České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačové grafiky a interakce

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Lenka Saidlová**

Studijní program: Otevřená informatika
Obor: Počítačová grafika a interakce

Název tématu: **Optimalizace založená na hledání minimálního řezu v grafech**

Pokyny pro vypracování:

Prostudujte algoritmy pro aproximativní optimalizaci energetických funkcí využívající výpočet minimálního řezu v grafech [1,2,3]. Vybraný algoritmus implementujte v jazyce C++, přičemž pro vlastní výpočet minimálních řezů využijte knihovnu GridCut [4]. K implementovanému algoritmu vytvořte jednoduché programátorské rozhraní a dokumentaci doplněnou sadou ukázkových úloh, které poslouží jako příklad použití.

Seznam odborné literatury:

[1] Kolmogorov & Zabih: What Energy Functions Can Be Minimized via Graph Cuts?, IEEE Transactions on Pattern Analysis and Machine Intelligence 26(2):147-159, 2004.
[2] Schmidt & Alahari: Generalized Fast Approximate Energy Minimization via Graph Cuts: alpha-expansion beta-shrink Moves, arXiv:1108.5710, 2011.
[3] Woodford et al.: Contraction Moves for Geometric Model Fitting, Proceedings of the European Conference on Computer Vision, vol. 3, pp. 181-194, 2012.
[4] Jamriška et al.: Cache-efficient Graph Cuts on Structured Grids, Proceedings of IEEE Conference on Computer Vision and Pattern Recognition, pp. 3673-3680, 2012. http://gridcut.com

Vedoucí: Ing. Daniel Sýkora, Ph.D.

Platnost zadání: do konce letního semestru 2015/2016

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická
-4-

L.S.

V Praze dne 31. 10. 2014

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Master's thesis

Graph cut based optimization

Bc. Lenka Saidlová

Supervisor: doc. Ing. Daniel Sýkora, Ph.D.

Study Programme: Open informatics

Field of Study: Computer graphics

January 5, 2015

Acknowledgements

I would like to thank to my supervisor Daniel Sýkora as well as opponent of my thesis Ondřej Jamriška, for consultations and providing valuable advice and comments.

Declaration

Abstract

Many problems of early vision and computer graphics can be formulated in terms of energy minimization. In this thesis, we focus on improvement of an algorithm, that solves the energy minimization for two labels via graph cuts and whose implementation is avaliable as GridCut library. We extend the GridCut by expansion move algorithm, which solves the energy minimization for multiple labels, and apply it to 2D grid-like graphs with 4 and 8 connected neighboring system and 3D grid-like graphs with 6 and 26 connected neighboring system. We compare the resulting implementation with the GCO library.

Abstrakt

Mnoho problémů počítačového vidění a grafiky lze formulovat jako problém minimalizace energie. V této práci se zaměřujeme na vylepšení algoritmu, který řeší minimalizaci energie pro dvě značky přes řezy v grafu a jeho implementace je k dispozici jako knihovna GridCut. Tuto knihovnu rozšiřujeme "expansion move" algoritmem, který řeší minimalizaci energie pro více než dvě značky a aplikujeme ho na 2D mřížkové grafy s 4 a 8 spojitým okolím a na 3D mřížkové grafy s 6 a 26 spojitým okolím. Výslednou implementaci porovnáváme s knihovnou GCO.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many problems from early vision and computer graphics can be formulated in terms of energy minimization. These problems include image segmentation, coloring and restoration, photomontage, motion and stereo and others.

Algorithms solving energy minimization for such problems already exist. In this thesis, we focus on improvement of an algorithm that is based on graph cuts and its implementation is available as GridCut [1].

GridCut implements the Boykov algorithm presented in [8], which is the state of the art solution to max-flow/min cut problem for grid-like graphs. Boykov algorithm was further extended in GridCut by two improvements. The first improvement is the parallelization of the algorithm according to [19] and the second improvement is the effective use of cache memory presented in [14]. Thanks to these new enhancements, GridCut promises to be the fastest solver of max-flow/min-cut problem for grid-like graphs.

Unlike the Boykov algorithm, which is directly linked to the GCO library [4] solving energy minimization for multiple labels, there is no example of how to use GridCut for that. Therefore, the goal of the thesis is to extend GridCut to enable its native using for solving energy minimization for multiple labels. We expect that the extension will be the fastest implementation of the state of the art of the problem ever.

For the implementation of the extension, we chose the expansion move algorithm, shortly called α-expansion, which is based on iterative solving max-flow/min-cut problem. The algorithm was first presented in [10] and the exact graph construction was improved and simplified in later version in [17].

We implement the α-expansion for 2D grids with 4 connected neighboring system as it is in the GCO library and for each max-flow/min-cut solving, we call the GridCut library. Moreover, we also apply the algorithm for other graph topologies, which are 2D grids with 8 connected neighboring system and 3D grids with 6 and 26 connected neighboring system.

Finally, we compare the resulting implementation to the GCO library and analyze the results.

Chapter 2

Energy Minimization in Early Vision

The problems of early vision and computer graphics mentioned in Introduction can be formulated as minimization of a certain type of energy function. To understand the problem of solving energy minimization, we generalize all the problems to a common problem of a pixel-labeling without loss of generality.

An input of a pixel-labeling problem is a set of n pixels $P = \{p_1, p_2, ..., p_n\}$ and a set of k labels $L = \{l_1, l_2, ..., l_k\}$. The set of labels can either represent dispartities (for stereo) or it can substitute pixel intensities (image restoration or segmentation).

The goal of the pixel-labeling problem is to find a labeling f which assigns a label $f_p \in L$ for every pixel $p \in P$ and, most importantly, the labeling f has to minimize the given energy function.

2.1 Energy Function

In order to be able to solve the pixel-labeling problem, we need to restrict the energy function to a certain form with several required features. A form of the energy function, which we take from [10], is

$$E(f) = E_{data}(f) + E_{smooth}(f), \tag{2.1}$$

where $E(f)$ is an energy function of labeling f, term $E_{data}(f)$ is a constraint provided by the data and term $E_{smooth}(f)$ is a smoothness constraint on the labelings.

Specifically, the term $E_{data}(f)$ is a data energy function dependent on the observed data and measures a cost of assigning the label f_p to the pixel p. A form of the data term is

$$E_{data}(f) = \sum_{p \in P} D_p(f_p).$$

The only restriction on E_{data} term is that it has to be nonnegative.

The term $E_{smooth}(f)$ is an energy function that measures a cost of assigning labels f_p, f_q to the pair of interacting pixels p, q and represents spatial smoothness. A form of the smoothness term is

$$E_{smooth}(f) = \sum_{\{p,q\} \in N} V_{p,q}(f_p, f_q),$$

where N is a set of interacting pairs of pixels. The set N usually consists of adjacent pixels, but it can be also arbitrary.

When the general form of the energy function 2.1 is rewritten according to the stated terms, we get the resulting form of the energy function

$$E(f) = \sum_{p \in P} D_p(f_p) + \sum_{\{p,q\} \in N} V_{p,q}(f_p, f_q).$$

This energy function is typically nonconvex, which means it has many local minima. Thus the major difficulty with energy minimization lies in the enormous computational costs. Moreover, the space of possible labelings has dimension P, which is many thousands.

2.2 Discontinuity Preserving Interaction Penalty

The choice of the smoothness term $V_{p,q}$ is a principal issue, which the final labeling f relies on. For pixel-labeling problem, we require the smoothness term to be discontinuity preserving, where the discontinuity is considered as the sharp change of labels. Smoothness term must not penalize these labeling changes and thus we also need to take interacting pixels of each pixel $p \in P$ into consideration.

We show the example of discontinuity in figure 2.1. The discontinuity is here at the borders of the objects, where the adjacent pixels has different labels.

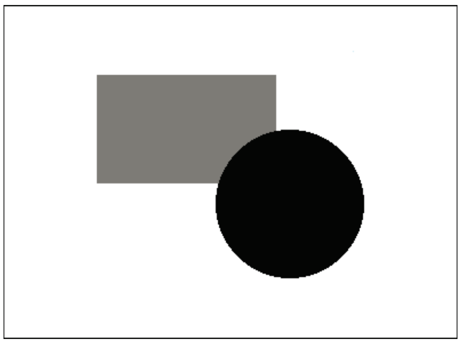

Figure 2.1: An example of discontinuity at the objects borders.

In this thesis, we focus on the general class of semimetric and metric interaction penalty $V_{p,q}$, which both satisfy the condition of preserving the discontinuity.

We call $V_{p,q}$ a semimetric interaction potential on the space of labels L, if for any pair of labels $f_p, f_q \in L$ it satisfies these following properties from [24]

$$V_{p,q}(f_p, f_q) = V_{p,q}(f_q, f_p) \geq 0,$$

$$V_{p,q}(f_p, f_q) = 0 \Leftrightarrow f_p = f_q.$$

If we restrict the assumptions above by

$$V_{p,q}(f_p, f_q) > 0 \Leftrightarrow f_p \neq f_q$$

and add the assumption that for any pair of labels $f_p, f_q, f_r \in L$ it satisfies the triangle inequality

$$V_{p,q}(f_p, f_q) \leq V_{p,q}(f_p, f_r) + V_{p,q}(f_r, f_q),$$

we call $V_{p,q}$ a metric interaction potential on the space of labels L.

A term $V_{p,q}$ satisfying these properties should also bound the largest possible penalty. This avoids overpenalizing sharp jumps between the labels of neighboring pixels.

Now we give particular examples of the semimetric and metric smoothness terms to undestand how the resulting labeling f differs acording to the used interaction penalty.
An example of the semimetric interaction penalty includes the truncated quadratic distance

$$V_{p,q}(f_p, f_q) = min(K, (|f_p - f_q|)^2),$$

where K is a constant bounding the penalty. A demonstration of how the truncated quadratic distance varies over the labels and the possible resulting labeling f is in figure 2.2.

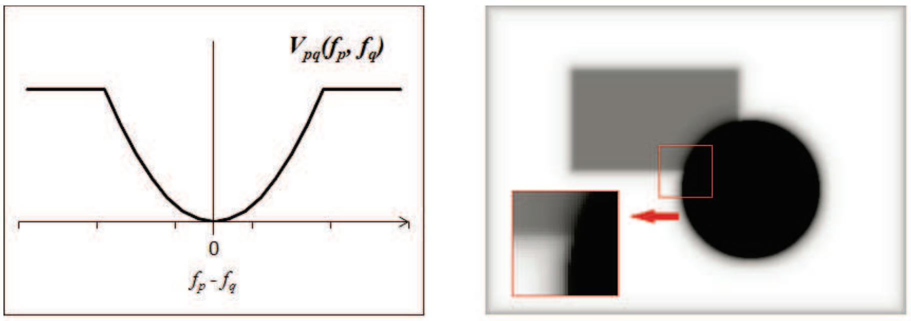

Figure 2.2: A graph of $V_{p,q}$ for truncated quadratic distance on the left and the possible resulting labeling on the right.

For the metric interaction penalty, the truncated absolute distance

$$V_{p,q}(f_p, f_q) = min(K, |f_p - f_q|)$$

can be used. A graph of how the truncated absolute distance varies over the labels and the possible resulting labeling f of the same input as before is in figure 2.3.

As we can see from the figures, these models lead to labelings which consist of regions where the pixels in the same region have similar labels and, therefore, we call them piecewise smoothness models.

Another important metric discontinuity preserving function is the Potts model [6]

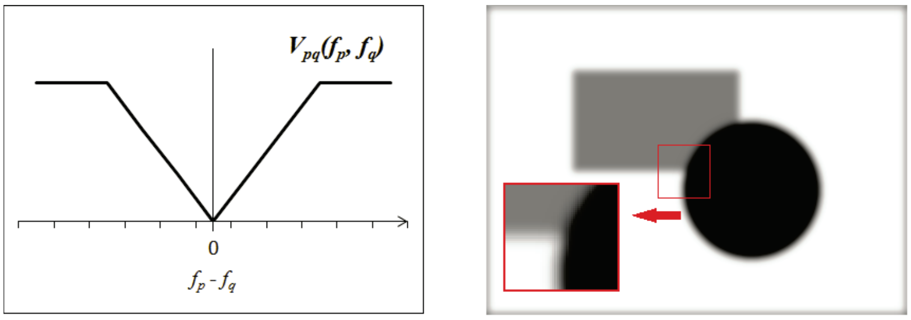

Figure 2.3: A graph of $V_{p,q}$ for truncated absolute distance on the left and the possible resulting labeling on the right.

$$V_{p,q}(f_p, f_q) = \begin{cases} 0 & \text{if } f_p = f_q \\ K & \text{otherwise,} \end{cases}$$

where K is a constant again. A graph of the Potts model and the possible resulting labeling f of the same input again is in figure 2.4.

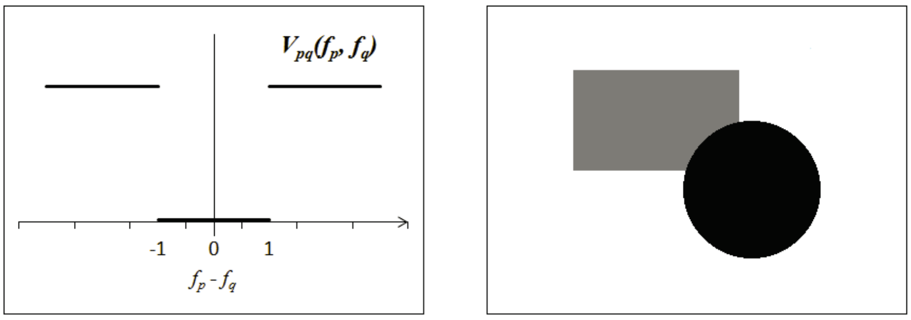

Figure 2.4: A graph of $V_{p,q}$ for Potts model on the left and the resulting labeling on the right.

This model leads to labelings which consist of several regions where pixels have equal labels in the same region and therefore, we call it piecewise constant model.

Chapter 3

Graph Cuts

Graph cuts is an energy minimization approach which works for a wide class of energy functions that we defined in the previous chapter.

First, we give some graph cuts basics and definitions, then we present how to minimize energy function via graph cuts.

3.1 Graph Cut Basics

Let $G = \{\mathcal{V}, \mathcal{E}\}$ be a directed weighted graph, where $\mathcal{V}$ is a set of vertices and $\mathcal{E}$ is a set of edges with nonnegative weights that connect the vertices. The set of vertices $\mathcal{V} = \{s, t\} \cup P$ consists of vertices which represent pixels P and special vertices $\{s, t\}$ called terminals, where s is called a source and t is called a sink.

Each edge of $\mathcal{E}$ is assigned a weight or cost $w(p, q)$, which has to be nonnegative. A cost of a directed edge (p, q) can be different from the cost of the reverse edge (q, p).

A set of edges $\mathcal{E}$ consists of two types. The first type of edges, called t-links, connect pixel vertex in P with a terminal. The other type of edges, called n-links, connect two pixel vertices between each other. An example of such a graph is in figure 3.1, where the green circles denote the terminals and the green edges represent t-links. The black circles denote the pixel vertices and black edges represent n-links.

3.1.1 Minimum Cut Problem

An s-t-cut C (shortly cut) is a subset of edges ($C \in \mathcal{E}$) with a property that the terminals are separated in the induced graph $G(C) = \langle \mathcal{V}, \mathcal{E} - C \rangle$. When we delete the edges of a cut C, we get a binary partition of the vertices $\mathcal{V}$ into two disjoint sets S and T, which satisfies the conditions [24] that

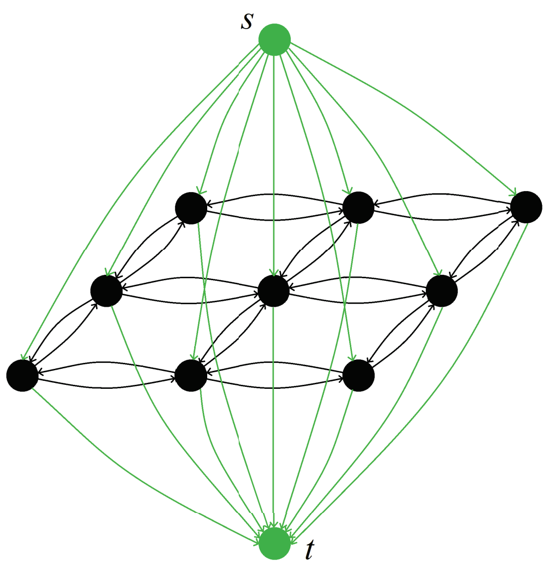

Figure 3.1: A general graph with terminals.

$$V = S \cup T,$$

$$S \cap T = \emptyset,$$

$$s \in S,$$

$$t \in T.$$

An example of a graph cut is in figure 3.2, where the red edges represent a cut.

The cost of the cut, we denote $|C|$, is the sum of costs of all its edge weights

$$c(S,T) = \sum_{u \in S, v \in T, (u,v) \in E} c(u,v).$$

The minimum s-t-cut (shortly min-cut) problem is to find a cut with the smallest cost among all possible cuts in the graph. This is equivalent to computing the maximum flow from the source to the sink, according to Ford and Fulkerson theorem [11].

