Efficient Graph Cuts for Multi-region Segmentation

Martin Rykfors

Contents

1 Introduction **1**

2 Flow Algorithms in Vision **3**
2.1 Graphs and flow networks 3
2.1.1 Definitions 3
2.1.2 Flow Networks 5
2.1.3 Maximum flows and minimum cuts 6
2.2 Max-Flow algorithms 9
2.2.1 Maximum flows through augmenting paths 9
2.2.2 Boykov-Kolmorogov's algorithm 15
2.2.3 Incremental Breadth First Search 18
2.3 Image segmentation 20
2.3.1 Introduction 20
2.3.2 The general setting 22
2.3.3 Multi-region energy 23

3 Implementation Aspects **25**
3.1 A generic implementation 25
3.2 Using the grid structure 26
3.2.1 Compact residual representation 26
3.2.2 Cache friendly indexing of the nodes 28
3.2.3 Structure splitting 29
3.2.4 Handling the grid boundary 30

4 The Multi-region Implementation **31**
4.1 The multi-region grid 31
4.2 Node enumeration in 3D 32
4.3 Handling the layer connectivity 32
4.4 IBFS variant: the rank-relabel step 33

5 Results and Evaluation **35**
5.1 A multi-region example 35
5.2 Speed comparison with other solvers 36
5.3 Memory comparison with other solvers 40

6 Conclusion **41**

Abstract

Many problems in computer vision can be formulated as optimization problems that can be solved using minimum cuts in flow networks. However, these applications can be demanding, with the memory bandwidth being the main bottleneck on the performance. This thesis introduces an efficient solver that reduces the memory requirements while improving the speed. The code is freely available on-line.

Acknowledgements

I would like to thank my supervisors Petter Strandmark and Johannes Ulén for their help and guidance through the underlying theory and through all the intricacies of programming at this level. I also want to thank my family and friends for their support and for keeping me motivated. Finally, I want to thank `std::cout` for always letting me know what's up.

Chapter 1

Introduction

This thesis project was done in the autumn semester 2012 at the Mathematical Imaging group at Lund Institute of Technology. The main theme is that of minimization of functions of the form

$$E(\mathbf{x}) = \sum_i E_i(x_i) + \sum_{j<i} E_{ij}(x_i, x_j), \tag{1.1}$$

where $\mathbf{x} \in \{0,1\}^N$, a vector of N binary variables. Minimizing such a function is in general a difficult problem, requiring heavy calculations or the use of approximate solvers. Interest in these kinds of functions grew when it was discovered that in certain cases an optimal solution can be found using graph cut techniques. The article [9] showed exactly under what conditions we are able to use graph cuts for minimization of these kinds of functions.

The main idea of these techniques is that we construct a *graph*, a kind of network of nodes and connections, on which we use fast algorithms to determine the so-called *minimum cut*. This minimum cut will then correspond to a global minimizer of Equation (1.1). The algorithms for finding these cuts usually do this by solving the dual problem – finding a *maximum flow* in the graph.

A big problem is the representation of such graphs. In many cases they are represented as pointer-heavy data structures. These pointers will in turn take up a lot of memory. For example, in medical image segmentation, one may deal with high-resolution 3D data, and the graphs constructed may need tens of gigabytes of memory, a big majority of which is made up of pointers for representing the connectivity of the graph. Can we find a better representation of the graphs used in these applications? As it turns out, the graphs used in vision instances follow a highly regular structure that we can take advantage of, greatly reducing memory consumption by removing the need to maintain such a large number of pointers in memory. This will also open up the possibility for many memory-related optimizations for speed.

A common use for minimum cuts in vision is for *image segmentation*. This is often used to split an image into two parts, for instance one segment representing the foreground and the other segment representing the background. For each segment we have a model, and we can evaluate how well each pixel conforms to the two models. The idea is to assign each pixel to one of the two segments according to how well they fit the models, but also in such a way that the

boundary between the segments is kept smooth. The behaviour of the boundary is governed by a *regularization parameter* that can be seen as a penalty for every discontinuity in the segmentation. By increasing this parameter, the segmentation is made smoother and starts rejecting small details that could be the result of errors caused by noise.

The goal of this thesis has been to write a memory efficient minimum cut solver for *multi-region segmentation* instances. This allows us to split the image into several regions that interact geometrically, for instance forcing two regions to be mutually exclusive, or forcing one region to be within another. The benefit of using multi-region segmentation is that it allows us to split an image into more than two parts through a single minimum cut.

The language of choice has been C++, as the implementation requires manual memory management. The popularity of the language in the vision community has also been a factor.

This thesis is structured as follows: The next section introduces all the theory needed; graphs, flow networks, the max flow/min cut theorem. It will also introduce some of the algorithms for computing the max flow. After that we will look at energy minimization and how we can use graph cuts for solving such problems. Finally, we will look at a specific class of energy functions – the multi-region energy functions.

