Graph Based Image Processing and Combinatorial Optimization –Mandatory Tasks

December 3, 2021

1 General instructions

This document contains the mandatory tasks for the course "Graph Based Image Processing and Combinatorial Optimization". To pass the course, you should solve all the tasks and hand in your solutions to Filip Malmberg.

You can do these tasks on your own, or form groups of up to 3 persons working together. Some of the tasks are theoretical questions, while others involve some programming.

1.1 Example code

For some of the programming related tasks, I have prepared some example code written in C++. These code examples have been tested with Visual C++ on Windows, but you are of course free to use any compiler/IDE/OS. You are also free to use any other language/programming environment that you feel more comfortable in, but even in that case I hope the code examples can still be a useful starting point.

For reading and writing images, the C++ examples use *CImg*, a small and free library for working with image data in C++. Documentation for this library can be found at `http://cimg.eu/`. This library is included with the source code for the exercises.

2 Task 1: Local search and Iterated conditional modes (ICM)

This task requires Lecture 1.

2.1 Part1

In the first lecture we talked about the Iterated Conditional Modes (ICM) algorithm, which is a variant of optimization based on local search for pixel (vertex) labeling problems. Starting from an arbitrary labeling, ICM iteratively tries to improve the current solution by changing the label of one pixel at a time.

Your first task is to implement the ICM algorithm for solving the pixel labeling problems with the objective function given on page 40 of the lecture notes combinatorial optimization from the first lecture:

$$f = \sum_{v \in V} \Phi(v) + \alpha |\partial \mathcal{L}| \; . \tag{1}$$

Use your implementation to segment an image of your choice (for example the image from the thresholding example in the first lecture). Select suitable values for α and the threshold t.

2.2 Part 2

A general algorithm for local search can be formulated as follows:

- Start at an arbitrary solution.
- While the current solution is not a local minimum, replace it with an adjacent solution for which f is lower.

Give a proof that this algorithm is guaranteed to find a locally optimal solution in a finite number of iterations. Hint: See slide 30 in “Notes 2” from the first lecture.

3 Task 2: Semi-supervised learning on graphs with minimum cost paths and minimum spanning trees

This task requires Lecture 1-2.

3.1 Part 1: Dijkstra’s algorithm

Implement Dijkstras algorithm to compute, for all pixels in an image, the cost of the shortest path between the pixel and a user specified set of seeds.

The `DijkstraTask` directory contains the following source files:

- `main.cpp`: Contains the function `main(...)`
- `Dijkstra.cpp`: Contains the function `Dijkstra(...)`. You will write most of your code in this file.
- `Dijkstra.h`: Header file for `Dijkstra.cpp`
- `QueueElement.h`: We will use the priority queue implementation available in the C++ standard library. This header defines a class of *queue elements* to populate the priority queue. A queue element represents a path throught the graph, and contains information about the coordinate of the pixel representing the endpoint of the path, and the cost of the path.

The main function loads two images, specified by the command line arguments, into memory. The first image is the image used to define the graph in which we compute the shortest paths. The second image (which must have the same dimensions as the first one) is used to define the seedpoints. Every pixel in the second image whose intensity is greater than zero is taken to be a seed. Then, the main function calls the `Dijkstra()` function to compute length of the shortest paths from the seeds to all other pixels. The resulting distance values are then normalized to the range $0 - 255$ and written to the file `result.bmp`.

Your task is to complete the implementation of Dijkstras algorithm in the file `Dijkstra.cpp`. Assume that every pixel is adjacent to its four horizontal and vertical neighbors. We will not store the graph edges and their weight explicitly. Instead, we will calculate them implicitly during the execution of Dijkstras algorithm. First, let the weight of each edge be 1. Once you have completed the implementation, try to change the weight of each edge to be the absolute difference in intensity between the pixels spanned by the edge.