In vision problems, the most common type of a graph is a grid-like graph, shorly grid. Thus, we will further focus only on grids, which we deeply discuss at the end of this chapter in section 3.3. For the grids, special min-cut algorithms were presented. Boykov and Kolmogorov [8] developed a fast augmenting path algorithm, where the observed running time is linear. Nowadays, an even faster algorithm exists and was presented in [14].

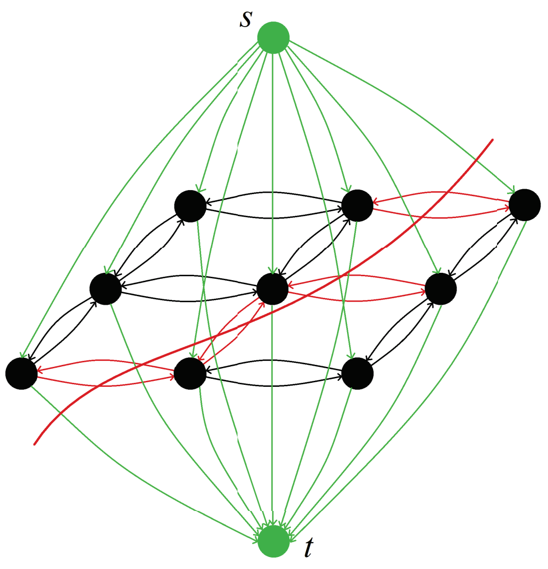

Figure 3.2: A graph cut.

3.2 Optimization via Graph Cuts

Basic min-cut algorithms are inherently binary techniques. Thus, binary problems represent the most basic case for graph cuts. In a binary optimization, the size of a label set is $|L| = 2$. For this case, a precise solution exists and the problem is solved via single min-cut in polynomial time [11].

Although graph cuts provide an inherently binary optimization, it is possible to use them also for energy minimization with more than two labels. In case of $|L| > 2$, we call the problem multilabeling or multi-label problem. In specific cases, energy minimization for multiple labels is precise, but in general we have to search for only approximate solution of minimization.

The example of the exact solution for multiple labels found in polynomial time is the case of linearly ordered set of labels. Unfortunately, this assumption is a quite big limitation, because in vision problems the labels usually can not be however order.

The last and the most interesting case for us is the case of multiple labels without any ordering. This case of energy minimization leads to solving multiway cut problem, which is NP-hard. Therefore, we need to apply special algorithms searching for approximate solutions. Only two types of algorithms solving the multiway cut exists. The first type of algorithms solves the problem via cuts on graph with k terminals, where k is the number of labels. One of them was presented in [15]. Due to slow performance of these algorithms, we rather focus on the second type of algorithms which are based on iterative solving of min-cut problem on graph with two terminals, described in the previous section. For these algorithms, it is proven that they find a solution within a factor of 2 from the global optimum [10].

Table 3.1 gives a brief summary of the optimization possibilities.

Optimization type	Set of labels	Class of problem	Solution
Binary labeling	|L| = 2	P	Exact
Multiple labeling	|L| > 2, linearly ordered	P	Exact
	|L| > 2, general	NP - hard	Approximate

Table 3.1: Possible optimization cases.

3.2.1 Graph Cuts For Binary Optimization

In a binary optimization, the size of a set of labels is $|L| = 2$, where $L = \{0, 1\}$. To find a globally optimal labeling, we also need to restrict smoothness term $V_{p,q}$ to

$$V_{p,q}(0,0) + V_{p,q}(1,1) \leq V_{p,q}(0,1) + V_{p,q}(1,0),$$

which is called regularity condition [9].

We can formulate the binary optimization problem as a simple optimization over binary variables corresponding to image pixels. A graph consists of vertices $p \in P$, which represent pixels, and two additional vertices, terminals s and t. The terminals represent two possible intensity values. Source s represents label 0 and sink t represents label 1.

According to [9], the data term $D_p(f_p)$ is a fixed penalty for assigning one of the label from L to a pixel p:

$$D_p(f_p) = K,$$

where K is a constant, which has to be nonnegative. To represent data term in a graph, we create two t-links for each pixel vertex. For t-link (s, p), we set the weight to $D_p(1)$ and for t-link (p, t), we set the weight $D_p(0)$.

Next we apply smoothness term $V_{p,q}$ in a graph. We denote the smoothness term as follows

$$V_{p,q}(f_p, f_q) = \lambda,$$

where $\lambda > 0$. We add n-links between neighboring pixel vertices using an arbitrary neighborhood system and set the weight of these edges to a constant λ. The resulting graph can be identical to what we presented in figure 3.1.

In order to solve the problem, we only need to find a single min-cut, which minimizes the energy function $E(f)$ and gives us the final labeling f. A min-cut just corresponds to a binary labeling f that assigns labels L to image pixels. If $p \in S$, we assign $f_p = 0$ and if $p \in T$, we assign $f_p = 1$. An example of the final labeling depending on the min-cut is in figure 3.3. The unfilled circles represent the set S, i.e. the label 0 and the filled circles represent the set T, i.e. the label 1.

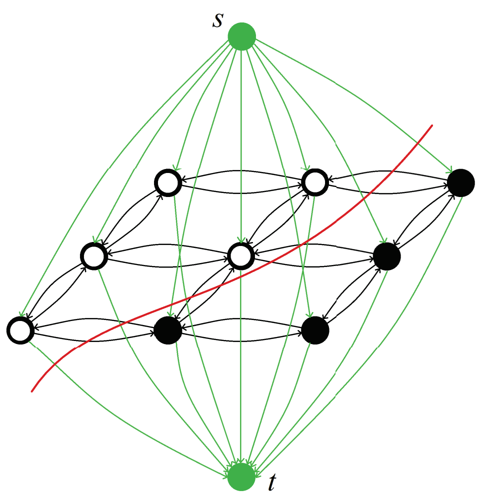

Figure 3.3: The graph cut and the labeling.

3.2.2 Exact Multilabeling Optimization

For the exact multilabeling optimization, we assume, as in [9], that the set of labels are linearly ordered integers in the range $L = \{1, ..., k\}$ and the smoothness cost is as follows

$$V_{pq} = \lambda_{p,q}|f_p - f_q|,$$

where $\lambda_{p,q}$ is a constant again.

We construct the graph in the following way. For each pixel p, we create a set of vertices $\{p_1, ..., p_{k-1}\}$ and add two additional terminals s and t as usual. Then we connect the vertices with edges $\{t_1^p, ..., t_k^p\}$, where $t_1^p = (s, p_1)$, $t_j^p = (p_{j-1}, p_j)$ and $t_k^p = (p_{k-1}, t)$. Each edge t_j^p has weight

$$w_{t^p} = D_p(j) + K_p = D_p(j) + 1 + (k-1)\sum_{q \in N} \lambda_{p,q},$$

where N is some neighboring system of pixel p. Next we add an edge (p_j, q_j) for each pair of neighboring sites p, q and for each $j \in \{1, ..., k-1\}$ with weight $\lambda_{p,q}$. We can see the example in figure 3.4. The shown part of graph corresponds to the three neighboring pixels $\{p, q, r\}$ and four labels $\{l_1, l_2, l_3, l_4\}$.

We get the final labeling via single min-cut on the created graph. For each pixel p, the weights for edges t_i^p are assigned so large that the min-cut separates always exactly one edge. If the min-cut separates the edge t_i^p, we assign label i to p. Figure 3.5 shows the min-cut and the final labeling. The min-cut crosses the t_3^p and t_3^q edge, so the pixels p and q get the labeling l_3, which we represent by the unfilled circles. For the pixel r, the min-cut crosses the edge t_2^r, so the pixel gets the labeling l_2, which we represent by the filled circles.

A generalized approach with convex V_{pq} was presented by Ishikawa [13]. The graph construction is similar to the construction given above, but it requires adding more edges

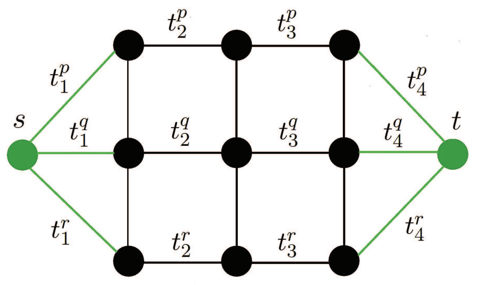

Figure 3.4: A graph corresponding to the three pixels p, q, r and four labels l_1, l_2, l_3, l_4.

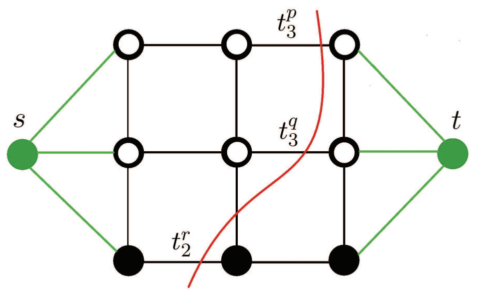

Figure 3.5: The min-cut and the resulting label assigning.

between p_i's and q_j's. Its disadvantage is that a convex V_{pq} is not usually suitable for the vision applications, because a convex V_{pq} corresponds to the everywhere smoothness model, while we require a piecewise smoothness model.

3.2.3 Approximate Optimization

Optimization for general multiple labels is the most interesting case for us. A single min-cut is not sufficient now and thus we need to apply algoritms solving the problem via iterative min-cut solutions.

In this subsection, we describe the most known approximation methods, the expansion and the swap algorithms from [10].

The swap algorithm can be used whenever

$$V_{pq}(\alpha, \alpha) + V_{pq}(\beta, \beta) \leq V_{pq}(\alpha, \beta) + V_{pq}(\beta, \alpha)$$

for all $\alpha, \beta \in L$, which we call the swap inequality.

The expansion algorithm may be used whenever

$$V_{pq}(\alpha, \alpha) + V_{pq}(\beta, \gamma) \leq V_{pq}(\alpha, \gamma) + V_{pq}(\beta, \alpha)$$

for all $\alpha, \beta, \gamma \in L$, which we call the expansion inequality and which is more restrictive than swap inequality.

Both inequalities grant discontinuity preserving V_{pq}'s as we require. In case of swap algorithm, it satisfies semimetric discontinuity preserving interaction potential and in the case of expansion algorithm it satisfies metric potential. Moreover we need to satisfy the condition that the V_{pq} corresponds to a piecewise smoothness model.

The algorithms find a local minimum of the given energy function. However, we need to define the meaning of a local minimum here. For each labeling f, we define a set of moves M_f which consists of the moves to other labelings that are allowed from the labeling f. Then the labeling f is a local minimum with respect to the set of moves, if for any $f' \in M_f$ is $E(f') \geq E(f)$.

For the swap algorithm, a move $f^{\alpha\beta}$ is called an α - β swap if the only difference between labeling f and $f^{\alpha\beta}$ is that some pixels labeled α in f are now labeled β in $f^{\alpha\beta}$, and some pixels labeled β in f are now labeled α in $f^{\alpha\beta}$. Then M_f is defined as the set of α - β swaps for all pairs of labels $\alpha, \beta \in L$.

In the expansion algorithm, a move f^{α} is called an α-expansion if the only difference between f and f^{α} is that some pixels that were not labeled α in f are now labeled α in f^{α}. M_f is then defined as the set of α-expansions moves for all labels $\alpha \in L$.

We show an example of both defined moves in figure 3.6.

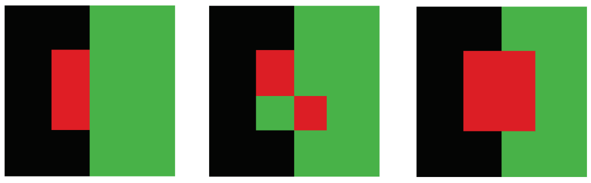

Figure 3.6: An example of the presented moves (from left to right): the initial labeling, the labeling within one red-green swap and the labeling within one red-expansion move from the labeling on the left.

Both algorithms find a local minimum with respect to their own moves, but the resulting solution depends havily on the initial labeling since a high dimensional energy has a huge number of local minima. Moreover, the number of moves from each labeling is exponential in the number of pixels, thus the direct search for an optimal move and solution is not feasible. This is where graph cuts are essential, because we compute the optimal move with the min-cut on a given graph.

The algorithms are iterative. They start with an initial labeling f and then they cycle in some order until convergence over all labels $\alpha \in L$ (pairs of $\alpha, \beta \in L$), find the optimal labeling out of all possible moves and change the current labeling to the found better one. They gradually converge to the local minimum with respect to their moves. Thus the key step is how to find the optimal move, which is performed by finding a min-cut on a certain graph. The actual graph constructions can be found in [10].

Further we focus only on expansion move algrithm that we implement in this thesis. We describe the algorithm in more detail in the following chapter 4.

3.3 Grid Graphs

The last assumption for this thesis and our implementation is the use of energy functions, which lead to grid-like graphs (shortly grids), because GridCut solves min-cut problem only for this type of graphs. Luckily, this assumption is not a big restriction here, because solving pixel-labeling problem naturally leads to the creation of grids.

The difference between a general graph and a grid is that each vertex in the grid is connected only with vertices in some neighboring system.

For 2D images, the neighboring system is typically 4 connected. It means that each pixel is connected with its neighbors on the right, bottom, left and top. The exception are pixels on the borders, where some neighbors are missing. An example of a 4 connected nighboring system is in figure 3.7.

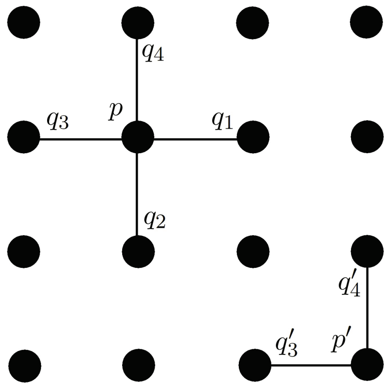

Figure 3.7: 4 connected nighboring system. For a pixel p, the set of adjacent pixels is $\{q_1, q_2, q_3, q_4\}$. For a pixel p' at the border, the set of adjacent pixels is $\{q'_3, q'_4\}$.

If we also consider diagonal neighboring pixels, we get an 8 connected neighboring system, shown in figure 3.8.

We can also extend the grids to 3D. The first neighboring system will be then 6 connected, because for each pixel we consider also the neighbors on the front and back. If we also consider diagonal neighboring pixels, we get a 26 connected neighboring system.

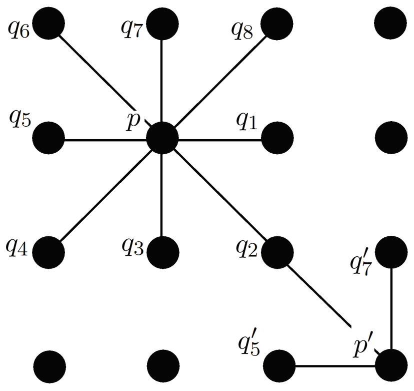

Figure 3.8: 8 connected neighboring system. For a pixel p, the set of adjacent pixels is $\{q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8\}$. For a pixel p' at the border, the set of adjacent pixels is $\{q_5', q_2, q_7'\}$.

Chapter 4

Expansion Move Algorithm

Expansion move algorithm or shortly α-expansion was introduced in [10] in 2001 and it is one of the most effective algorithms for approximate optimization via graph cuts. The algorithm can be used whenever $V_{p,q}$ is a metric interaction potential on the space of labels.

The algorithm runs in cycles. In each cycle, the algorithm performs an iteration for every label $\alpha \in L$, thus the one cycle takes |L| iterations. The certain order of labels can be fixed or random. A cycle is successful if a better labeling f^α, ie. labeling with lower energy, is found at any iteration and the better labeling f^α becomes the current labeling f. The algorithm terminates after the first unsuccessful cycle since no further improvement is possible. The process is described in algorithm 1:

```
1  Start with an arbitrary labeling $f$;
2  Set $success = 0$;
3  for each label $\alpha \in L$ do
4      Find $f^\alpha = \text{argmin } E(f')$ among $f'$ by one $\alpha$-expansion of $f$;
5      if $E(f^\alpha) < E(f)$ then
6          Set $f = f^\alpha$;
7          Set $success = 1$;
8      end
9  end
10 if $success = 1$ then
11     Go to 3;
12 end
13 return labeling $f$;
```

Algorithm 1: The pseudocode of the expansion move algorithm.

The algorithm always terminates in a finite number of cycles. In fact, it is proven that it terminates in O($|f|$) cycles [24]. However, the experiments in [10] show that the algorithm terminates after a few cycles and moreover, most of the improvements occur during the first cycle. After the algorithm terminates, the energy of the final labeling is a local minimum of the energy with respect to the expansion moves. It is also proven that such a local minimum lies within a multiplicative factor of the global minimum [10].

4.1 Energy Minimization by Expansion Moves

The main part of the algorithm is to compute the lowest energy labeling within a single α-expansion move of f. This subproblem is solved efficiently with a single min-cut. The structure of the graph is determined by the current labeling and by the label α and the graph dynamically changes after each iteration.

