Pixels to circles

INF205 programming project

December 05, 2022

Group 25

Mohammed Idris Omar

Sigmund Grimsrud

Lavanyan Rathy

Introduction

This project is our final exam in the course INF205 - Resource-efficient programming (autumn 2022) at Norwegian University of Life Sciences NMBU.

Our task is to have a two-dimensional image given by an n x n square of black and white pixels as input, and create an approximation of this landscape with superimposed circular disks. The tasks aims on best possible accuracy for a given maximum number of circular disks.
We want to create a lossless and a lossy representation of the image using unlimited number of circular disks.

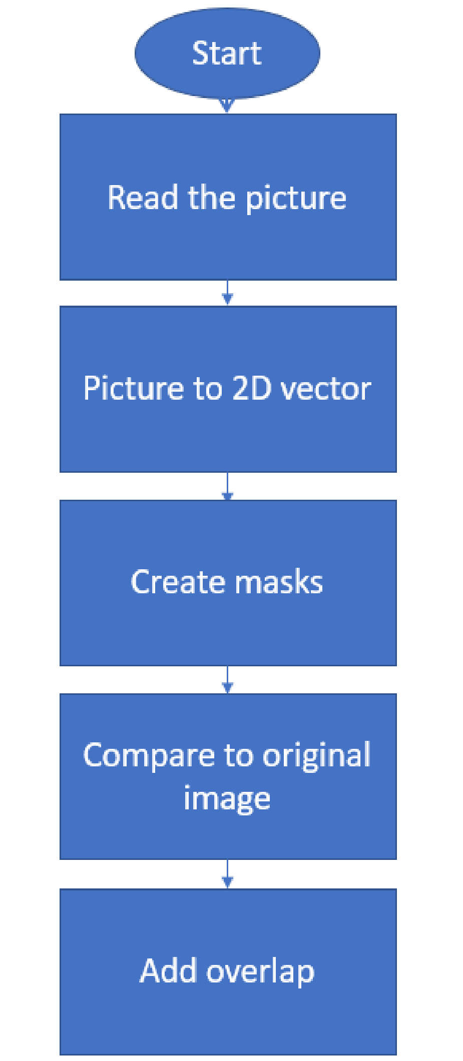

Figure 1: Flowchart at the beginning of the project

This is how we wanted the process to go. The process starts with reading the picture. Then it converts it to a 2D vector, and creates masks. We want to compare with the original image and lastly add the overlaps.
After trying several methods, this is what we landed on. We started a little late, but came to a solution.

Data structures

We want to write our code with a focus on object-oriented programming(OOP). The reason we want to write in OOP because it's faster and easier to execute. It also creates a clear structure for the program. It will be easy to debug and edit the code.

Lossy representation

Lossy representation reduces a file, and permanently eliminate some information.

Lossless representation

With a lossless representation the data from the original file remains as it was, and the information.

We wanted to try both lossy and lossless representation with our solutions.

Algorithms/performance

In this part we want to explain the chosen algorithm, and explain why we didn't go with our previous algorithms.

Random circle search

This is the algorithm we went with. This is a recursive algorithm, which means that it starts with a big radius, and shrink until it finds a circle. It searches for circles with a specific radius. This worked good for us, since we didn't have one type of circles, but different types of circles.

Depth-First Search(DFS) algorithm

We have a 2D matrix, and we want to detect the edges. DFS is a recursive algorithm. Which means that it shows the input value as a "smaller" value. This algorithm uses backtracking principle. It searches all the neighbour nodes.
We didn't go with this algorithm because we wanted to find more circles, and not only find the edges. This algorithm would have been good if we just had one stack of circles, but since it is a overlapping image we couldn't proceed with this solution.

Edge detection

Edge detection is an image processing technique we learned in our course in image processing (INF250) for finding the boundaries of objects within images. The reason for not using this is the same as for DFS.

Concurrency

In this part we want to explore the opportunity with concurrency in our project.

We want to work with OpenMP and see the possibility to parallize.
Every core has access, and does not need MPI functions
The reason for that is because of efficiency, which can save time.