The third section covers how we can implement graph cut solvers in an efficient way, reducing memory consumption and improving performance, all by drawing advantage from the regular structure of the graph.

The fourth section describes the implementation, called MRGraph, that was written in this thesis project. The graphs for multi-region segmentation have a similar, regular structure as the ones discussed in Section 3, but there are some differences that need to be taken care of if we want to put the memory saving techniques to use.

The fifth section shows some experimental results, evaluating the speed and memory requirements of the implementation contra other graph cut solvers.

Chapter 2

Flow Algorithms in Vision

The path to understanding multi-region segmentation is a long one, with a lot of theory needed to be taken care of. Those already familiar with graphs could simply skim the first part of this section, but I recommend familiarizing oneself with the notation that will be used. If you are already familiar with flow networks and their related algorithms, I recommend skipping to the section about the Boykov-Kolmorogov and IBFS algorithms, as they are quite recent developments and may be novel for some readers.

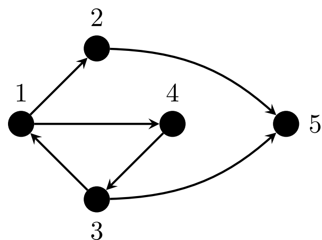
2.1 Graphs and flow networks

2.1.1 Definitions

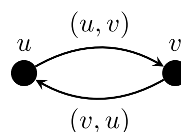
We begin by providing a definition for a graph.

Definition 1 (Graph)**.** *A graph G is a pair $G = (V, E)$, where V is called the set of nodes and E is called the edge set consisting of pairs $(u, v), \quad u, v \in V$.*

We can represent small graphs using *graph drawings* where every node is represented by a dot and an edge $e = (u, v)$ is represented by an arrow between the dots corresponding to the nodes u and v. Figure 2.1a shows an example



(a) Example of a graph. Directed edges are drawn as arrows.



(b) We differentiate between the edge (u, v) and the edge (v, u).

Figure 2.1: Examples of graphs.

of this. In this thesis we will exclusively focus on *directed* graphs, meaning the edge $e_1 = (u, v)$ is not the same as the edge $e_2 = (v, u)$. An edge (u, v) is seen as a connection starting at its *tail node* v and ending at the *head node* u (Figure 2.1b).

Before we move on to defining flow networks, we expand a bit on the theory of graphs by providing some definitions that are needed to develop the theory of flow networks and the Max-flow/Min-cut problem. We start by looking at graph cuts:

Definition 2 (Graph cut)**.** *Let $G = (V, E)$ be a graph. A graph cut is a splitting of V into two disjoint subsets S, T where $S \cup T = V$ and $S \cap T = \emptyset$.*

In the context of flow networks, graph cuts where each part is forced to contain one specific node are of specific interest to us. We make an extension of the definition of graph cuts:

Definition 3 (s-t separating graph cut)**.** *A graph cut S, T is said to be s-t separating if $s, t \in V$, $s \in S$, $t \in T$.*

As we will see later, if we have some graph cut S, T it will be important to access the edges leaving S and entering T. In general, if V' is some subset of the set of nodes, what edges begin in V and end in $V \setminus V'$? The following notation will help us.

Definition 4 (Outgoing and incoming edges)**.** *We define the maps $\delta_+, \delta_- : \mathbb{P}(V) \to \mathbb{P}(E)$ as*

- $\delta_+(V') = \{(u, v) \in E \,|\, u \in V, v \in V \setminus V'\}$
- $\delta_-(V') = \{(u, v) \in E \,|\, v \in V, u \in V \setminus V'\}$

for any $V' \subset V$. $\mathbb{P}(X)$ is defined as the set of all subsets of the set X.

For some subset of nodes V', the set $\delta_+(V')$ are all edges that start in V' but do not end there and vice versa for $\delta_-(V')$. We extend the notation to include single nodes, that is, if $v \in V$, $\delta_+(v) = \delta_+(\{v\})$. The same applies for δ_-.

Definition 5 (Value of a graph cut)**.** *Let $G = (V, E)$ be a graph, and S, T be a graph cut. If f is a map from the edge set to $\mathbb{R}$, we define the value of the cut to be*

$$\sum_{e \in \delta_+(S)} f(e). \tag{2.1}$$

With all of these definitions established we are now ready to move on to flow networks and eventually, the central theorem of flow networks – the Max-flow/Min-cut theorem.

2.1.2 Flow Networks

Definition 6 (Flow network)**.** *A flow network is a tuple* (G, s, t, c) *where*

- $G = (V, E)$ *is a graph,*
- $s, t \in V, s \neq t$
- c *is a map* $c : E \to \mathbb{R}$.

The nodes s and t are called the *source* and the *sink* respectively and the function c is called the *capacity function*.