3.2 Part 2: Path cost functions

As shown in the paper by Falcão et al., Dijkstra's algorithm can be used to compute optimum path forests for a number of different path cost functions. There are, however, path cost functions for which the algorithm does not return the correct result. One such example is the following path cost function:

$$\sum_{v_i \in \pi} |I(v_i) - I(v_0)| \,, \tag{2}$$

where π is a path traversing vertices $v_0, v_1, \ldots$, and $I(v_i)$ is an intensity value associated with the vertex v_i. In other words, the path cost function sums, for all vertices along the path, the difference in intensity between that vertex and the first vertex of the path. Your task is to prove that Dijkstra's algorithm is not guaranteed to work with this path cost function by finding a counterexample, i.e. a graph with vertex intensities and a set of seeds for which the algorithm returns the wrong result. A hint is that at least two seedpoints are required. The graph also does not have to be that large – there exists valid counterexamples that have only four vertices and three edges.

3.3 Part 3: Minimum spanning trees

Implement Prim's algorithm to compute a *segmentation by minimum spanning forests* relative to a set of labeled seeds. When completed, your program should output an image where every pixel is assigned the same label as the seed to which it is connected on a minimum spanning forest relative to the seeds.

The `MSTTask` directory contains the following source files:

- `main.cpp`: Contains the function `main(...)`
- `MSF.cpp`: Contains the function `MSF(...)`. You will write most of your code in this file.
- `MSF.h`: Header file for `MSF.cpp`
- `QueueElement.h`: We will use the priority queue implementation available in the C++ standard library. This header defines a class of *queue elements* to populate the priority queue. A queue element represents an edge to be added to the forest, and contains information about the coordinate of the pixel representing the endpoint of the edge, the weight of the edge, and the label of the starting point of the edge.

The main function loads three images, specified by the command line arguments, into memory. All images must have the same dimensions.The first image is the image used to define the graph in which we compute the MSF. The second image is used to define the seedpoints. Every pixel in the second image whose intensity is greater than zero is taken to be a seed. The label of the each seed is read from the corresponding pixel in the third image. Then, the main function calls the `MSF()` function to compute an MSF segmentation. The resulting labeled image is written to the file `result.bmp`.

Your task is to complete the implementation of Prim's algorithm in the file `MSF.cpp`. Assume that every pixel is adjacent to its four horizontal and vertical neighbors. We will not store the graph edges and their weights explicitly. Instead, we will calculate them implicitly during the execution of

Prim's algorithm. Let the weight of each edge to be the absolute difference in intensity between the pixels spanned by the edge. (If you want to, you can also try other edge weights!)

4 Task 3: Optimization of submodular energy functions with minimal graph cuts

In this task, you will use minimal graph cuts and move-making (α-expansion) to solve an optimization problem. For this, I suggest that you use the gco-v3.0 library (`http://vision.csd.uwo.ca/code/`), which implements the optimization algorithm described in the paper "Fast Approximate Energy Minimization via Graph Cuts", by Boykov, Veksler and Zabih (PAMI 2001). The library is written in C++, and also comes with a MATLAB wrapper. If you can find similar libraries/wrappers for other languages, you are free to use those instead.

The optimization problem you should solve is an image filtering one. Given a grayscale image with integer valued intensities, find a new image I' the (locally) minimizes:

$$\sum_v (I'(v) - I(v))^2 + \lambda \sum_{v,w \in E} (I'(v) - I'(w))^2 \tag{3}$$

where V is the set of pixels in the image, $I(v)$ is the instensity of image I at pixel v, and E is the set of all pairs of 4-adjacent pixels. The real valued constant λ controls the degree of smoothing.

Instead of performing α-expansion with the set of all possible intensities as our label set, we will consider two types of moves: "increase intensity by 1 for a subset of pixels" and "decrease intensity by 1 for a subset of pixels". For both types of moves, the problem of finding the best possible move is a binary labeling problem ("should this pixel increase/decrease or not") that can be solved via minimal graph cuts. Your task has two parts:

- Implement an iterative move-making procedure for optimizing the objective function given above, using the two types of moves. For both types of moves, the best possible solution in the move set can be found by solving a minimal graph cut problem.
- Prove that in each step, the binary labeling problem to be solved by graph cuts is submodular. As a starting point, you may look at the appendix of the paper "Fast graph-cut based optimization for practical dense deformable registration of volume images" where a similar proof is given.