_mosaic: color vector -> mosaic BMP
  For each color vector, goes through our database of mosaic images and finds the one that best matches the color vector to put into the BMP file

Module 5:
- export: BMP file -> actual file path
  Simply saves the file as a .bmp and tells the user where it is stored in memory

3: Implementing Functionality

Module 1: Simple functionality, we just need to make sure we know how to read in a file name and check for it, but that was part of a CS50 problem set, so we should be able to pretty quickly get that part working. Simple and done

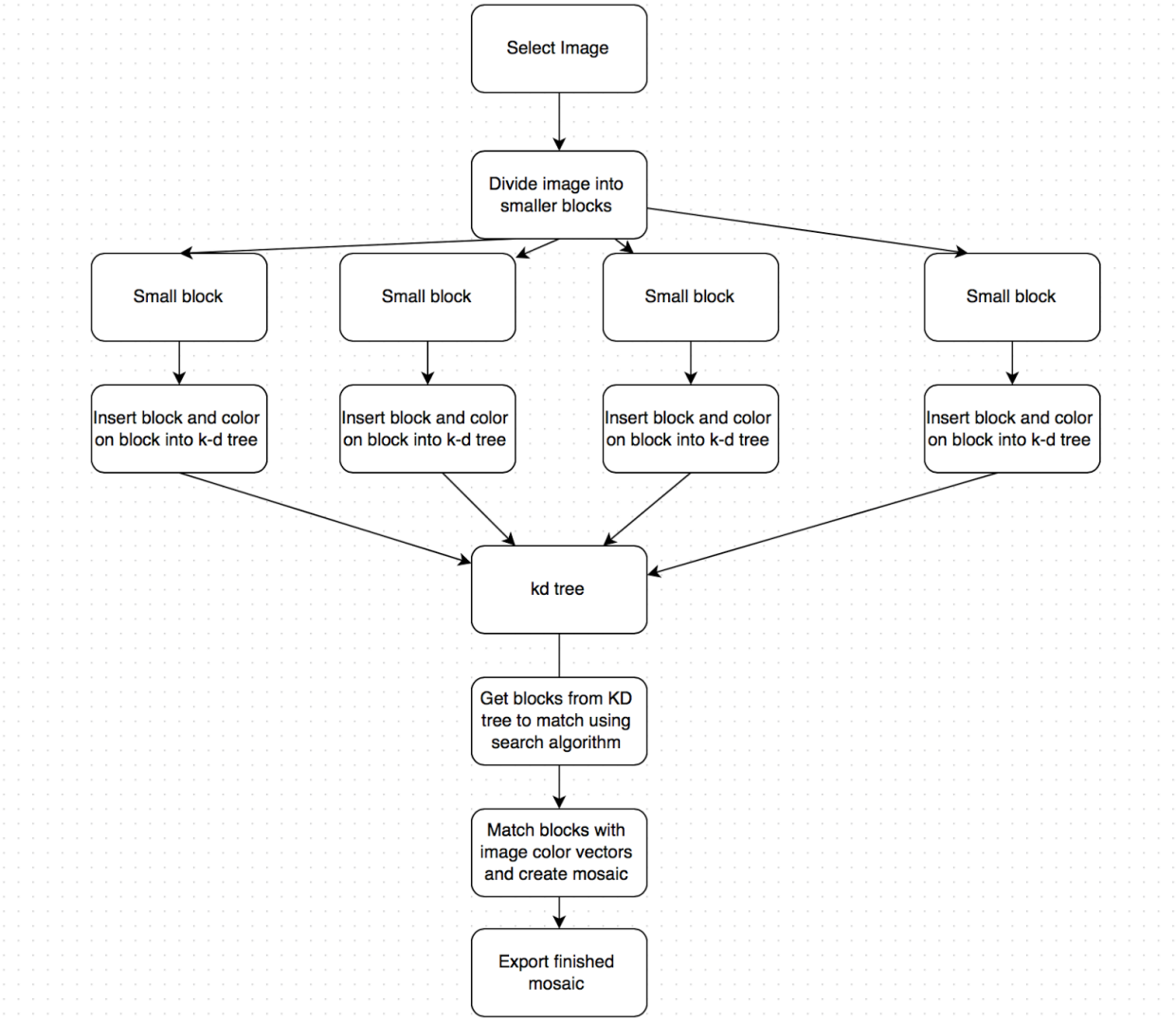
Module 2: Dividing the BMP into an array may prove too time or space inefficient, so we may find another way to combine modules 1 and 2 that just adds smaller parts of the large image to the KD file as it goes, but again, that will be something that becomes clearer once we begin actually implementing.
We basically did the latter idea, combining modules 1 and 2