The subproblem is an energy minimization problem over binary variables, altough the overall problem that the expansion move algorithm is solving involves nonbinary variables. The perspectiove of binary variables is here because each pixel will either keep its old value from labeling f or get the new label α .

Given a labeling f, after a single min-cut, the new labeling f^α can be encoded by $f^\alpha(p) = f(p)$ if $p \in T$ and $f^\alpha(p) = \alpha$ if $p \in S$. It tells us that in the first case the label stays the same and otherwise the label is changed to α.

4.2 Graph Construction

We construct a graph G for each term of energy function separately and then we merge all these partial graphs together like in [17]. The graph G contains $n + 2$ vertices $V = \{s, t, v_1, \ldots, v_n\}$. Each nonterminal vertex v_i represents the binary variable (pixel) x_i. There are no additional vertices for representing binary interactions of binary variables so this approach leads to the smaller graph unlike the graph construction in [10] and thus, the min-cut can be solved faster.

For each term of E, we add one or more edges to graph. First, we add edges for term E^i that depend on one variable x_i. If $E^i(0) < E^i(1)$, then we add edge (s, v_i) with the weight $E^i(1) - E^i(0)$. Otherwise, we add the edge (v_i, t) with the weight $E^i(0) - E^i(1)$. We show both possibilities in figure 4.1.

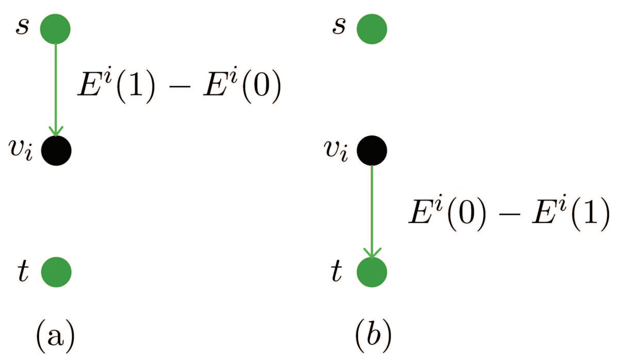

Figure 4.1: Graphs for term E^i. (a) Graph for E^i, where $E^i(0) > E^i(1)$. (b) Graph for E^i, where $E^i(0) \leq E^i(1)$.

Next, we add an edge for a term $E^{i,j}$, which depends on two variables x_i and x_j. We can see the representation of the term in table 4.1.

$$E^{i,j} = \begin{array}{|c|c|}\hline E^{i,j}(0,0) & E^{i,j}(0,1) \\ \hline E^{i,j}(1,0) & E^{i,j}(1,1) \\ \hline \end{array} = \begin{array}{|c|c|}\hline A & B \\ \hline C & D \\ \hline \end{array}$$

$$\begin{array}{|c|c|}\hline A & B \\ \hline C & D \\ \hline \end{array} = A + \begin{array}{|c|c|}\hline 0 & 0 \\ \hline C - A & C - A \\ \hline \end{array} + \begin{array}{|c|c|}\hline 0 & D - C \\ \hline 0 & D - C \\ \hline \end{array} + \begin{array}{|c|c|}\hline 0 & B + C - A - D \\ \hline 0 & 0 \\ \hline \end{array}$$

Table 4.1: Representation of the term $E^{i,j}$ on top and a rewritten expression on bottom.

The first term is a constant function, so we don't add any edges. The second and the third terms depend only on one variable. The second term depends on the variable x_i and the third on x_j. Therefore, we use construction given above and add one edge for x_i and one edge for x_j. For the last term, we add edge (v_i, v_j) with the weight $B + C - A - D$. See figure 4.2.

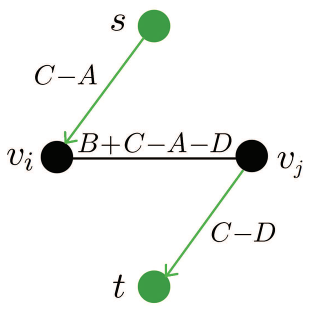

Figure 4.2: Graph for the $E^{i,j}$, if $C > A$ and $C > D$. Green edges are for the second and the third terms. The black one is for the fourth term.

Now, when we want to create a full graph of two nonterminal vertices, we have to apply all rules given above and merge the graphs together. We start with vertices and then we apply rules for adding edges. The whole process is described in algorithm 2.

From the algorithm, the resulting graph will have a source, a sink and a vertex for each pixel. There are edges between neighboring vertices and also every vertex is connected to the source and/or the sink.

```
Create a source $s$ and a sink $t$;
for each term $E^i(x_i)$ do
    Add a vertex $v_i$ corresponding to pixel $i$;
    if $E^i(1) > E^i(0)$ then
        Create an edge from $s$ to $v_i$ of cost $E^i(1) - E^i(0)$;
    else
        Create an edge from $v_i$ to $t$ of cost $E^i(0) - E^i(1)$;
    end
end
for each term $E^{ij}(x_i, x_j)$ do
    if $E^{ij}(1,0) - E^{ij}(0,0) > 0$ then
        Create an edge from $s$ to $v_i$ of cost $E(1,0) - E(0,0)$;
    else
        Create an edge from $v_i$ to $t$ of cost $E(0,0) - E(1,0)$;
    end
    if $E^{ij}(1,1) - E^{ij}(1,0) > 0$ then
        Create an edge from $s$ to $v_j$ of cost $E(1,1) - E(1,0)$;
    else
        Create an edge from $v_j$ to $t$ of cost $E(1,0) - E(1,1)$;
    end
    Connect $v_i$ to $v_j$ by an edge of cost $E(0,1) + E(1,0) - E(0,0) - E(1,1)$
end
```

Algorithm 2: . Full graph creation for an expansion move algorithm.

Chapter 5

Applications

Energy minimization via graph cuts has a wide range applications in computer vision and graphics. We show applications of energy minimization for general multiple labels and for each of them we give instruction what both data term and smoothness term should look like.

5.1 Segmentation

Segmentation problem is a partitioning an image or multi-dimensional space into segments. The goal of the segmentation is to change the representation of the input to locate objects and boundaries or just simplify it.

The result of the segmentation is a set of segments that cover the entire image or space. Each of the pixels in a region are similar with respect to some characteristic, user interaction or computed property, such as color, intensity, or texture. Adjacent regions are significantly different with respect to the same characteristic.

5.1.1 Image Segmentation

The application of image segmentation is usually interactive. The user draws strokes with different colors to mark different segments. The strokes' colors represent a set of labels L and data term is derived from the user strokes according to [9] in the following way:

$$D_p(f_p) = \begin{cases} \lambda K & \text{if } p \text{ has a brush stroke,} \\ -lnPr(I_p|f_p) & \text{otherwise,} \end{cases}$$

where

$$K = 1 + max_{p \in P} \sum_{\{p,q\} \in N} V_{p,q}$$

and $\lambda = \{0, 1\}$. The value of λ indicates whether a brush stroke f_p is presented. $\lambda = 1$ if the pixel p has a brush stroke f_p and $\lambda = 0$ if the pixel p has a brush stroke $L \backslash \{f_p\}$. Otherwise, we use intensities of marked pixels to get histograms for intensity distributions $Pr(I|f_p)$ and then, we use these histograms to set $D_p(f_p)$ as negative log-likelihoods.

A particular example of smoothness term that we take from [23] is defined as a variant of the Potts model

$$V_{p,q}(f_p, f_q) = \begin{cases} \lambda_1 exp\big(-\beta(I_p - I_q)^2\big) + \lambda_2 & \text{if } I_p \neq I_q \\ 0 & \text{otherwise,} \end{cases}$$

where I_p and I_q are pixel intensities, λ_1 and λ_2 are constants and $\beta = \big(2(I_p - I_q)\big)^{-1}$.

An example of image segmentation for multiple labels is in figure 5.1.

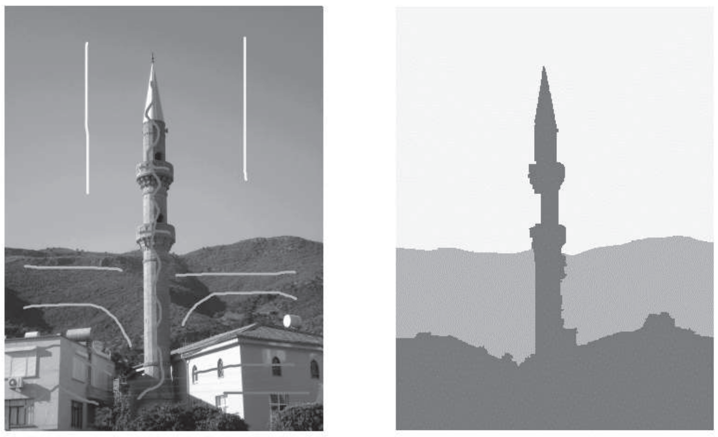

Figure 5.1: An image segmentation for multiple labels.

5.1.2 Multi-dimensional Segmentation

The image segmentation problem can be extended into a multi-dimensional segmentation. The papers [7] and [9] discuss this type of a segmentation and give the forms of the smoothness and data term for two typical uses of this application. The application is binary in these papers, but it can be easily extended to segmentation with multiple labels.

The first use they discuss is a human organs segmentation from medical data such as CT images. Again, the application is interactive as the image segmentation. The user draws the strokes on a single CT image and the whole organs, like bones or tumors are then segmented. An example of a bone segmentation is in figure 5.2.

The second use they discuss is object segmentation in a video sequence. The application is interactive again. The user marks the objects he wants to follow in the following frames and data term is then derived from the marks. The visual example is in figure 5.3, where the three cars are segmented in each frame.

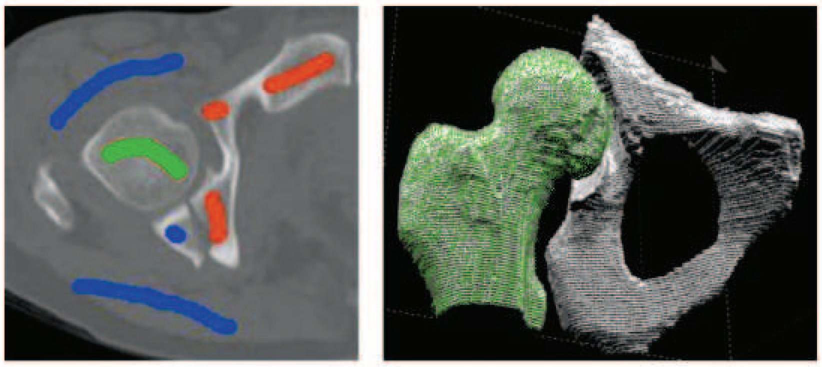

Figure 5.2: 3D segmentation from [9].

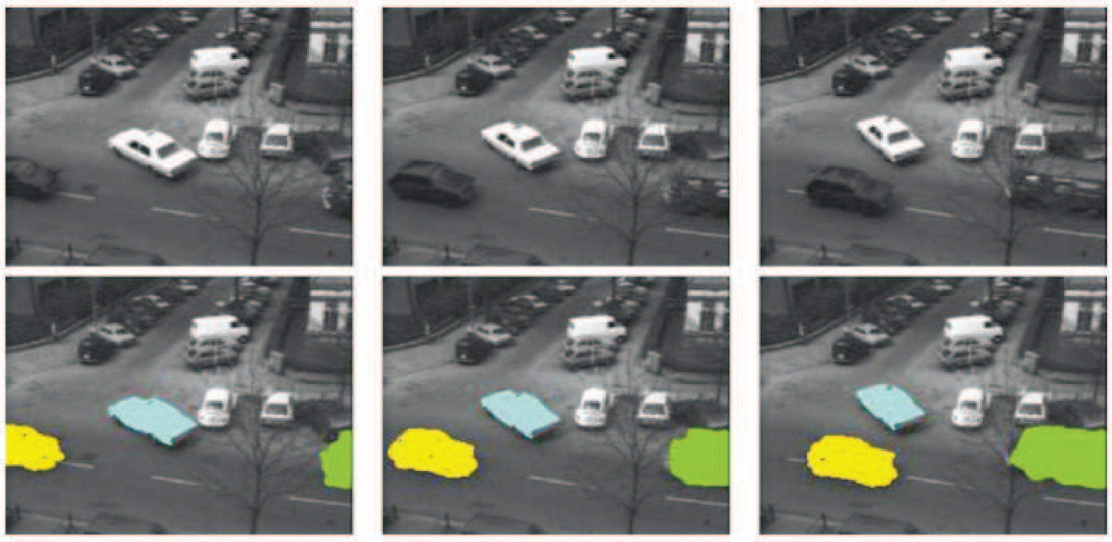

Figure 5.3: An example of a segmentation in a video sequence from [9].

5.1.3 Coloring

Coloring or painting is a similar application of image segmentation. It is a process of adding colors to hand-made drawings or black and white images, which is a standard operation in image manipulation programs. In fact, the goal of the problem stays in partitioning image to multiple segments.

In most of the programs, a variant of a floodfill algorithm is used to speed-up painting. The floodfill algorithms work well for images with homogenous regions and continuous outlines, but they are almost useless for hand-made drawings with more complicated structures. Recently, LazyBrush algorithm was developed and it promises to handle the issues and to work well for various drawing styles [22].

Naturally, the application is interactive, where the user draws strokes of colors, which are applied to image segments. The strokes' colors represent the set of labels and the strokes occurrence are considered constraints applied to different range of λ values in the data term

$$D_p(f_p) = \lambda K,$$

where K is a constant and $\lambda = \langle 0, 1\rangle$. If the pixel p does not have a brush stroke, we set $\lambda = 0$. If the pixel p has a brush stroke, we set $\lambda = 1$ in case of a hard constraint, otherwise we set λ to some value from $(0, 1)$.

Smoothness term is a simple Potts model

$$V_{p,q}(c_p, c_q) = \begin{cases} I_p & \text{if } c_p \neq c_q \\ 0 & \text{otherwise,} \end{cases}$$

where I_p is a pixel intensity and c_p and c_q colors for assigning. The final result of coloring can be even improved by preprocessing of input image. More information can be found in the paper of LazyBrush [22].

An example of the application is in the figure 5.4.

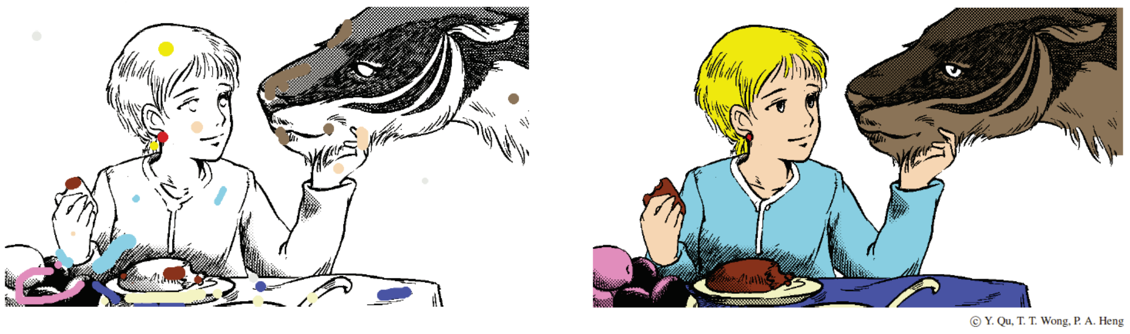

Figure 5.4: Coloring of a hand-made drawing from [22].

5.2 Image Restoration

Image restoration is a vision problem where the goal is to restore the original pixel intensities from the observed noisy data.

In this application, the set of labels are pixel intensities and the data term is usually the squared difference between the label and the observed distance. Some of the pixels can be even obscured. These pixels have automatically a data cost of 0 for any intensity:

$$D_p(f_p) = \begin{cases} (I_p - f_p)^2 & \text{if } p \text{ is not obscured} \\ 0 & \text{otherwise.} \end{cases}$$

The smoothness energy can be the squared absolute distance

$$V_{p,q}(f_p, f_q) = min\big(K, (|f_p - f_q|)^2\big),$$

but it works as well for the absolute distance or Potts model according to [23] and [6]. In figure 5.5 is an example of image restoration.

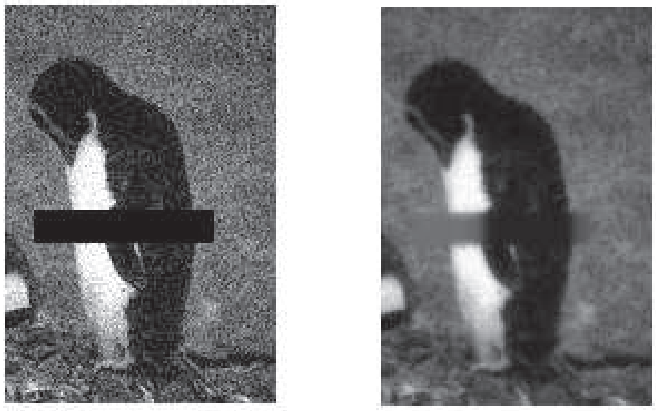

Figure 5.5: An image restoration with obscured pixels.