Picture to matrix lossy

Our idea was to turn the picture to a matrix

Figure 2: Benchmark example

This is to create a grid of values so that we can see the borders of the circles.

readlossy.cpp

First we wanted to include a header file

```cpp

#include <cassert>
#include <iostream>
#include <vector>
#include <iterator>
#include <fstream>  // Used in make_vect std::ifstream
#include <chrono>
#include <cmath>

#include <omp.h>

#include "read_omp.h"

```

Reading the image, and construct a 2d vector.

```cpp
void VectorArray::make_vect(std::string filename) {
    /*
    * Based on code by Martin Thomas Horsch
    */

    // Read from start of file
    std::ifstream file(filename);
    file >> this->x_size >> this->y_size;

    // construct 2D vector where the outer vector has y elements, each
    of which is a vector with x elements
    std::vector<std::vector<int>> figure(this->y_size, std::vector<int>
    (this->x_size));
    this->image = figure;

    // attention! there is a line break character that we need to get rid of
    char pixel;
    file.get(pixel);

    // Reads in from file
    for (int j = 0; j < this->y_size; j++)
        for (int i = 0; i < this->x_size; i++) {
            assert(!file.eof());
            file.get(pixel);
            if (pixel) this->image[j][i] = this->foreground_color;
            else this->image[j][i] = this->background_color;
        }
    file.close();
} // make_vect
```

Returns true if a circle with radius r fits on cell x, y

```cpp
bool VectorArray::overlap(int x, int y, int r) {
    /*
     * Returns true if a circle with radius r fits on cell x, y
     */

    // Limiting search ranges
    int x_min = -r;
    int x_max = r;
    int y_min = -r;
    int y_max = r;

    if (x <= r) x_min = -x;
    if ((this->x_size - 1 - x) <= r) x_max = this->x_size - x - 1;
    if (y <= r) y_min = -y;
    if ((this->y_size - 1 - y) <= r) y_max = this->y_size - y - 1;

    int tiles = 0; // Number of tiles in the circle
    int matches = 0; // Number of lit up tiles
    // Checking neighboring cells
    for (int j = y_min; j <= y_max; j++) {
        for (int i = x_min; i <= x_max; i++) {
            // Calculates if the cell is inside a circle with radius r
            float dd = (x+i - x) * (x+i - x) + (y+j - y) * (y+j - y);
            if (r * r >= dd) {
                tiles++;
                // Checks color of that cell
                if(this->image[y+j][x+i] == this->foreground_color) matches++;
            }
        }
    }
    // Checks if a percentage of the circle tiles are lit up
    if (matches/tiles > 0.9) return true;
    else return false;
}

```

Creates a map of the max radius for a circle that fits in each cell

```cpp
void VectorArray::compress() {
    /*
     * Creates a map of the max radius for a circle that fits in each cell
     */

    // Creates a map of size y_size, x_size
    std::vector<std::vector<int>> figure(this->y_size, std::vector<int>
    (this->x_size));
    this->approximation = figure;

    // Divides the work load based on number of threads
    this->split = 1 + (this->y_size - 1) / this->num_threads;
    // Tests which radius fits in each cell
    for (int y = omp_get_thread_num() * this->split; y <
    (omp_get_thread_num() + 1) * this->split; y++) {
        if (y < this->y_size) {
            for (int x = 0; x < this->x_size; x++) {
                int r = 1;
                while (VectorArray::overlap(x, y, r)) {
                    r++;
                }
                this->approximation[y][x] = r - 1;
            }
        } // if
    }
} // Compress()
```

Remove unnecessary points from the approximation map

```
void VectorArray::Clean_Approx(){
    /*
     * Remove unnecessary points from the approximation map
     */

    int r=2; // Radius around each cell we compare against

    // Iterates through each cell in the map, sliced for each rank
    for (int y = this->split*omp_get_thread_num(); y < this->split*
    (omp_get_thread_num()+1); y++) {
        if (y < this->y_size) {
            for (int x = 0; x < this->x_size; x++) {
                // Checks each neighbor in a square
                for (int j = -r; j <= r; j++) {
                    for (int i = -r; i <= r; i++) {
                        // Avoids checking own cell
                        if ((j == 0) && (i == 0)) continue;
                        else{
                            // Avoids stepping outside the array
                            if (y + j >= 0)
                                if (x + i >= 0)
                                    if (y + j < this->y_size)
                                        if (x + i < this->x_size) {
                                            // Compares current cell
                                            with its neighbors
                                            if (this->approximation[y]
                                            [x] <= this->approximation[y
                                            + j][x + i]) {
                                                this->approximation[y]
                                                [x] = this-
                                                >background_color;
                                            }
                        }} // if if
                    }} // For j and i
            } // For x
        } // if y < y_size
    } // For y
}