Module 3: Finding the average color vector will be challenging, but we likely plan to base it on this algorithm: http://ieeexplore.ieee.org.ezp-prod1.hul.harvard.edu/stamp/stamp.jsp?tp=&arnumber=6511349 which allows us to extract RGB vectors from the individual color "squares" in our image
Additionally, we will implement the KD tree by learning the ins-and-outs of k-d trees and finding the most efficient way to implement them. One resource we plan to use is Stanford's k-d tree explanation:
http://web.stanford.edu/class/cs106l/handouts/assignment-3-kdtree.pdf
Was not quite as challenging as we thought, at least compared to actually matching with nearest neighbor.

Module 4: We will traverse the KD tree in an O(n) fashion if we can figure out the implementation, though the O(n log n) would also work in a bind. Finding the most appropriate mosaic images will also be difficult, but we plan to have all of the mosaic sub images and their corresponding color vectors stored in some kind of data structure (perhaps just an array) so it will be quick to match it with the k-d tree. Again, the hardest part of what we are doing is related to finding the most efficient way to utilize the data structures at our disposal.
Our prediction was correct in the last line – it was very hard to figure out how to use this data structure!

Module 5: Finally, outputting the BMP file should be very simple, we just need to make sure our File I/O skills are applied correctly. Simple and done

4: Process flow diagram This is actually what we did!



Next Steps

We plan to implement this project in OCaml because of the necessity for recursive K-dimensional tree traversal. We will also be using git to share code because of our familiarity with it, as well as backing up our code in DropBox for extra security. Problems we foresee including keeping our run-time down for large photos, and being able to completely understand KD trees. Because it could prove difficult for us to manipulate photos and access pixels in Ocaml, we are also considering using C.

The decision of how to manipulate photos was definitely the hardest part of our project, and the constant struggle with camlimages made us want to default back to C several times – we underestimated how much of a problem this would pose.

Final Specification:

Signatures/Interfaces:

Our main tasks are:

1. Creating a module for a KD tree with insert, balance, and searching for nearest_neighbor functionality Completed this succesfully
    a. Average R,G, and B color content will each be a dimension in the tree
    b. Each node in the tree will be list of tuples. The first part of the tuple will be the average color vector for the image, the second part will be the image itself. The reason each node is a list of tuples is to account for the case where two images have the same average color vector.
2. Creating an algorithm for taking in an image file and splitting it into monosized tiles Completed this successfully
3. Writing a function that calculates the average RGB color vector for tiles and base pictures (In order to do this more effectively, we'll be interfacing C and OCaml). This function will work with 24-bit uncompressed BMP files. Wound up not using C, using Camlimages instead, but did this algorithm
4. Writing a function that reconstructs the source image using base images from our database instead of tiles. The color vectors of the base images and tiles should match via NearestNeighbor. Completed this succesfully

None of these will be visible to the user. In the final project, we hope to make it so that the user simply executes a main file with command line arguments listing the starting image, the number of images they want to comprise the resulting moo-saic, and the maximum number of times a base image can be re-used in the making of the moosaic. The output will hopefully display a GIF which flashes back and forth between the old image and the new moosaic. Aside from some difficulties with compiling, we have successfully hidden this information from the use

The signatures for these components are listed below:

```
module type KDTREE =
sig
  exception EmptyTree
  exception NodeNotFound

  (* The type of an element in the tree *)
  type elt

  (* What this type actually looks like is left up to the
   * particular KDTREE implementation (i.e. the struct) *)
  type tree

  (* Returns an empty tree *)
  val empty : tree

  (* Insert elt into tree*)
  val insert : elt -> tree -> tree

  (* Balances a tree so that a nearest neighbor
   * search ban be run on it *)
  val balance: tree -> tree

  (* Search a KD tree for the given value and returns
  * the nearest neighbor *)
  val nearest_neighbor : elt -> tree -> elt

  (* Run invariant checks on the implementation of this KD tree.
   * May raise Assert_failure exception *)
  val run_tests : unit -> unit
end
```

```
module type COMPARABLE =
sig
   type t
   val compare : int -> t -> t -> order
   (*testing will be implemented later*)
end
```

We wound up not implementing the below, because Camlimages was much more effective and had already created methods to do this algorithm