5.3 Stereo

Stereo is another classic application of energy minimization. The goal of stereo is to compute the correspondence between pixels of two or more images of the same scene, which is obtained by cameras with slightly different view points.

We consider a formulation of stereo problem given in [10] and [8]. The data term is the squared difference in intensities between corresponding pixels from the left and from the right image.

$$D_p(f_p) = (I_p - I_q)^2,$$

where I_p is pixel intensity from the left image and I_q is intensity of corresponding pixel q to p from the right image and the smoothness term is the same as for the image restoration.

Stereo images of multi-depth objects usually contain occluded pixels. In that case, the energy function consists of the data term, the smoothness term and also the occlusion penalty. Solving stereo with occlusions was reported for example in [5] and recently in [23], where the occlusion is considered without making extra assumptions about scene geometry.

An example of the application is in figure 5.6.

5.4 Optical Flow

Accurate estimation of optical flow is another application of gruph cuts, which is similar to stereo. The input is also two or more images of the same scene given by the same cameras with slightly different view points.

In [18], they show how non-convex energies can be formulated and optimized discretely in the context of optical flow estimation. They start with several solutions produced by fast continuous flow estimation algorithms and then they iteratively fuses these solutions by the computation of min-cuts on graphs.

Data term for this case is

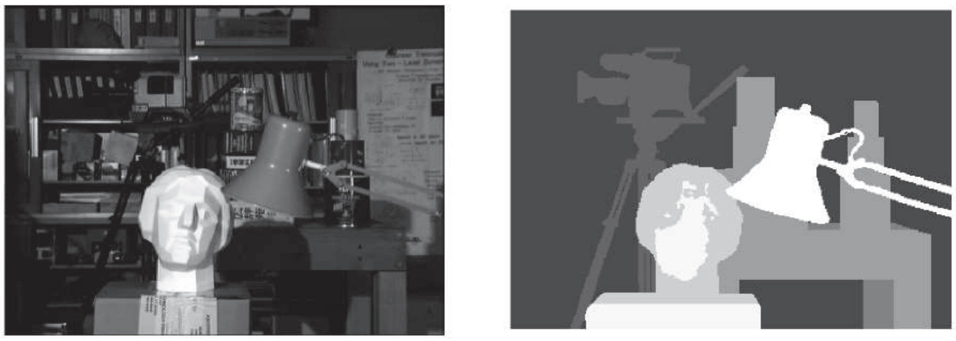

Figure 5.6: Stereo example from [3].

$$D_p(f_p) = \rho\big(I^1(p + f_p) - I^0(p)\big),$$

where I^0 and I^1 represent the images and ρ is the Geman-McClure robust penalty function [18].

Smoothness term penalizes changes in horizontal and vertical flow between adjacent pixels

$$V_{p,q}(f_p, f_q) = \rho\Big(\frac{u_p - u_q}{||p - q||}\Big) + \rho\Big(\frac{v_p - v_q}{||p - q||}\Big),$$

where $||p - q||$ is the Euclidean distance between the pixel centers p and q and ρ is negative logarithm of a Student-t distribution [18].

Example of the fusion flow result is in figure 5.7.

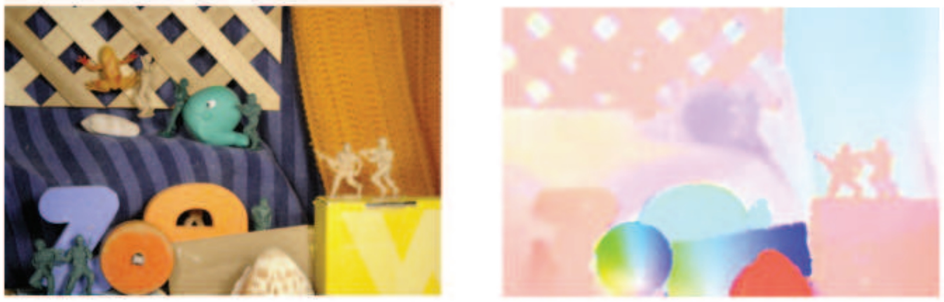

Figure 5.7: An accurate estimation of optical flow. The left input image is on the left, the result is on the right.

5.5 Photomontage

The photomontage application seamlessly stitches together multiple photographs and can be used for panoramic stitching or group photo merging.

The input is a set of m aligned images $\{S_1, S_2, ..., S_m\}$ of equal dimension and the labels are the image indexes, i.e. $\{1, 2, ..., m\}$. The final output image is formed by copying colors from the input images according to the computed labeling [23].

In case of panoramic stitching, the data term depends on the fact, if a pixel is in the field of view of the image or not:

$$D_p(f_p) = \begin{cases} 0 & \text{if } p \text{ is in the field of view of image} \\ \infty & \text{otherwise.} \end{cases}$$

The smoothness energy is derived from the distance of pixels between two images and it measures how visually noticeable the seam between p and q is in the composite:

$$V_{p,q}(f_p, f_q) = |S_{l_p}(p) - S_{l_q}(p)| + |S_{l_p}(q) - S_{l_q}(q)|$$

See the example of panoramic stitching in figure 5.8.

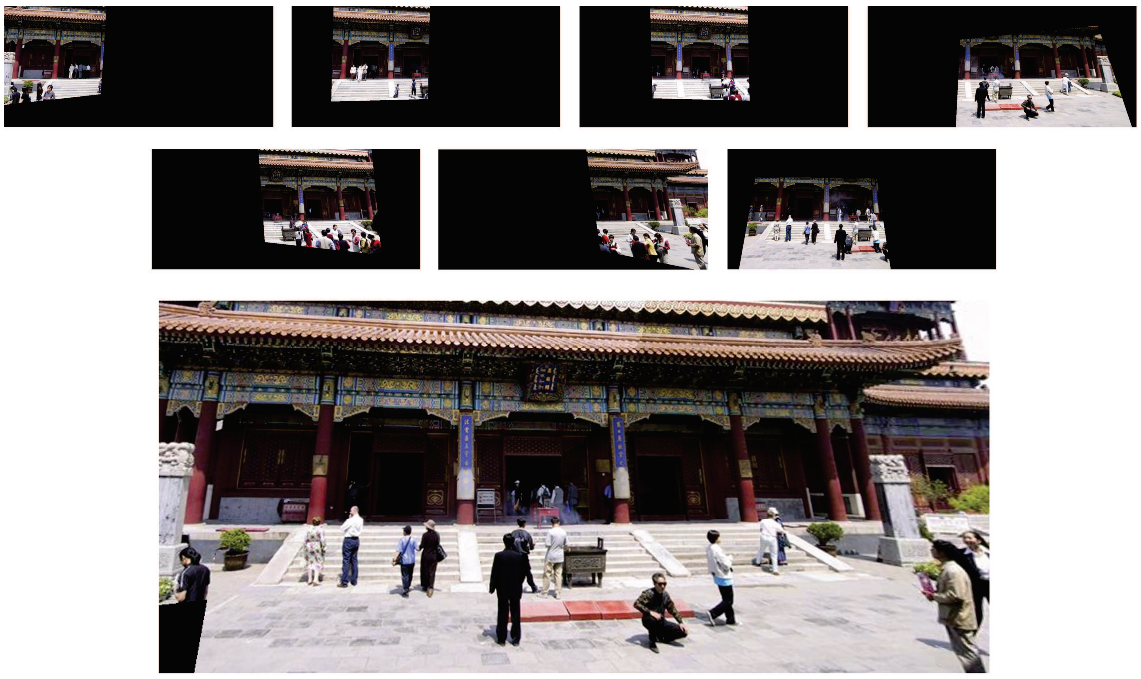

Figure 5.8: Panoramic stitching. The resulting image is on the bottom.

The second case stitches together the group of photographs with people. The best depiction of each person is included in a composite. This case is interactive and the data term is then derived from the user's strokes in the same way as in image segmentation. The smoothness term is modified from the first case to encourage seams along strong edges. More information is in [23] and an example of this application is in figure 5.9.

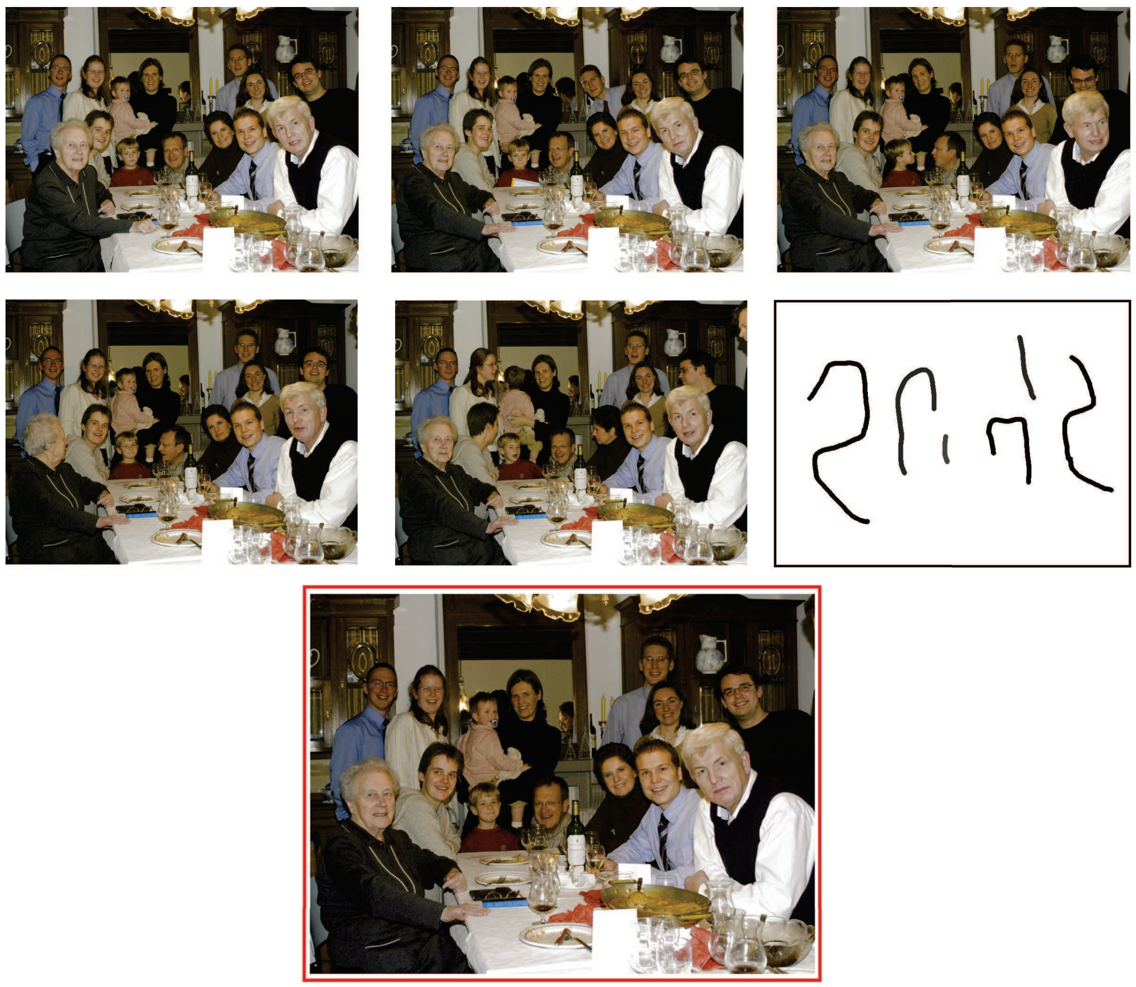

Figure 5.9: A group photo merging. The resulting image is in the red border.

5.6 Image Rearrangement

Geometric rearrangement of images includes operations such as image retargeting, object removal, or object rearrangement. Each such operation can be characterized by a shift-map: the relative shift of every pixel in the output image from its source in an input image. This shift-map M represents the selected label for each output pixel.

We take the examples of terms from [20], but more information can also be found in [16] and [25]. Data term indicates constraints such as the change in image size, object rearrangement, a possible saliency map, etc. If a pixel (u, v) is supposed to originate in output image from location (x, y) in the input image, the appropriate shift gets zero energy, otherwise the shift gets a very high energy:

$$D_p\big(M(u,v)\big) = \begin{cases} 0 & \text{if } u + t_x = x \text{ and } v + t_y = y \\ \infty & \text{otherwise.} \end{cases}$$

Some of the pixels from the input image should disappear in the output image. For this case, the data term has to be extended in the following way: for the pixel which should be removed, data cost is set to very high energy, otherwise the data cost is set to some low energy.

Smoothness term minimizes the discontinuities in the output image. A discontinuity exists between two neighboring pixels (u_1, v_1) and (u_2, v_2) in the output image if their shift-maps are different: $M(u_1, v_1) \neq M(u_2, v_2)$. The smoothness term relies on color differences and also on gradient differences between corresponding neighboring pixels in the output and input image. The smoothness term is then:

$$V_{p,q}(M) = \sum_{(u,v)\in R} \sum_i \Big(R\big((u,v)+e_i\big) - I\big((u,v)+M(u,v)+e_i\big)\Big)^2 + \\ + \beta \sum_{(u,v)\in R} \sum_i \Big(\nabla R\big((u,v)+e_i\big) - \nabla I\big((u,v)+M(u,v)+e_i\big)\Big)^2$$

where e_i are the four unit vectors representing the four spatial neighbors of a pixel, the color differences are Euclidean distances in RGB, ∇I and ∇R are the magnitude of the input and output image gradients at these locations, and β is a weight to combine these two terms.

Example of the image rearrangement is in figure 5.10

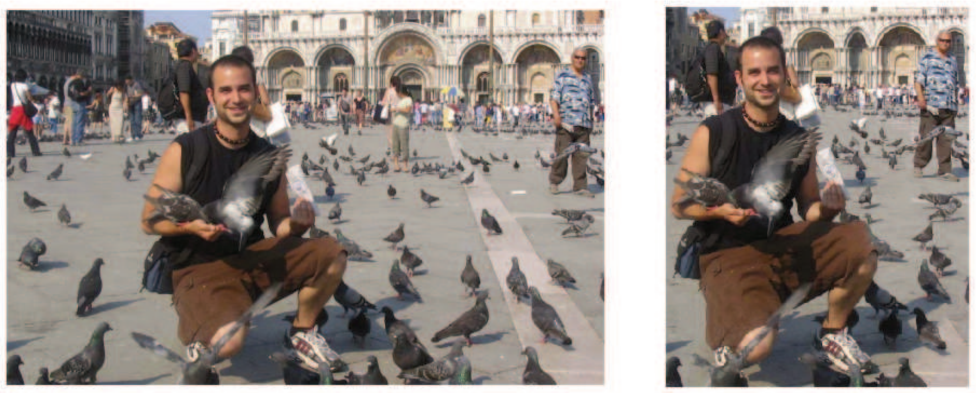

Figure 5.10: Geometric rearrangement of image. Input image on the left and the result on the right.

5.7 Image Registration

Image registration is the process of transforming different images into one coordinate system. It is usually used in medical imaging or the analysis of images from satellites where registration is useful in order to be able to compare or integrate the images.

Deformable image registration aims to go further and search for subject correspondences between the images. The problem consists of finding a non-linear local transformation that aligns the pixels that have in general an unknown relationship in the spatial and the intensity domain.

The problem is solved via graph cuts in [12], where a label assignment to a pixel p is associated with shifting the pixel by the corresponding vector d_p. The formulation of data cost is

$$D_p(f_p) = \int_\Omega \eta(|x-p|) \cdot \rho_h\Big(g(x), f\big(T(x)\big)\Big) dx,$$

where Ω is the source image, $\eta(\cdot)$ computes the influence of an image point x to a control point p, $g(x)$ is the target image, $T(x)$ is the transformation and ρ_h is a function measuring the intensity relation between the two images.

The smoothness term is defined as a distance function computing the magnitude of vector differences

$$V_{p,q}(f_p, f_q) = \lambda_{pq}|\big(R(p)+d_p\big) - \big(R(q)+d_q\big)|,$$

where λ_{pq} is a weighting factor which may vary over the spatial domain, $R(\cdot)$ projects the current displacement field on the level of the control points and d_p and d_q are the shifting vectors.

The example of the image registration is in figure 5.11.

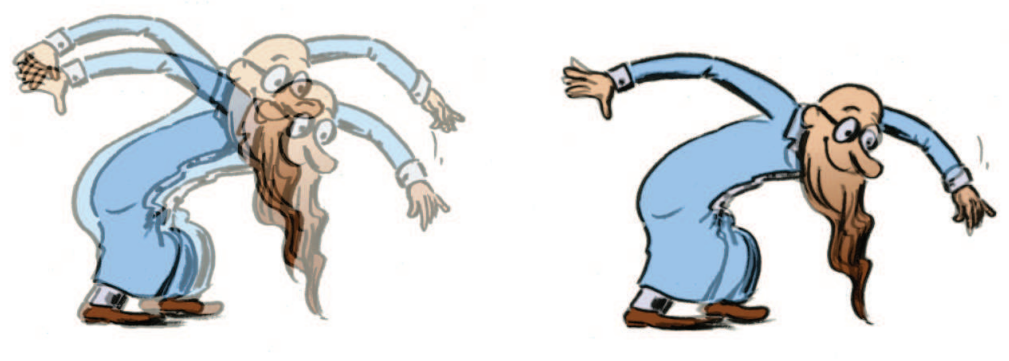

Figure 5.11: Image registration of a drawn person. The two input images, which are overlapping on the left, and the overlapping result from [21].