```

```
void VectorArray::PrintOut(std::ostream* target){
    /*
     * Prints out a vector representation of disks approximating the image
     */

    // Prints out:
    // x y
    // background color
    *target << this->x_size << " " << this->y_size << '\n' << this-
    >background_color << "\n\n";

    // Creates a large vector array to hold the vector representation of
    // the disks
    #pragma omp parallel
    {
        std::vector<std::vector<int>> figure(0, std::vector<int>(4));

            for (int y = this->split * omp_get_thread_num(); y <
            (omp_get_thread_num() + 1) * this->split; y++) {
                if (y < this->y_size){
                    for (int x = 0; x < this->x_size; x++) {
                        // Checks a list of radii, hens !=bg_color
                        if (this->approximation[y][x] != this-
                        >background_color) {
                            // Adds a disk vector to its figure, Disk
                            {x_coordinate, y_coordinate, radius, color}
                            figure.push_back({x, y, this-
                            >approximation[y][x], this-
                            >foreground_color});
                            #pragma omp atomic
                            this->num_disks++;
                        }
                    }
                }
            }
        // Collects all Disk vectors to a single vector
        for (std::vector<int> v : figure)
        {
            #pragma omp critical
            this->disks.push_back(v);
        }
    } // omp parallel

    // Prints out number of disks
    *target << this->num_disks <<'\n';


    // Iterates through each disk and prints out its contents
    int work_shared = 1 + (this->num_disks - 1)/this->num_threads;

    #pragma omp parallel
    {
        std::string string_out;
        for (int i = work_shared * omp_get_thread_num(); i <
        (omp_get_thread_num() + 1) * work_shared; i++) {
            if (i < this->num_disks) {
                string_out = "";
                for (int element : this->disks[i]) {
                    string_out += std::to_string(element) + '\t';
                }
                #pragma omp critical
                *target << string_out << '\n';
            }
        }

    } // omp parallel
}

```

A single function to convert a pixelmap to a vector representation of disks

```
void VectorArray::vectorize(std::string input_filename, std::string output_filename)
    /*
     * A single function to convert a pixelmap to a vector representation of disks
     */

    auto t0 = std::chrono::high_resolution_clock::now();
    VectorArray::make_vect(input_filename);

    auto t_make_vect = std::chrono::high_resolution_clock::now();
    std::chrono::time_point<std::chrono::system_clock,
    std::chrono::duration<long, std::ratio<1, 1000000000>>>t_compress;

    #pragma omp parallel
    {
    VectorArray::compress();

    #pragma omp barrier
    #pragma omp single
        {
            t_compress = std::chrono::high_resolution_clock::now();
        }
    VectorArray::Clean_Approx();
```

```cpp
    #pragma omp barrier
    }
    auto t_clean = std::chrono::high_resolution_clock::now();
    std::ofstream output(output_filename);
    VectorArray::PrintOut(&output);
    output.close();

    auto t1 = std::chrono::high_resolution_clock::now();


    auto time_make_vect = std::chrono::duration_cast<std::chrono::nanoseconds>
    (t_make_vect-t0).count()/std::pow(10,6);
    auto time_compress = std::chrono::duration_cast<std::chrono::nanoseconds>
    (t_compress - t_make_vect).count()/std::pow(10,6);
    auto time_approx = std::chrono::duration_cast<std::chrono::nanoseconds>
    (t_clean - t_compress).count()/std::pow(10,6);
    auto time_printout = std::chrono::duration_cast<std::chrono::nanoseconds>
    (t1 - t_clean).count()/std::pow(10,6);
    std::cout << "Make vect = " << time_make_vect << " ms\n";
    std::cout << "Compress  = " << time_compress << " ms\n";
    std::cout << "Approx    = " << time_approx << " ms\n";
    std::cout << "Print out = " << time_printout << " ms\n";

}
```

Picture to matrix lossless

Same concept, but it's lossless

readlossless.cpp

```cpp
#include <cassert>
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <iterator>

#include "read.h"

std::vector<std::vector<int>> get_vect(std::string filename) {
    // Read from image
    std::ifstream file(filename);
    int x; // Size in x dim
    int y; // Size in y dim
    file >> x >> y;

    std::cout << x << "\t" << y << std::endl;

    // construct 2D vector where the outer vector has x elements,
    // each of which is a vector with y elements
    std::vector<std::vector<int>> figure(x, std::vector<int>(y));

    // attention! there is a line break character that we need to get rid of
    char pixel;
    file.get(pixel);

    // read in file into the vector such that vector[x][y]
    // contains the value of pixel (x, y) as 0 or 1
    // note that the file format contains the pixels in (y, x) ascending order

    for(int j = 0; j < y; j++)
      for(int i = 0; i < x; i++)
      {
        assert(!file.eof());
        file.get(pixel);
        if(pixel) figure[j][i] = 1;
        else figure[j][i] = 0;
      }

   file.close();
   return figure;
} // get_vect
```