Pseudo-code for C code:

```
#include <stdio.h>
#include <stdlib.h>
#include "bmp.h"
int main(int argc, char* argv[])
{
  // ensure user entered a file name
  // remember filename
  char* file = argv[1];
  // open file
  // read infile's BITMAPFILEHEADER
  // read infile's BITMAPINFOHEADER
  // determine padding for scanlines
  int padding =  (4 - (bi.biWidth * sizeof(RGBTRIPLE)) % 4) % 4;
  //declare color vector
  float color[3] = {0.,0.,0.}
  // iterate over file's scanlines
```

```
  for (int i = 0, i < height t; i++)
  {
    // iterate over pixels in scanline
    for (int j = 0; j < width,  j++)
    {
      // temporary storage
      RGBTRIPLE triple;
      // read RGB triple from infile
      fread(&triple, sizeof(RGBTRIPLE), 1, inptr);
      //update the values in the color vector
      color[0] = triple.rgbtRed + color[0];
      color[1] = triple.rgbtGreen + color[1];
      color[2] = triple.rgbtBlue + color[2];
    }
  }
  //update color values to be averages
  float average = (abs(bi.biHeight)* bi.biWidth);
  color[0] = color[0] / average;
  color[1] = color[1] / average;
  color[2] = color[2] / average;
  // close infile
  fclose(inptr);
  // return the average color vector
  return color;
}
```

Strategy for dividing Source Image into Monosized tiles

1. User argument (M) = minimum number of base images to be used
   L = length of source image (in pixels)
   W = width of source image (in pixels)
   s^2 = number of pixels in monosized tile
2. Solve for s^2
   $L * W >= M(s^2)$
3. Because we know size of tiles, find average color of each tile of the source image and output a list of lists (matrix) where each element is the same as an average color vector of a tile.

Yes, what we did

Strategy for Rebuilding Source Image from Base Images

1. Replace each tile in the list of lists returned in the previous function
2. Reconstruct the source image out of the new list of base images by recursively writing the tiles into a new BMP file

Modules/Actual Code

See attached starting implementation of kdtrees.ml, average_color_vector.c, and what we've started of NearestNeighbor.ml

Timeline

We actually somewhat stuck to this timeline, thankfully!

April 20th

1. Working color vector and picture divisor functions in C or the integration of CamlImages' photo-manipulation library
2. Semi-functional implementation of KD Trees, at least insert implemented and tested in OCaml
3. Start of a function that reassembles pictures in C

April 24th

1. Color vector and picture divisor C functions successfully interfaced with OCaml
2. Fully functional implementation of KD Trees, nearest neighbor function tested and working
3. Working function that swaps and reassembles pictures in C

April 29th

1. Interfacing for swapping and reassembling done
2. Being done except for finishing touches, extensions, and touch-ups, etc.
3. Done with video, other non-technical aspects of project.

May 1st

1. Have Final Project Complete!

Progress Report

As of now, we're starting to get a firm understanding of KD trees, using resources like this handout from Stanford http://web.stanford.edu/class/cs106l/handouts /assignment-3-kdtree.pdf. We've started our implementation of KD trees, having written the basic signature and an insert function that has been tested and works (woohoo!). (See kdtrees.ml lines 103-119 for the function, and lines 224-248 for tests for the insert function). We also have the beginnings of is_balanced and balance functions, which as of now haven't been tested. (See kdtrees.ml lines 121-199). The tests for that are written, but for some reason it's throwing errors for is_balanced, so we're working on that. (See lines 205-219 for the tests). We've also started writing our functions in C that return the average color vector of an image as well as (hopefully) a pointer to the image (see average_color_vector.c). We've started writing a nearest neighbor function, even though we're a little shake on the logic (NearestNeighbor.ml). We're also learning about interfacing C and OCaml. (http://caml.inria.fr/pub/docs/manual-ocaml-4.00/manual033.html) and are working on how we're effectively going to do that. If we decide on CamlImages (http://gallium.inria.fr/camlimages/) this will make our job easier and we will be less dependent on C assuming we have pixel access. We've begun the process of

installing CamlImages to get a better idea of what exactly it can do. Little did we know how long it would take to install it! But still a good choice in hindsight

Version Control

We set up a respoitory on git called moosaic. As of now, the only files in that repo are kdtrees.ml, average_color_vector.c, and NearestNeighbor.ml.