Chapter 6

Implementation

We implemented the expansion move algorithm in C++ programming language for 2D grids with 4 and 8 connected neighboring system and for 3D grids with 6 and 26 connected neighboring system. The resulting implementation of the algorithm is extension for the sequential version of GridCut and we also provide extension for the parallel GridCut for 2D grids with 4 connected neighboring system and for 3D grids with 6 connected neighboring system. We call the resulting implementation AlphaExpansion.

The algorithm is implemented according to [10] and [17] in the same way as we described in the previous chapters and we used GridCut [1] for solving each min-cut problem .

By implementing the expansion move algorithm, we extended the GridCut to be able to solve the multilabeling problem natively that was not possible before.

6.1 Application Programming Interface

We implemented the algorithm in the way that the programming interface is as simple as possible. The interface provides several functions to easily use the algorithm only in three steps:

1. Initialize the algorithm.
2. Run the algorithm.
3. Get the result.

6.1.1 Data and smoothness costs

First we need to define how the data costs and smoothness costs should be initialized. We assume that w, h and eventually d is a width, a height and a depth of a grid (image or space), n is a number of pixels (voxels), given by wh for 2D grids and whd for 3D grids, k is a number of labels and $NEIGHBORS$ is a constant different for each neighboring system and it represents a number of the pixel's neighbors in a neighboring system. We assume that for pixel p, the index p_i in a grid is $p_i = y \cdot w + x$, eventually $p_i = z \cdot d\_step + y \cdot w + x$,

Variable	Meaning	Calculated
w	Width of a grid	-
h	Height of a grid	-
d	Depth of a grid	-
n	Number of pixels (voxels)	wh in 2D whd in 3D
$NEIGHBORS$	Number of the pixel's neighbors	-
x	Coordinate in X axis	-
y	Coordinate in Y axis	-
z	Coordinate in Z axis	-
p_i	Pixel's index in a grid	$y \cdot w + x$ in 2D $z \cdot d\_step + y \cdot w + x$ in 3D
d_step	Depth step	$d\_step = wh$

Table 6.1: The smoothness table.

where x, y, z are the pixel coordinates in a grid and $d\_step = wh$. The summary is in table 6.1.

We expect that data costs are provided by the array of nk length, i.e. the array contains k values for each pixel. The values are ordered in the way, that for the index p_i of pixel p and the label l, the data cost value is on the array index $p_i \cdot k + l$. The ordering is shown in table 6.2.

$l_0(p_0)$	$l_1(p_0)$	$\dots$	$l_{k-1}(p_0)$	$l_0(p_1)$	$l_1(p_1)$	$\dots$	$l_{k-1}(p_1)$	$\dots$	$\dots$	$l_{k-1}(p_{n-1})$

Table 6.2: The data cost array.

Unlike data costs, smoothness costs can be provided in several ways. The first way is to provide smoothness costs by the function:

```
typedef type_energy (*SmoothCostFn)(int pix1, int pix2, int lab1, int lab2);
SmoothCostFn smooth_fn;
```

The parameters of the function are two neighboring pixels and their labels. The return value has to be of a data type for the resulting energy (we will explain energy data type later).

The second way is to provide smoothness costs by the smoothnes tables. The smoothness table is a 1D array of k^2 length defining smoothness costs for two neigboring pixels depending on their labels. For two neighboring pixels with indexes p, q and label l, the corresponding smoothness cost lies on index $p \cdot k + l$ in the table. For clarity, see table 6.3, where the numbers correspond to an index in the smoothness array.

When we want to provide different smoothness tables for each pair of neighbors, we need to provide smoothness costs as an array of $n \cdot NEIGHBORS/2$ smoothness tables. All these tables are in such order that there are $NEIGHBORS/2$ tables for the first pixel, then $NEIGHBORS/2$ tables for the second pixel etc. Thus, for the pixel p with index p_i, its

	l_0^q	l_1^q	l_2^q	$\dots$	l_{k-1}^q
l_0^p	0	1	2	$\dots$	$k-1$
l_1^p	k	$k+1$	$k+2$	$\dots$	$2k-1$
l_2^p	$2k$	$2k+1$	$2k+2$	$\dots$	$3k-1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
l_{k-1}^p	$k(k-1)$	$k(k-1)+1$	$k(k-1)+2$	$\dots$	k^2-1

Table 6.3: The smoothness table.

first smoothness table lies on index $p_i \cdot NEIGHBORS/2$. See table 6.4 for clarity, where S stands for one smoothness table and t is number of smoothness tables for each pixel.

$S_0(p_0)$	$S_1(p_0)$	$\dots$	$S_{t-1}(p_0)$	$S_0(p_1)$	$S_1(p_1)$	$\dots$	$S_{t-1}(p_1)$	$\dots$	$\dots$	$S_{t-1}(p_{n-1})$

Table 6.4: The smoothness cost array.

Now we describe an ordering of the smoothness tables for all the neighboring systems. For 2D grids with 4 connected neighboring system, we expect two tables for each pixel. The first table corresponds to the pair of the pixel p and its right neighbor and the second table corresponds to the pair of its bottom neighbor. We can rewrite this fact by offsets of a pixel's coordinates, see table 6.5:

Table	Offset X	Offset Y
1	$+1$	0
2	0	$+1$

Table 6.5: The smoothness tables for 2D grids with 4 connected neighboring system.

The meaning of the offset is clearly demonstrated in figure 6.1.

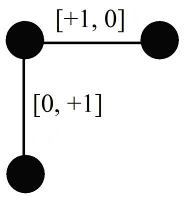

Figure 6.1: A demonstration of the two neighbors given by offsets.

For 2D grids with 8 connected neighboring system, we expect four tables for each pixel. The first two tables are the same as before and there are two other tables for diagonal neighbors. The order of the smoothness tables is in table 6.6, which are described by the neighbors' offsets again.

Table	Offset X	Offset Y
1	+1	0
2	0	+1
3	+1	-1
4	+1	+1

Table 6.6: The smoothness tables for 2D grids with 8 connected neighboring system.

A demonstration of the offsets in a grid is in figure 6.2.

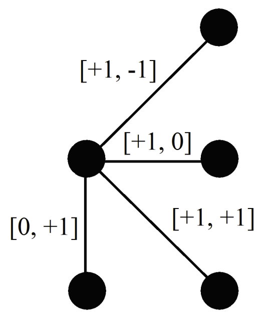

Figure 6.2: A demonstration of four neighbors given by offsets.

For 3D grids with 6 connected neighboring system, we expect three smoothness tables for each pixel and they are desribed by the offsets in table 6.7.

Table	Offset X	Offset Y	Offset Z
1	+1	0	0
2	0	+1	0
3	0	0	+1

Table 6.7: The smoothness tables for 3D grids with 6 connected neighboring system.

The last case are 3D grids with 26 connected neighboring system, where we expect thirteen smoothness tables for each pixel, which are desribed by the offsets again in table 6.8.

Table	Offset X	Offset Y	Offset Z
1	+1	0	0
2	0	+1	0
3	+1	-1	0
4	+1	+1	0
5	0	0	+1
6	0	-1	+1
7	0	+1	+1
8	-1	0	+1
9	-1	-1	+1
10	-1	+1	+1
11	+1	0	+1
12	+1	-1	+1
13	+1	+1	+1

Table 6.8: The smoothness tables for 3D grids with 26 connected neighboring system.

6.1.2 Initialization

The AlphaExpansion is header-only library and doesn't need to be compiled separately. There is a single header file for each grid topology. In order to start using AlphaExpansion, we have to add the "./include" directory to compiler's include path and #include the required header file in the code. The multithread "MT" versions are AlphaExpansion implementations calling the parallel version of GridCut.

```
#include <AlphaExpansion_2D_4C.h>
#include <AlphaExpansion_2D_4C_MT.h>

#include <AlphaExpansion_2D_8C.h>

#include <AlphaExpansion_3D_6C.h>
#include <AlphaExpansion_3D_6C_MT.h>

#include <AlphaExpansion_3D_26C.h>
```

The next step in using the algorithm is to initialize it. We allow to choose different data types of labels and smoothness, data and energy costs by template mechanism. The first parameter of the template is data type for labels, the second is for data and smoothness costs and the third parameter is for the resulting energy:

```
template<typename type_label, typename type_cost, typename type_energy>
```

To initialize the algorithm itself, it is only required to call the appropriate constructor. We provide three types of constructor for 2D grids with 4 connected neighboring system:

```
AlphaExpansion_2D_4C(int width, int height, int n_labels, type_cost *data, SmoothCostFn smooth_fn);

AlphaExpansion_2D_4C(int width, int height, int n_labels, type_cost *data, type_cost *smooth);

AlphaExpansion_2D_4C(int width, int height, int n_labels, type_cost *data, type_cost **smooth);
```

For all cases, the constructor parameters are width and height of the grid (image), number of labels and a 1D array with data costs. There are three ways setting the smoothness costs. In the first case, smoothness costs are given by the function with properties we defined earlier. In the second case, smoothness costs are given by a 1D array, which represents one smoothness table for all pairs of neighbors and in the third case, smoothness costs are given by 2D array, which represents $2n$ smoothness tables, i.e. two smoothness tables for each pixel as we already defined.

For "MT" version, we provide three similar constructors:

```
AlphaExpansion_2D_4C_MT(int width, int height, int n_labels, type_cost *data,
    SmoothCostFn smooth_fn, int num_threads, int block_size);

AlphaExpansion_2D_4C_MT(int width, int height, int n_labels, type_cost *data,
    type_cost *smooth, int num_threads, int block_size);

AlphaExpansion_2D_4C_MT(int width, int height, int n_labels, type_cost *data,
    type_cost **smooth, int num_threads, int block_size);
```

The only difference is in the parameter *num_threads*, which represents how many threads GridCut will use and parameter *block_size* representing the size of blocks in memory, which GridCut will allocate. See [14] for details.

For 2D grids with 8 connected neighboring system, we provide two types of constructor:

```
AlphaExpansion_2D_8C(int width, int height, int n_labels, type_cost *data, SmoothCostFn smooth_fn);

AlphaExpansion_2D_8C(int width, int height, int n_labels, type_cost *data, type_cost **smooth);
```

The first constructor is the same as we saw in 4 connected neighboring system. The second constructor is for the case of smoothness tables. Smoothness costs are given by $4n$ smoothness tables, which means four tables for each pixel as we require for this typology.

For 3D grids with 6 connected neighboring system, we also provide two types of constructor and two types of constructor fot the "MT" version:

```
AlphaExpansion_3D_6C(int width, int height, int depth, int n_labels, type_cost *data,
    SmoothCostFn smooth_fn);

AlphaExpansion_3D_6C(int width, int height, int depth, int n_labels, type_cost *data,
    type_cost **smooth);

AlphaExpansion_3D_6C_MT(int width, int height, int depth, int n_labels, type_cost *data,
    SmoothCostFn smooth_fn, int num_threads, int block_size);

AlphaExpansion_3D_6C_MT(int width, int height, int depth, int n_labels, type_cost *data,
      type_cost **smooth, int num_threads, int block_size);
```

The constructors are moreless the same as for 2D. We only added the parameter of a grid's depth and we expect different number of tables, which is $3n$. For each pixel, there has to be three tables for the neighbors by definition.

For 3D grids with 26 connected neighboring system, we also provide two types of constructor:

```
AlphaExpansion_3D_26C(int width, int height, int depth, int n_labels, type_cost *data,
    SmoothCostFn smooth_fn);

AlphaExpansion_3D_26C(int width, int height, int depth, int n_labels, type_cost *data,
    type_cost **smooth);
```

The only difference is in the number of expected tables, which is $13n$ smoothness tables. For each pixel, there has to be thirteen tables for the neighbors by definition.

Since the algorithm is sensitive to initial labeling, we allow to set it by these functions:

```
void set_labeling(type_label* labeling);

void set_labels(type_label label);
```

The first function expects an array of length n, where a label for each pixel is set. The second function allows to set the given label to all pixels. If neither of these functions is called, the initial labeling is set to $(type\_label)0$ for all the pixels.

6.1.3 Running the algorithm

To run the algorithm, we provide four similar functions:

```
void perform();

void perform(int max_cycles);

void perform_random();

void perform_random(int max_cycles);
```

If the algorithm is supposed to iterate over the labels in a fixed order, from 0 to $k-1$, we call one of the first two functions. If we set the parameter max_cycles, the algorithm stops after this number of cycles. Otherwise, it stops when no further improvement is possible. When we want to iterate over the labels in a random order, we call one of the remaining functions. The parameter meaning remains the same.

6.1.4 Getting the result

After the algorithm terminates, we can get the result of the whole labeling or we can get the label for each pixel separately by calling one of the following functions

```
type_label* get_labeling(void);

type_label get_label(int pix);

type_label get_label(int x, int y);

type_label get_label(int x, int y, int  z);
```

The first function returns the final labeling, i.e. a 1D array of n length with one value for each pixel. The second function returns a single label for the given index of pixel and the third and the fourth function return a single label too, but the parameteres are the coordinates of the pixel in a 2D or in a 3D grid.

If we need to get the final energy, we can simply call the function

```
type_energy get_energy(void);
```

6.2 Examples

To show how we use the implementation in reality, we present four examples of use, two examples for basic use of API and two examples of use in practice.

The first example show the case of a 2D grid with 4 connected neighboring system, where the smoothness costs are defined by an array of different smoothness tables for each pixel. The second example shows the case of a 3D grid with 6 connected neighboring system and define the smoothness costs in a function.

The two remaining examples shows the real application of our implementation. The first application is image segmentation and the second example is image restoration.

6.2.1 Example 1 - Smoothness Term as Multiple Tables

The first example focuses on how to work with a 2D grid of 4 connected neighboring system and how to initialize all the costs. First we present a function which fills the array of data costs:

```
void get_data_costs(unsigned char *image, unsigned char *labels, int n_labels, int width, int height,
    int *data){

    int K = 1000;
    for (int pix = 0; pix < width*height; pix++)
        for (int j = 0; j < n_labels; j++){
            if (image[pix] == labels[j]) data[pix*n_labels + j] = 0;
            else data[pix*n_labels + j] = K;
        }
}
```

The parameter *image* represents the array of image pixels, the parameter *labels* represents the array of label values and the *data* array represents the data costs. We already know the other parameters. Note that we store the specific value in $data[pix * n\_labels + j]$ as we defined.

Next, we present a function which fills the array of smoothness costs:

```
void get_smooth_costs(unsigned char *image, unsigned char *labels, int n_labels, int width, int
    height, int **smooth){

   int pix;
   for (int y = 0; y < height; y++)
       for (int x = 0; x < width; x++){

           pix = y*width + x;
           //smoothness table for the right neighbor
           smooth[2*pix] = new int[n_labels*n_labels];
           if (x < width - 1){
               for (int j = 0; j < n_labels; j++)
                   for (int k = 0; k < n_labels; k++){
                       if (j == k) smooth[2*pix][n_labels*j+k] = 0;
                       else smooth[2*pix][n_labels*j+k] = WEIGHT(img[pix]-img[pix+1]);
                   }
           //just for sure, the pixel on the right border does not have a right neighbor
           } else {
               for (int j = 0; j < nLabels*nLabels; j++){
                   smooth[2*pix][j] = 0;
               }
           }
           //smoothness table for the bottom neighbor
           smooth[2*pix+1] = new int[n_labels*n_labels];
           if (y < height - 1){
               for (int j = 0; j < n_labels; j++)
                   for (int k = 0; k < n_labels; k++){
                       if (j == k) smooth[2*pix+1][n_labels*j+k] = 0;
                       else smooth[2*pix+1][n_labels*j+k] = WEIGHT(img[pix]-img[pix+w]);
                   }
           //just for sure, the pixel on the bottom border does not have a bottom neighbor
           } else {
               for (int j = 0; j < n_labels*n_labels; j++){
                   smooth[2*pix+1][j] = 0;
               }
           }
       }
}
```

The parameters of the function are the same except for the last one. That one represents the array of smoothness costs. Note that we fill two smoothness tables for each pixel, the first table for the right neighbor is on the index $2 * pix$, where pix is pixel index and the second one for the bottom neighbor is on the index $2 * pix + 1$. The $WEIGHT$ can represent any smoothness function we showed earlier.

After we defined the functions, we can initialize the data cost array and the smoothness cost array:

```
int *data = new int[width*height*n_labels];
get_data_costs(image, labels, n_labels, width, height, data);

int **smooth = new int*[2*width*height];
get_smooth_costs(image, labels, n_labels, width, height, smooth);
```

To run the algorithm, we simply call the corresponding constructor, call the *perform* function and get the final labeling:

```
typedef AlphaExpansion_2D_4C<unsigned char, int, int> Expansion;
Expansion *exp = new Expansion(w, h, length, data, smooth);

exp->perform();

unsigned char* final_labeling = exp->get_labeling();
```

6.2.2 Example 2 - Smoothness Term as a Function

The second example shows how to use a smoothness function in practice. For this case, we show the example on a 3D grid with 6 connected neighboring system for the multithread version.

First we define the smoothness function in the following way:

```
int smoothFn(int pix1, int pix2, int label1, int label2){

    if(label1 == label2) return 0;

    int weight = WEIGHT(img[pix1]-img[pix2]);
    int p_z = pix1%d_step;
    int q_z = pix2%d_step;

    //if q is the back neighbor of p, modify the weight
    if(p_z != q_z){
        weight = weight / 2;
    }
    return weight;
}
```

The variable *d_step* has to be a global variable. Note that we distinguish the weights for the neighbors in the same depth in a grid and for the neighbors with different depth. For the data costs, we can use the same function as before.

We already know the rest of use. We call the constructor, where we set 2 threads and set the size of blocks in memory to 160x160, then we run the algorithm (in this case for random order of labels) and get the result:

```
typedef AlphaExpansion_3D_6C_MT<unsigned char, int, int> Expansion;
Expansion *exp = new Expansion(w, h, length, data, &smoothFn, 2, 160);

exp->perform_random();

unsigned char* final_labeling = exp->get_labeling();
```

6.2.3 Example 3 - Image Segmentation

In the third example, we show the real application of image segmentation described in chapter 5. The input consists of two images, the image intended for the segmentation and a mask, where the users' scribbless define different regions. The input is in figure 6.3.

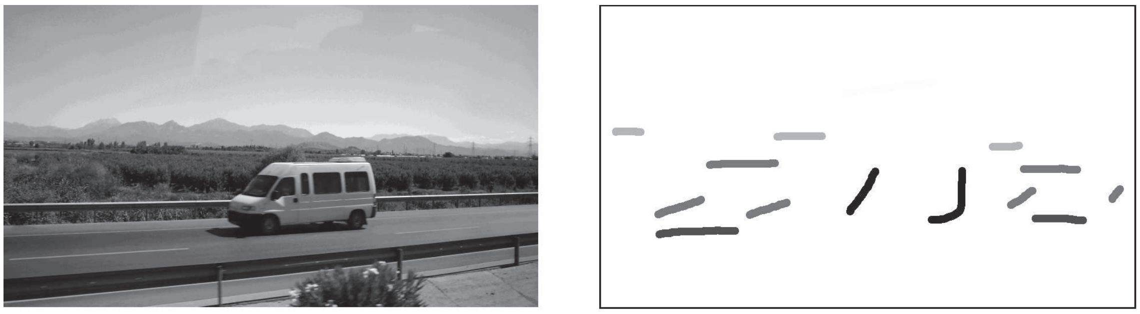
Figure 6.3: The input of image segmentation.

Now we need to define the data and smoothness costs. For data costs, we can use the same function as before:

```
void get_data_costs(unsigned char *mask, unsigned char *labels, int n_labels, int width, int height,
     int *data){

    int K = 1000;
    for (int pix = 0; pix < width*height; pix++){
        for (int j = 0; j < n_labels; j++){
            if (mask[pix] == labels[j]) data[pix*n_labels + j] = 0;
            else data[pix*n_labels + j] = K;
        }
    }
}
```

Here, the first parameter represents the mask image. If there is a scribble on a given index, the resulting cost is 0, otherwise it is a K.

For smoothness costs, we define a smoothness function and a global variable *dataImg* that represents the input image and the smoothness costs rely on:

```
unsigned char *dataImg;
int smoothFn4(int pix1, int pix2, int label1, int label2){

    if (label1 == label2) return 0;

    int weight = WEIGHT(dataImg[pix1]-dataImg[pix2]);
    return weight;
}
```

Now we can show the whole proccess of image segmentation:

```
int main(int argc, char **argv){

    //image width and height
    int w,h;

    //number of scribbless with different intensity in the mask
    int n_labels = 5;

    // intensities of the labels
    unsigned char *labels = new unsigned char[n_labels];
    labels[0] = 250; labels[1] = 180; labels[2] = 120; labels[3] = 80; labels[4] = 30;

    //reference to the global variable needed for the smoothness function
    unsigned char* &img = dataImg;

    //reading image and mask
    img = load_TGA("image.tga",w,h);
    unsigned char *msk = load_TGA("mask.tga",w,h);

    //getting data costs
    int *data = new int[w*h*n_labels];
    get_data_cost(msk, labels, w, h, n_labels, data);

    //running the algorithm
    typedef AlphaExpansion_2D_4C<unsigned char, int, int> Expansion;
    Expansion *exp = new Expansion(w, h, n_labels, data, &smoothFn4);
    exp->perform();

    //getting the result
    unsigned char* final_labeling = exp->get_labeling();

    //saving the result as an image
    unsigned char label_index;
    for (int i = 0; i < w*h; i++){

        label_index = final_labeling[i];
        img[i] = labels[label_index];
    }
    save_TGA("output.tga", img, w, h);
}
```

The code is mostly self explanatory. The only thing we should add is that the function *load_TGA* is a simple function for reading the image by its file name and it returns the image in array and updates the variables w and h. The function *save_TGA* then saves the updated image array to a file according to its file name.

The output of this example is in figure 6.4

6.2.4 Example 4 - Image Restoration

The last example is a real application of image restoration. There are two images as an input again, the image for the restoration and a mask. The input is shown in figure 6.5.

We follow the steps as before and therefore we firstly give the functions for data and smoothness costs. For this case, data costs depend on the image as well as on the mask. If there is

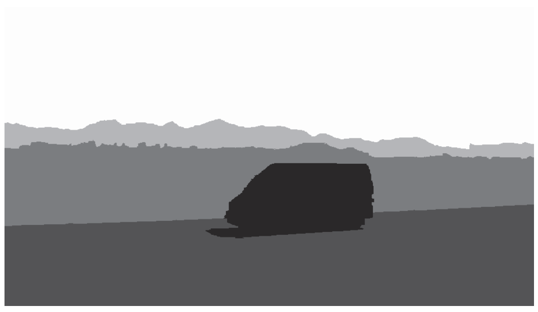
Figure 6.4: The output of image segmentation.

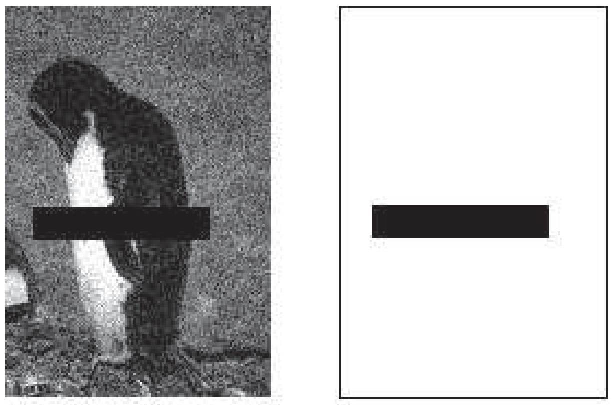
Figure 6.5: The input of image restoration.

a black pixel on a mask, it means the pixel in the image is covered and the resulting cost is 0, otherwise it is a quadratic distance of the pixel intensity from the image and a label:

```
void get_data_cost(CByteImage img, CByteImage msk, int nLabels, int width, int height, int *data){

   int index = 0;

   for (int y = 0; y < height; y++) {
      for (int x = 0; x < width; x++) {
         for (int j = 0; j < n_labels; j++){
            if(msk.Pixel(x, y, 0) == 0) data[index] = 0;
            else data[index] = (img.Pixel(x, y, 0) - j)*(img.Pixel(x, y, 0) - j);

            index++;
         }
      }
   }
}
```

The smoothness costs are given by a function for the truncated quadratic distance or the truncated absolute distance of two labels and it represents the case of one common smoothness table for all the pairs of pixels:

```
void get_smooth_cost(CByteImage img, int n_labels, int width, int height, int smoothExp, int
    smoothMax, int lambda, int *smooth){

   for (int i=0; i<n_labels; i++)
       for (int j=i; j<n_labels; j++){
           int cost = ((smoothExp == 1) ? abs(j - i) : (j - i)*(j - i));
           if (cost > smoothMax) cost = smoothMax;
           smooth[n_labels*i+j] = smooth[n_labels*j+i] = cost*lambda;
       }
}
```

The process of image resotration is then as follows:

```
int main(int argc, char **argv){

   //number of labels
   int n_labels = 256;

   //variables for the calculations of smoothness costs
   int trunc = n_labels * n_labels;
   int lambda = 1;
   int smoothExp = 2;

   //reading the image
   CByteImage img;
   ReadImageVerb(img, "image.png", 1);
   CShape sh = img.Shape();
   int width = sh.width;
   int height = sh.height;

   //reading the mask
   CByteImage mask;
   ReadImageVerb(mask, "mask.png", 1);

   //getting data costs
   int *data = new int[width*height*n_labels];
   get_data_cost(img, maskimg, nLabels, width, height, data);

   //getting smoothness costs
   int *smooth = new int[n_labels*n_labels];
   get_smooth_cost(img, n_labels, width, height, smoothExp, trunc, lambda, smooth);

   //running the algorithm
   typedef AlphaExpansion_2D_4C<unsigned char, int, int> Expansion;
   Expansion *exp = new Expansion(w, h, n_labels, data, smooth);
   exp->perform_random();

   //saving the result as an image
   CByteImage outimg(sh);
   for (int pix = 0; pix < w*h; pix++){
       outimg.Pixel(x, y, 0) = exp->get_label(pix);
   }
   WriteImage(outimg, "output.png");
}
```

The code is mostly self explanatory. The functions and data types we use for handling the images are from the library published in [2].

The resulting output is in figure 6.6.

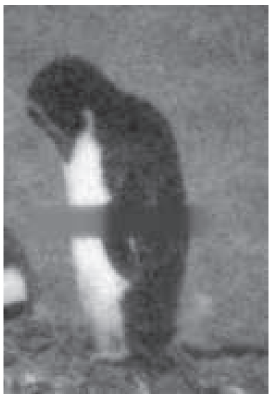

Figure 6.6: The output of image restoration.

Chapter 7

Results

To test our resulting implementation, AlphaExpansion (shortly AE), we made a set of several test cases, which contain the applications of multilabeling problem in computer graphics and vision.

For 2D grids, the data set consists of the commonly known instances from Middlebury College [3]. These instances represent the applications of the image restoration (1-2), stereo (3-5) and photomontage (6-7). Moreover, we added several test data (8-15) from LazyBrush application [22] for image coloring and our own data for image segmentation (16-20). Table 7.1 gives us the review of our test data.

#	Instance	Application	Labels	Width	Height
1	/mid/denoise/house	Restoration	256	256	256
2	/mid/denoise/penguin	Restoration	256	122	179
3	/mid/stereo/teddy	Stereo	60	450	375
4	/mid/stereo/tsukuba	Stereo	16	384	288
5	/mid/stereo/venus	Stereo	40	434	383
6	/mid/photomontage/family	Photomontage	5	752	566
7	/mid/photomontage/panorama	Photomontage	7	1071	480
8	/ctu/lazybrush/tree	Coloring	5	1026	578
9	/ctu/lazybrush/girl	Coloring	7	1026	578
10	/ctu/lazybrush/dog	Coloring	4	720	576
11	/ctu/lazybrush/man	Coloring	6	578	1026
12	/ctu/lazybrush/man2	Coloring	7	578	1026
13	/ctu/lazybrush/kid	Coloring	6	1026	578
14	/ctu/lazybrush/bottle	Coloring	3	1026	578
15	/ctu/lazybrush/mouse	Coloring	6	1026	578
16	/saidl/segment/tower	Segmentation	3	342	456
17	/saidl/segment/nature	Segmentation	4	456	342
18	/saidl/segment/car	Segmentation	5	912	514
19	/saidl/segment/head	Segmentation	4	1824	1028
20	/saidl/segment/tower	Segmentation	6	1824	1028

Table 7.1: Test data for 2D grids.

We made a set of three test cases for these instances. The first test case consists of comparison of our implementation with the GCO library, which is implemented in accordance with the same papers [10] and [17] as our implementation. We chose the latest version of GCO 2.3 from [4], which we compare by execution time and memory use.

The second test case focuses on the comparison of using different smoothness term representations, smoothness arrays and smoothness functions, and the third test case shows the comparison of using different neighboring system.

For 3D grids, the data set consists of five simple instances for 3D segmentation. Since GCO does not provide support for 3D grids, we only compared the execution times of our implementation. Specifically, we compared execution times when we used different neighboring system, 6 and 26 connected neighboring system. Table 7.2 gives the review of our test data for 3D grids.

#	Instance	Application	Labels	Width	Height	Depth
25	/saidl/segment3D/geom1	3D segmentation	4	40	40	40
26	/saidl/segment3D/geom2	3D segmentation	4	80	80	80
27	/saidl/segment3D/geom3	3D segmentation	3	120	120	120
28	/saidl/segment3D/geom4	3D segmentation	3	160	160	160
29	/saidl/segment3D/geom5	3D segmentation	3	200	200	200

Table 7.2: Test data for 3D grids.

We tested all the test cases on the computer with configuration described in table 7.3.

Processor	Intel Core i7 3517U
Frequency	2.40GHz
Cores	2
Cache	4MB
Compiler	Visual C++ Compiler
Operating system	Windows 7

Table 7.3: The computer's configuration.

7.1 Comparison with GCO

First, we compared our implementation with GCO library on 2D grids with 4 connected neighboring system, which is the only one topology supported by GCO. We tested both implemented versions, AE_2D_4C and AE_2D_4C_MT. Because GCO supports only smoothness function and one smoothness table as a smoothness term representation, we run the tests only for these representations.

We can see the detailed comparison of GCO and AE_2D_4C in table 7.4.

As we can see from the table, we used both types of smoothness representation, arrays (one smoothness table) for the first five test instances and smoothness function for the rest. Obviously, the resulting energy for each test instance is always equal for both implementations. We can mainly see the execution times for both implementations and the speed-up of AE compared to GCO. Clearly, AE is faster for all the test instances, almost three times on average.

#	Function /	Equal	GCO [s]			AlphaExpansion [s]			Speed-up	
	Array	Energy	Init	Expansion	Total	Init	Expansion	Total	Expansion	Total
1	Array	✓	0,001	638,464	638,465	0,000	342,044	342,044	1,87x	1,87x
2	Array	✓	0,000	51,004	51,004	0,000	20,968	20,968	2,44x	2,44x
3	Array	✓	0,002	40,095	40,098	0,000	11,703	11,703	3,43x	3,43x
4	Array	✓	0,002	3,904	3,906	0,000	1,536	1,536	2,55x	2,55x
5	Array	✓	0,002	18,318	18,319	0,000	6,715	6,715	2,73x	2,73x
6	Function	✓	0,005	7,879	7,884	0,000	3,754	3,754	2,1x	2,11x
7	Function	✓	0,006	13,242	13,248	0,000	6,018	6,018	2,21x	2,21x
8	Function	✓	0,007	9,033	9,040	0,001	2,941	2,941	3,08x	3,08x
9	Function	✓	0,007	11,941	11,948	0,000	4,130	4,130	2,9x	2,9x
10	Function	✓	0,005	5,813	5,818	0,000	2,272	2,272	2,56x	2,57x
11	Function	✓	0,007	13,472	13,479	0,000	4,076	4,076	3,31x	3,31x
12	Function	✓	0,007	8,578	8,585	0,000	2,585	2,585	3,32x	3,33x
13	Function	✓	0,007	9,923	9,931	0,000	2,993	2,993	3,32x	3,32x
14	Function	✓	0,007	2,655	2,662	0,000	0,772	0,772	3,44x	3,45x
15	Function	✓	0,007	9,160	9,168	0,000	2,833	2,833	3,24x	3,24x
16	Function	✓	0,002	0,421	0,423	0,000	0,143	0,143	2,95x	2,96x
17	Function	✓	0,002	0,899	0,901	0,000	0,288	0,288	3,13x	3,13x
18	Function	✓	0,006	3,511	3,517	0,000	1,072	1,072	3,28x	3,29x
19	Function	✓	0,022	28,348	28,370	0,001	10,994	10,994	2,58x	2,59x
20	Function	✓	0,023	26,142	26,165	0,000	7,566	7,566	3,46x	3,46x
										2,89x

Table 7.4: Comparison of execution times of GCO and AE_2D_4C.

Next, we tested the same instances for AE_2D_4C_MT, where we set 2 threads and 80x80 block size. Table 7.5 gives the comparison of GCO, AE_2D_4C and also AE_2D_4C_MT. Now we show only the total execution times, because the initialization times stays the same.

As we can see from the table, most of the instances are solved even faster by AE_2D_4C_MT as we expected. The execution times stays better for all the test instances compared to GCO, more than three times on average. To better see the speed-up, we show graph 7.1 of the execution times for all tested implementations. Unfortunately, the execution times of the first and the second instances are so high that we had to divide the execution times by twenty for the first instance and by two for the second one.