Design & Implementation

Camlimages is a very difficult piece of software to use. Implementing it was honestly terrible – there is no other way to say it. As we discussed numerous times among our group and with our TF, Camlimages is not properly documented, and its functions are often written with names like "x y z a b c". Furthermore, there are almost no type signatures throughout its functions, and the code is hardly commented at all. Figuring out what Camlimages is designed to do probably takes more time than the authors spent writing it. Nonetheless, we were finally able to succeed with using it, but not before debating switching to C, Python, or other languages many times throughout the process.

Furthermore, actually installing Camlimages is very difficult because of the lack of a good readme with it. There were a lot of permissions issues, so as we explained to our TF, we even had to blanket chmod –R 777 files in order to get it to work. Although the library was very useful once we were able to learn how to use it and get it up and running, it was a very burdensome program to install and prepare to use.

Our implementation of nearest neighbor was also challenging. It was very difficult to think about the 3-d nature of k-d trees at times, but ultimately what helped us a great deal was visualizing it. When we thought of splitting distances along a hyperplane as a kind of 3d cognate to binary search on a 2d data structure, we were able to understand better what we needed to do. Nonetheless, it takes a lot of time to pin down the logic of such a search algorithm, and being able to find the most accurate color quickly and efficiently was definitely the most challenging part of our implementation, but also the most rewarding to see completed.

Figuring out how we were supposed to compile the programs we wrote was also terrible, because camlimages requires omake, which we aren't familiar with. We also couldn't open camlimages because of the nature of its incredibly forced installation. So we had to run everything from the top-level, #requiring "camlimages" every time. In order to get everything to work nicely for the user, we had to put all of our modules in one file, which can be compiled in emacs. We know it's technically bad style, but we felt it was more important that the user be able to

see our project work. The modules are available individually to be perused at your leisure.

Reflection

We were pleasantly surprised by how much we learned about abstract computer science concepts throughout development of the project. We initially thought most of our work would be spent on photo analysis and mosaic parts of the project, but the largest component was organizing the abstractions for our KD tree. We were less pleasantly surprised by how difficult it was to actually work through implementation of the tree – it took several team meetings and some assistance from our TF to ultimately figure it out.

One choice we made that worked well was deciding to use Camlimages for image processing, rather than manually coding our image processing functions or re-implementing code we had designed for CS50's problem set that dealt with bitmap images. Although we had a steep learning curve with Camlimages, and had a great amount of difficulty installing it on all of our virtual machines, not to mention learning how it worked, the decision to use it ultimately it paid off. Rather than reinventing the wheel (albeit a complicated one), we learned how to use existing functionality to optimize our time spent on the project.

One choice we made that did not pay off was implementing our entire k-d tree before determining how exactly we would need to use it. For example, we created a function that balanced a k-d tree that took hours and was very difficult throughout – when we finished, we realized because of the way we inserted images into our k-d tree, and the fact that we did not need to update it midway through the program, we did not even need a balance function; our tree would always be balanced anyway! Ultimately, our takeaway from this experience was that it really makes sense to plan your algorithms completely before writing any significant amount of code.

If we had more time, we would have implemented a "Where's Waldo" style game where we randomly used one image of a cow, as described in an earlier proposal, but ultimately decided that it was a waste of our time compared to improving our KD tree algorithm and ensuring that our core functionality generated the most aesthetically pleasing and accurate mosaics possible.

If we were to redo this project from scratch, we would likely have allowed for dynamic image resizing during the mosaic-creation process. Essentially, we would have created a data structure and algorithms that would allow us to easily create a mosaic that incorporated smaller images of different widths and heights and allowed for dynamic resizing to enable us to create final mosaics with most accurate color matching properties.

Advice for Future Students

At the end of the day, we believe that anyone planning a project like this would be wise to heed the following advice:

First, plan out your modules well in advance! As described above, we spent time implementing functions for k-d trees that we did not even need to use in the end, and we could have saved a lot of time that we could have spent more efficiently if we had done a little bit better planning in advance.

Second, always try to avoid reinventing the wheel. Although we definitely had our fair share of struggles with the Camlimages libraries, it did spare us from having to rewrite image manipulation functions in Ocaml or C that would have been an inefficient use of our time.

Finally, make sure you think about the higher-level aspects of your projects before the nitty-gritty of implementing specifics for the program. We spent a lot of time refining our ideas about k-d trees, which ultimately wound up being the most important and interesting part of our project; we could have spent a lot of time learning about image manipulation and done more “cool tricks” with regards to that part of the project, but ultimately would not have learned about such an interesting data structure, and our project would probably not have been as successful as it was.