We also tested these implementations for memory use. We show the measured memory of GCO and AE in table 7.6. In the last two columns, we can easily see that AE uses less memory compared to GCO for all the test instances, almost twice less on average.

Again, we also provide the corresponding graph 7.2 with measured memory use to clearly see the difference.

#	Function / Array	GCO [s]	AlphaExpansion [s]		Speed-up	
			4C	4C_MT	4C	4C_MT
1	Array	638,465	342,044	301,907	1,87x	2,12x
2	Array	51,004	20,968	17,374	2,44x	2,94x
3	Array	40,098	11,703	10,123	3,43x	3,97x
4	Array	3,906	1,536	1,415	2,55x	2,77x
5	Array	18,319	6,715	5,751	2,73x	3,19x
6	Function	7,884	3,754	3,253	2,11x	2,43x
7	Function	13,248	6,018	6,002	2,21x	2,21x
8	Function	9,040	2,941	2,774	3,08x	3,26x
9	Function	11,948	4,130	3,709	2,9x	3,23x
10	Function	5,818	2,272	2,438	2,57x	2,39x
11	Function	13,479	4,076	3,240	3,31x	4,17x
12	Function	8,585	2,585	2,119	3,33x	4,06x
13	Function	9,931	2,993	3,143	3,32x	3,16x
14	Function	2,662	0,772	0,777	3,45x	3,43x
15	Function	9,168	2,833	2,458	3,24x	3,73x
16	Function	0,423	0,143	0,143	2,96x	2,96x
17	Function	0,901	0,288	0,295	3,13x	3,06x
18	Function	3,517	1,072	1,041	3,29x	3,38x
19	Function	28,370	10,994	10,205	2,59x	2,79x
20	Function	26,165	7,566	7,076	3,46x	3,7x
					2,89x	3,14x

Table 7.5: Comparison of execution times of AE_2D_4C, AE_2D_4C_MT and GCO. The last two columns represent the speed-up compared to GCO.

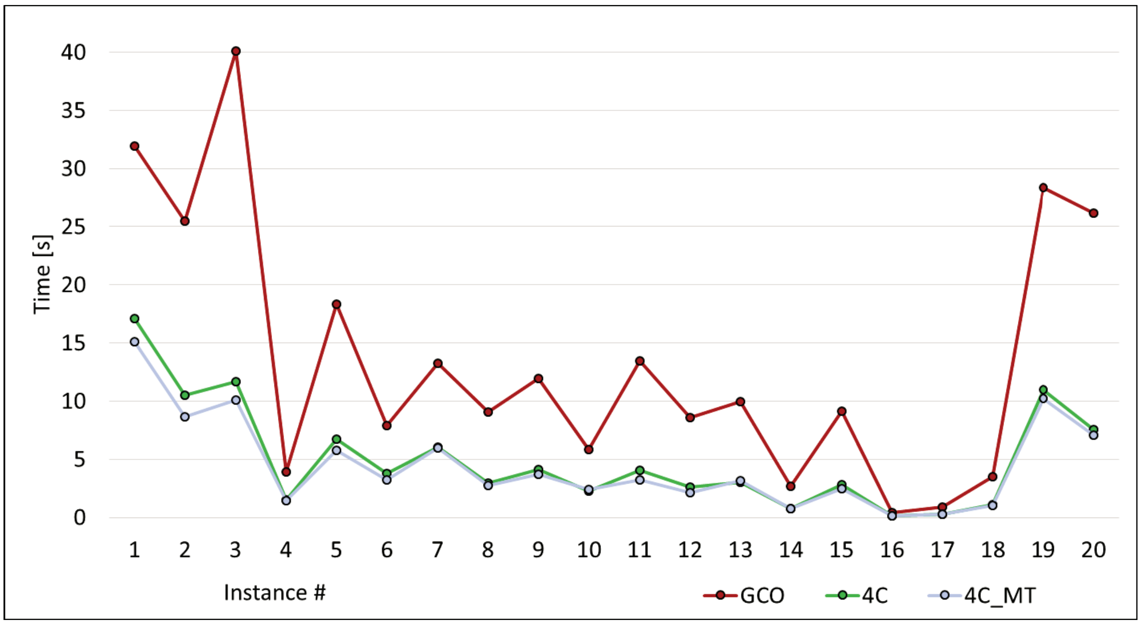

Figure 7.1: A graph showing the execution times of AE_2D_4C, AE_2D_4C_MT and GCO.

#	GCO [MB]	AlphaExpansion [MB]		Saving	
		4C	4C_MT	4C	4C_MT
1	75,42	71,78	73,25	1,06x	1,03x
2	27,95	26,88	27,27	1,04x	1,03x
3	60,09	50,71	52,76	1,19x	1,14x
4	20,62	17,04	18,13	1,21x	1,14x
5	45,66	37,35	38,82	1,23x	1,18x
6	78,38	49,54	54,63	1,59x	1,44x
7	77,57	45,04	51,64	1,73x	1,51x
8	76,35	39,71	46,35	1,93x	1,65x
9	78,61	42,79	48,61	1,84x	1,62x
10	51,46	26,35	30,61	1,96x	1,69x
11	76,36	42,78	46,38	1,79x	1,65x
12	78,59	44,14	48,68	1,79x	1,62x
13	76,34	41,00	46,37	1,87x	1,65x
14	69,53	33,18	39,57	2,1x	1,76x
15	76,36	40,27	46,34	1,9x	1,65x
16	21,54	11,67	13,66	1,85x	1,58x
17	22,13	12,27	14,27	1,81x	1,56x
18	59,29	29,20	35,63	2,04x	1,67x
19	216,55	100,84	121,37	2,15x	1,79x
20	230,86	118,91	135,73	1,95x	1,71x
				1,70x	1,55x

Table 7.6: Comparison of memory use of AE_2D_4C, AE_2D_4C_MT and GCO. The last two column shows, how many times AE uses less memory compared to GCO.

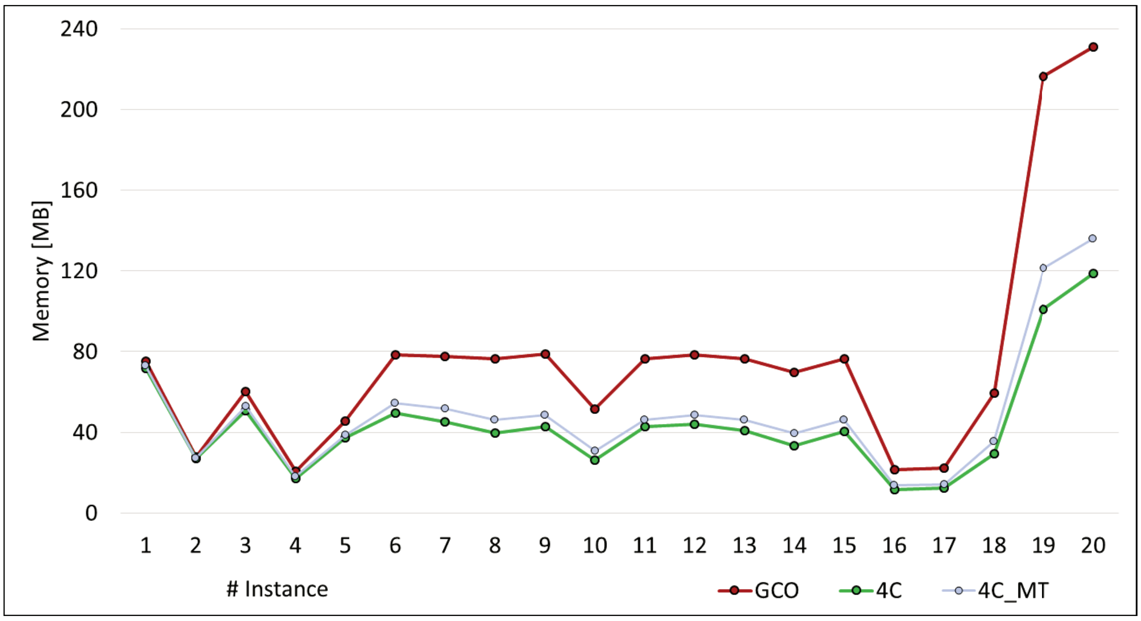

Figure 7.2: A graph showing memory use of GCO and AE_2D_4C.

7.2 Comparison of Smoothness Term Representation

In the second test case we focus on comparison of smoothness term representations. We compare the execution times for the use of smoothness arrays (smoothness tables) and for the use of smoothness function for AE_2D_4C and AE_2D_4C_MT. For the instances 1 - 5, we keep the using of one smoothness table and for the rest, we present the results of using multiple smoothness tables. We show the results in table 7.7.

#	GCO [s]	Function [s]		Arrays [s]		Speed-up		
		4C	4C_MT	4C	4C_MT	4C	4C_MT	Total
1	638,465	356,186	314,674	342,044	301,907	1,05x	1,05x	2,12x
2	51,004	23,936	20,374	20,968	17,374	1,15x	1,18x	2,94x
3	40,098	13,183	13,125	11,703	10,123	1,13x	1,3x	3,97x
4	3,906	1,731	1,601	1,536	1,415	1,13x	1,14x	2,77x
5	18,319	7,31	6,566	6,715	5,751	1,09x	1,15x	3,19x
6	7,884	3,754	3,253	2,491	2,226	1,51x	1,47x	3,55x
7	13,248	6,018	6,002	3,410	3,412	1,77x	1,76x	3,89x
8	9,040	2,941	2,774	2,886	2,676	1,02x	1,04x	3,38x
9	11,948	4,130	3,709	4,154	3,700	1x	1,01x	3,23x
10	5,818	2,272	2,438	2,156	2,341	1,06x	1,05x	2,49x
11	13,479	4,076	3,240	4,127	3,244	0,99x	1x	4,16x
12	8,585	2,585	2,119	2,615	2,181	0,99x	0,98x	3,94x
13	9,931	2,993	3,143	2,987	3,142	1,01x	1,01x	3,17x
14	2,662	0,772	0,777	0,769	0,775	1,01x	1,01x	3,44x
15	9,168	2,833	2,458	2,816	2,361	1,01x	1,05x	3,89x
16	0,423	0,143	0,143	0,111	0,098	1,29x	1,46x	4,32x
17	0,901	0,288	0,295	0,206	0,236	1,4x	1,25x	3,82x
18	3,517	1,072	1,041	0,728	0,678	1,48x	1,54x	5,19x
19	28,370	10,994	10,205	9,879	8,928	1,12x	1,15x	3,18x
20	26,165	7,566	7,076	5,735	5,406	1,32x	1,31x	4,84x
						1,17x	1,19x	3,57x

Table 7.7: Comparison of execution times of smoothness term representations. The last column represents the total speed-up of the fastest AE_2D_4C_MT using arrays compared to GCO (highlighted by the grey color).

As we can see from the table, when we use multiple smoothnes tables, the execution times are usually faster compared to the smoothness function. Thus the support of the multiple smoothnes tables further improves AE and AE_2D_4C_MT using arrays becomes the fastest version and it is more than three and a half times faster compared to GCO.

We can also see these results of total execution times in graph 7.3, where we compare GCO and our faster AE_2D_4C_MT. Again, we had to divide the resulting execution times for the first test instance by twenty and for the second test instance by two.

Altough the execution times are better for the smoothness arrays, the memory use logically increases since we have to store the arrays in memory. We show that in graph 7.4, where memory use of GCO and AE_2D_4C_MT are presented.

In result, we further improved our implementation by supporting multiple smoothness tables and thanks to this, we reached the speed-up of more than three and a half times on average by using AE_2D_4C_MT with smoothness arrays compared to GCO. However,

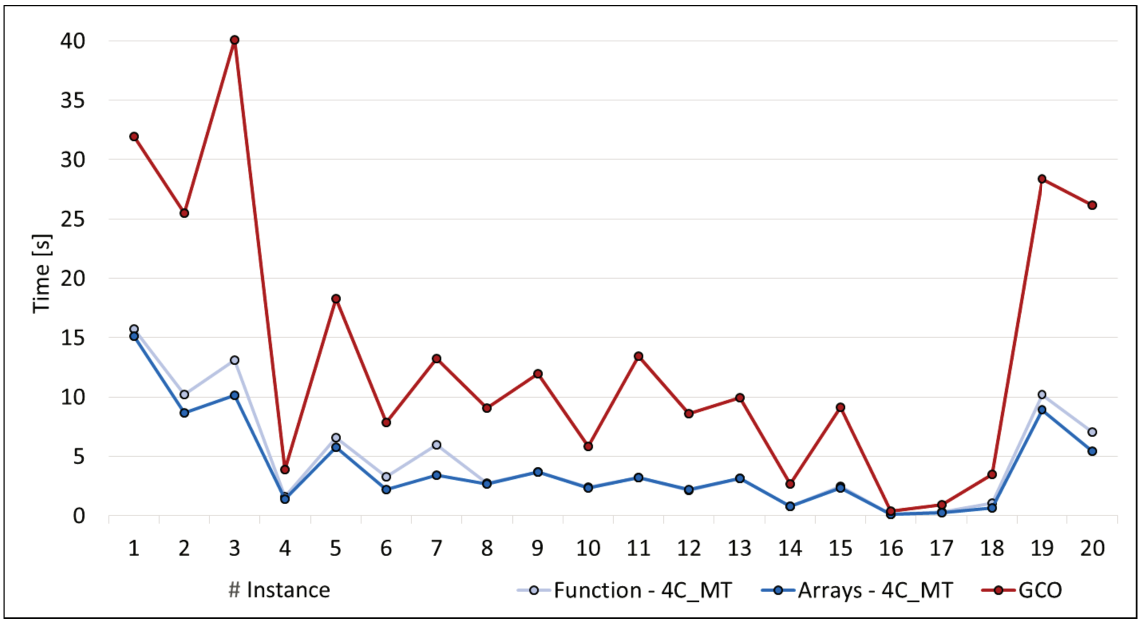

Figure 7.3: A graph showing execution times of GCO and AE_2D_4C_MT for both smoothness term representations.

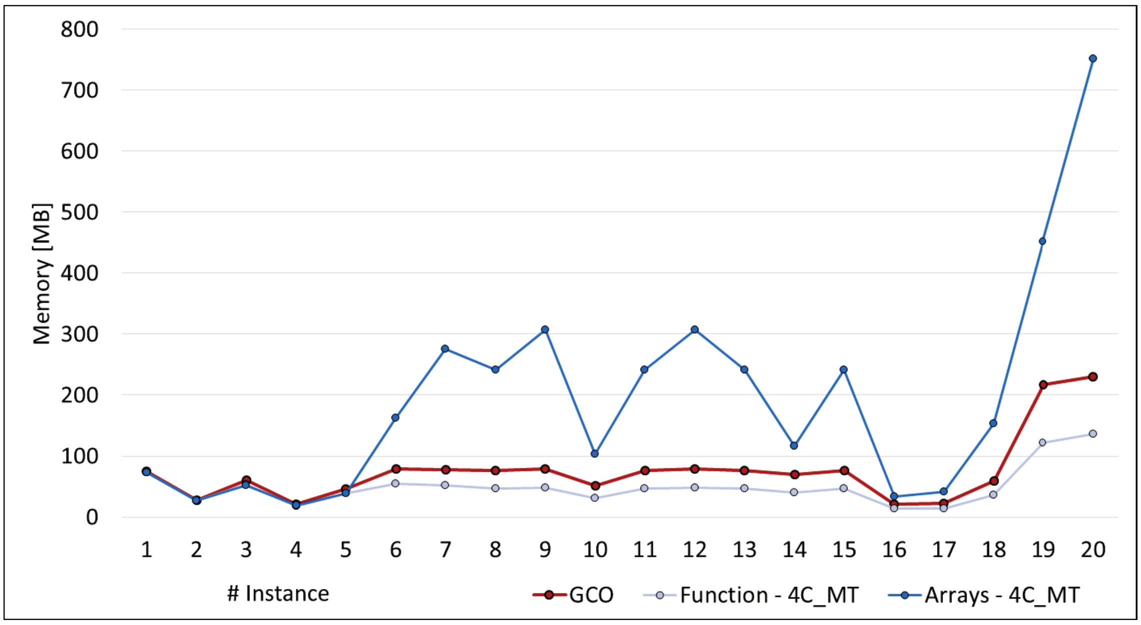

Figure 7.4: A graph showing memory use of GCO and AE_2D_4C_MT for the both smoothness term representations.

if we also need to consider the memory use, the best option is to use AE_2D_4C_MT with smoothness function.

7.3 Comparison of Neighboring Systems

The third test case focuses on the comparison of the neighboring system in 2D, specifically 4 and 8 connected neighboring systems (AE_2D_4C and AE_2D_8C). We show the results in table 7.8, where the total exuction times are presented and mainly the slowdown of the case of AE_2D_8C, which is 3.6 times on average.

#	4C [s]	8C [s]	Slowdown
1	356,186	848,915	2,4x
2	23,936	75,254	3,2x
3	13,183	68,449	5,2x
4	1,731	19,582	11,4x
5	7,31	24,028	3,3x
6	3,754	17,644	4,8x
7	6,018	22,178	3,7x
8	2,941	13,275	4,6x
9	4,130	13,489	3,3x
10	2,272	11,981	5,3x
11	4,076	8,824	2,2x
12	2,585	8,286	3,3x
13	2,993	8,647	2,9x
14	0,772	3,45	4,5x
15	2,833	8,894	3,2x
16	0,143	0,346	2,5x
17	0,288	0,512	1,8x
18	1,072	1,769	1,7x
19	10,994	17,614	1,7x
20	7,566	17,472	2,4x
			3,6x

Table 7.8: Comparison of execution times of AE_2D_4C and AE_2D_8C.

As usual, we also provide the corresponding graph 7.5, where we had to devide the execution time for the first case by twenty and for the second case by two again.

Although the execution time of 8 connected neighboring system is much worse, the resulting labeling is usually better. We show an example from the application of coloring in figure 7.6. The difference of the labeling is shown in a white zoomed rectangle. While there are some red and yellow pixels in the trunk in case of 4 connected neighboring system, they disappeared in case of 8 connected neighboring system.

As a result, the labelings are slightly better from the AE_2D_8C, but due to the higher execution times, it is suitable and sufficient to use AE_2D_4C in most cases.

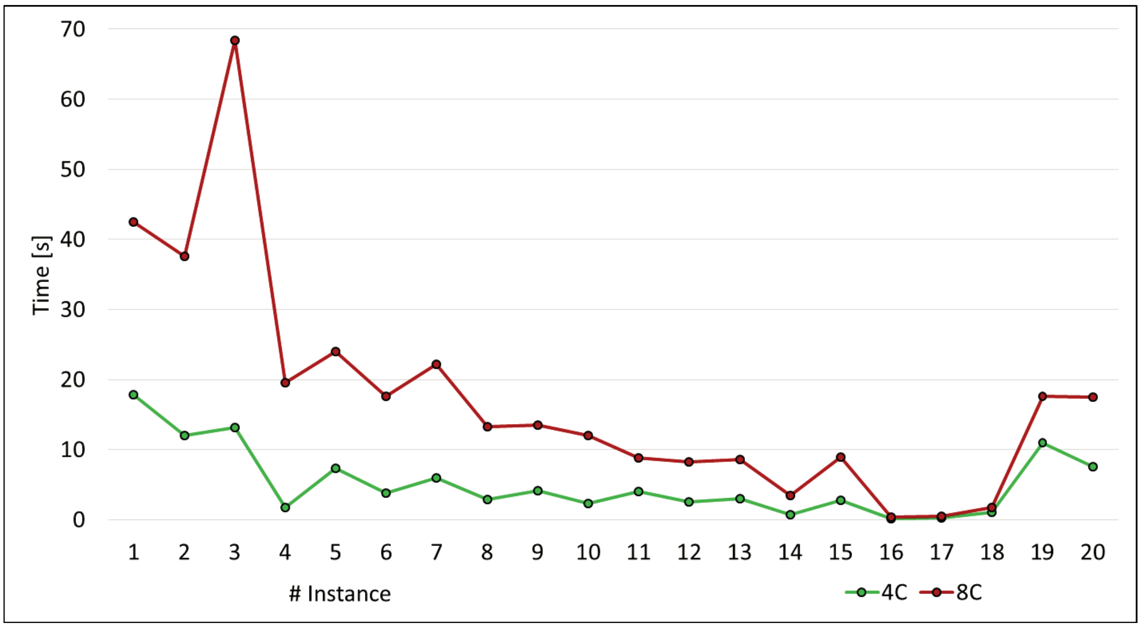

Figure 7.5: A graph showing the execution times of AE_2D_4C and AE_2D_8C.

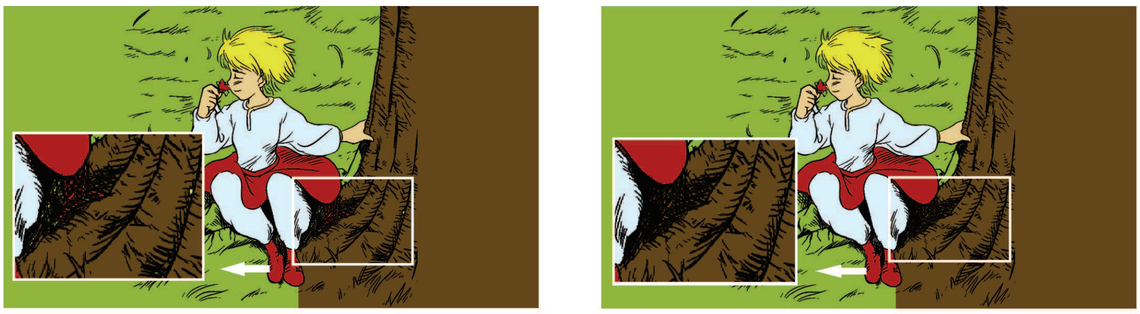

Figure 7.6: Comparison of coloring for AE_2D_4C on the left and for AE_2D_8C on the right.

7.4 Comparison of 3D Topologies

The last test case focuses on the comparison of the neighboring systems in 3D, specifically 6 and 26 connected neighboring systems (AE_3D_6C, AE_3D_6C_MT and AE_3D_26C). We show the results in table 7.9, where the total exuction times and comparison of the implementations are presented.

#	Function / Array	AlphaExpansion [s]			Slowdown of 26C	
		6C	6C_MT	26C	6C	6C_MT
25	Function	0,283	0,283	0,904	3,2x	3,2x
26	Function	2,488	2,493	7,844	3,16x	3,15x
27	Function	5,576	4,238	19,258	3,46x	4,55x
28	Function	13,757	11,054	47,568	3,46x	4,31x
29	Function	29,320	26,210	97,499	3,33x	3,72x
					3,32x	3,78x

Table 7.9: Comparison of execution times of AE_3D_6C, AE_3D_6C_MT and also AE_3D_26C.

We can see that AE_3D_6C_MT is faster than AE_3D_6C as we expected. In case of AE_3D_26C, the execution times are naturally higher, almost four times compared to AE_3D_6C_MT. As usual, we also provide the corresponding graph 7.7, where we can better see the results.

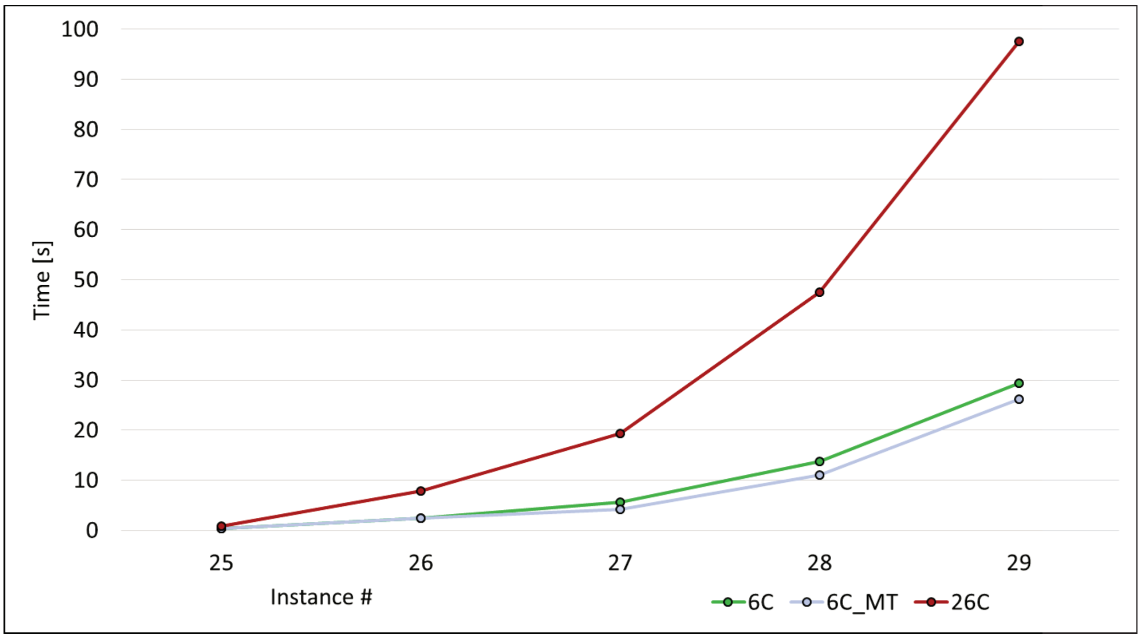

Figure 7.7: A graph showing the execution times of AE_3D_6C, AE_3D_6C_MT and AE_3D_26C.

Chapter 8

Conclusion

The goal of the thesis was to implement the expansion move algorithm solving multi-label energy minimization, which is based on iterative solving min-cut problem. By the implementation of the algorithm, we should have extended the GridCut library and thus enabled its native using for solving multi-label energy minimization.

We succesfully implemented the expansion move algorithm as an extension of GridCut library for commonly used 2D grids with 4 connected neighboring system. Moreover, we also applied the algorithm for other graph topologies, 2D grids with 8 connected neighboring system and 3D grids with 6 and 26 connected neighboring system. For 2D grids with 4 connected neighboring system and 3D grids with 6 connected neighboring system, we provide two versions of the implementation, one of them is linked to the sequential version of GridCut, while the second one is linked to the multithread version. We call the resulting implementation AlphaExpansion.

To test our implementation, we made a set of four test cases that mainly focused on comparison of our implementation with the GCO library and also on comparison of all the supported topologies, their properties and adventages.

The first test case proved that our both implementations for 2D grids with 4 connected neighboring system are much more faster compared to the GCO library, the multithread version is more than three times faster on average and also the memory use is lower compared to GCO, less than one and a half times on average.

The second test case focused on comparison of a smoothness term representation for 2D grids with 4 connected neighboring system. We found out that the execution times for multiple smoothness tables were even better compared to the execution times for the smoothness function. Since GCO does not support multiple smoothness tables, it further improved our implementation compared to GCO and we reached the speed-up of more than three and a half times on average for the multithread version. However, storing multiple smoothness tables significantly increases the memory use, so if we also need to consider memory use, the best option is to use multithread version with the smoothness function.

The third test case dealt with comparison of different 2D topologies. When we used 8 connected neighboring system for the same test instances, the execution times were logically higher compared to the case of using 4 connected neighboring system, but we were able to reach slightly better labelings. Thus the use of specific neighboring system depends on the fact, if we prefer the speed or the labeling's quality.

The last test case focused on comparison of 3D topologies, where the results were very similar to the results from 2D. When we compared the sequential and the multithread version for 6 connected neighboring system, we reached the better execution times by using multithread version. When we used 26 connected neighboring system for the same test instances, the execution times were logically higher, almost four times on average compared to the multithread version for 6 connected neighboring system.

Our resulting implementation has significant benefits for applications of computer graphics and vision, because it solves the problem of multi-label energy minimization in the shortest time ever and moreover, it supports other grid topologies and multiple smoothness tables. The future work could focus on the algorithm extension by the label cost support or adaptive cycles.

Bibliography

[1] GridCut. <`http://gridcut.com/`>.

[2] ImageLib. <`http://wohlberg.net/public/software/misc/imagelib/`>.

[3] MRF Minimization Benchmark. <`http://vision.middlebury.edu/MRF/`>.

[4] Code for Optimization with Graph Cuts. <`http://www.csd.uwo.ca/~olga/code.html`>.

[5] BESAG, J. On the statistical analysis of dirty pictures. *Journal of the Royal Statistical Society.* 1986, s. 259–302.

[6] BHUSNURMATH, A. *Applying convex optimization techniques to energy minimization problems in computer vision.* PhD thesis, University of Pennsylvania, 2008.

[7] BOYKOV, Y. – JOLLY, M.-P. Interactive graph cuts for optimal boundary & region segmentation of objects in ND images. In *Proceedings of IEEE International Conference on Computer Vision*, 1, s. 105–112, 2001.

[8] BOYKOV, Y. – KOLMOGOROV, V. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 2004, 26, s. 1124–1137.

[9] BOYKOV, Y. – VEKSLER, O. Graph cuts in vision and graphics: Theories and applications. In *Handbook of mathematical models in computer vision*. 2006. s. 79–96.

[10] BOYKOV, Y. – VEKSLER, O. – ZABIH, R. Fast Approximate Energy Minimization via Graph Cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence.* 2001, 23, s. 1222–1239.

[11] FORD, L. – FULKERSON, D. R. *Flows in networks.* 1962. 1962.

[12] GLOCKER, B. et al. Deformable medical image registration: Setting the state of the art with discrete methods. *Annual review of biomedical engineering.* 2011, 13, s. 219–244.

[13] ISHIKAWA, H. Exact optimization for Markov random fields with convex priors. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 2003, 25, s. 1333–1336.

[14] JAMRIŠKA, O. – SÝKORA, D. – HORNUNG, A. Cache-efficient Graph Cuts on Structured Grids. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, s. 3673–3680, 2012.

[15] KARGER, D. R. et al. Rounding algorithms for a geometric embedding of minimum multiway cut. *Mathematics of Operations Research*. 2004, 29, s. 436–461.

[16] KAV-VENAKI, E. – PELEG, S. Feedback Retargeting. In *Media Retargeting Workshop at ECCV*, 2010.

[17] KOLMOGOROV, V. – ZABIN, R. What energy functions can be minimized via graph cuts? *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 2004, 26, s. 147–159.

[18] LEMPITSKY, V. – ROTH, S. – ROTHER, C. FusionFlow: Discrete-continuous optimization for optical flow estimation. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, s. 1–8, 2008.

[19] LIU, J. – SUN, J. Parallel graph-cuts by adaptive bottom-up merging. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, s. 2181–2188, 2010.

[20] PRITCH, Y. – KAV-VENAKI, E. – PELEG, S. Shift-Map Image Editing. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, s. 151–158, 2009.

[21] SÝKORA, D. – DINGLIANA, J. – COLLINS, S. As-rigid-as-possible Image Registration for Hand-drawn Cartoon Animations. In *Proceedings of International Symposium on Non-photorealistic Animation and Rendering*, s. 25–33, 2009.

[22] SÝKORA, D. – DINGLIANA, J. – COLLINS, S. LazyBrush: Flexible Painting Tool for Hand-drawn Cartoons. *Computer Graphics Forum*. 2009, 28, s. 599–608.

[23] SZELISKI, R. et al. A comparative study of energy minimization methods for markov random fields. In *Proceedings of European Conference on Computer Vision*. 2006. s. 16–29.

[24] VEKSLER, O. *Efficient graph-based energy minimization methods in computer vision*. PhD thesis, Cornell University, 1999.

[25] Y. PRITCH, Y. P. – PELEG, S. Snap Image Composition. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 6930, s. 181–191, 2011.

Appendix A

List of Abbreviations

1D	One-dimensional space
2D	Two-dimensional space
3D	Three-dimensional space
AE	AlphaExpansion
API	Application programming interface
MT	Multithread

Appendix B

Content of the Attached CD

```
CD
 |-- data                         - Test data
 | |--ctu                           - Test data from LazyBrush
 | | |--lazybrush                     - Test data for coloring
 | |--mid                           - Test data from Middlebury College
 | | |--denoise                       - Test data for image restoration
 | | |--photomontage                  - Test data for photomontage
 | | |--stereo                        - Test data for stereo
 | |--saidl                         - Our own test data
 | | |--segment                       - Test data for image segmentation
 | | |--segment3D                     - Test data for 3D segmentation
 |-- gco                          - GCO library and test projects
 | |--coloring                      - Project for coloring
 | |--denoise                       - Project for restoration
 | |--GCOv2p3                       - GCO 2.3 library
 | |--imageLib                      - imageLib - library handling PNG images
 | |--photomontage                  - Projects for photomontage
 | |--segmentation                  - Project for segmentation
 | |--stereo                        - Project for stereo
 |-- gridcut                         - code and test projects
 | |--examples                      - Test projects of AlphaExpansion
 | | |--coloring                      - Project for coloring
 | | |--denoise                       - Project for restoration
 | | |--photomontage                  - Project for photomontage
 | | |--segmentation                  - Project for segmentation
 | | |--segmentation3D                - Project for segmentation
 | | |--stereo                        - Project for stereo
 | |--include                       - Required libraries
 | | |--AlphaExpansion                - AlphaExpansion library
 | | |--GridCut                       - GridCut library
 | | |--imageLib                      - imageLib - library handling PNG images
 |-- measurements               - Measurements of all the test cases
 | |--ctu                           - Measurements of data from LazyBrush
 | | |--lazybrush                   - Measurements of coloring
 | |--mid                           - Measurements of data from Middlebury College
 | | |--denoise                       - Measurements of restoration
 | | |--photomontage                  - Measurements of photomontage
 | | |--stereo                        - Measurements of stereo
 | |--results                       - Results of measurements
 | |--saidl                         - Measurements of our own data
 | | |--segment                       - Measurements of segmentation
 | | |--segment3D                     - Measurements of 3D segmentation
 |-- text                       - Text of this thesis
```