Imagine the graph as a collection of pipes going between pairs of nodes. Through each pipe e there can flow a maximum of $c(e)$ units of water in one direction. If water is poured into the source node s, what is the maximum flow of water that can be transported to the sink node t? This is an intuitive way of thinking of what is known as the *Max-flow* problem. To approach this formally, we need to define what a flow is:

Definition 7 (Flow)**.** *Let* (G, s, t, c) *be a flow network. A flow is a function* $f : E \to \mathbb{R}$ *that satisfies*

- $0 \leq f(e) \leq c(e)$ *for all edges* $e \in E$ *(feasibility condition),*
- $$\sum_{e \in \delta_-(v)} f(e) = \sum_{e \in \delta_+(v)} f(e),$$
  for all nodes $v \in V \setminus \{s, t\}$ *(law of conservation).*

The first condition claims that a flow never exceeds the capacity of an edge. The second condition claims that nothing is added or removed to the flow at the nodes. The amount of flow entering each node is exactly the amount that leaves the node. Of course this does not hold at the source and the sink, where the flow enters and leaves the network respectively. Figure 2.2 shows an example of a flow network with a flow. In illustrations of flow networks, capacities are written in black and flow values are written in blue.

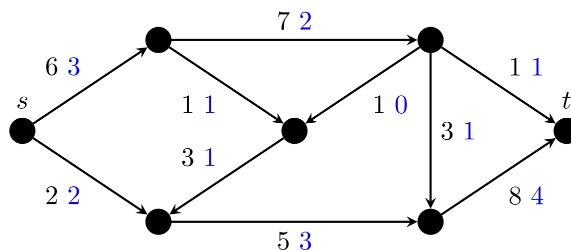


Figure 2.2: An example of a flow network with a flow. The capacities and flow values for every edge is printed using black and blue respectively. As an exercise it can be verified that the flow fulfils the feasibility condition and conservation law.

We define the *value* of a flow f as the sum of the flows leaving the source s minus the flows entering it.

$$\mathrm{val}(f) = \sum_{e \in \delta_+(s)} f(e) - \sum_{e \in \delta_-(s)} f(e) \tag{2.2}$$

A *maximum flow* is a flow for which there exists no flow with a higher value.

2.1.3 Maximum flows and minimum cuts

Lemma 1. *Let f be a flow in the flow network (G, s, t, c). If S, T is a s-t separating graph cut, then the following holds:*

$$\mathrm{val}(f) = \sum_{e \in \delta_+(S)} f(e) - \sum_{e \in \delta_-(S)} f(e). \tag{2.3}$$

We will prove this by induction. If we extend S by one node z we will see that the conservation law causes the above expression to hold also for the set $S \cup \{z\}$. Figure 2.3 illustrates how things are set up in the lemma.

Proof. Let $z \in T \setminus \{t\}$ and denote the set $Q = T \setminus \{z\}$. Then the following holds

$$\sum_{e \in \delta_+(S \cup \{z\})} f(e) = \sum_{e \in \delta_+(S)} f(e) + \sum_{e \in \{z \to Q\}} f(e) - \sum_{e \in \{S \to z\}} f(e) \tag{2.4}$$

and

$$\sum_{e \in \delta_-(S \cup \{z\})} f(e) = \sum_{e \in \delta_-(S)} f(e) + \sum_{e \in \{Q \to z\}} f(e) - \sum_{e \in \{z \to S\}} f(e), \tag{2.5}$$

where the set $\{A \to B\}$ is defined as all edges starting in A and ending in B. Putting this together we get that

$$\begin{aligned} \sum_{e \in \delta_+(S \cup \{z\})} f(e) - \sum_{e \in \delta_-(S \cup \{z\})} f(e) = & \sum_{e \in \delta_+(S)} f(e) - \sum_{e \in \delta_-(S)} f(e) \\ & + \left(\sum_{e \in \{z \to Q\}} f(e) + \sum_{e \in \{z \to S\}} f(e) \right) \\ & - \left(\sum_{e \in \{S \to z\}} f(e) + \sum_{e \in \{Q \to z\}} f(e) \right). \end{aligned} \tag{2.6}$$

By noting that

$$\sum_{e \in \{z \to Q\}} f(e) + \sum_{e \in \{z \to S\}} f(e) = \sum_{e \in \delta_+(z)} f(e) \tag{2.7}$$

and

$$\sum_{e \in \{S \to z\}} f(e) + \sum_{e \in \{Q \to z\}} f(e) = \sum_{e \in \delta_-(z)} f(e) \tag{2.8}$$

equation (2.6) becomes

$$\begin{aligned}\sum_{e\in\delta_+(S\cup\{z\})} f(e) - \sum_{e\in\delta_-(S\cup\{z\})} f(e) &= \sum_{e\in\delta_+(S)} f(e) - \sum_{e\in\delta_-(S)} f(e) \\ &\quad + \sum_{e\in\delta_+(z)} f(e) - \sum_{e\in\delta_-(z)} f(e) \\ &= \sum_{e\in\delta_+(S)} f(e) - \sum_{e\in\delta_-(S)} f(e),\end{aligned} \tag{2.9}$$

since

$$\sum_{e\in\delta_+(z)} f(e) - \sum_{e\in\delta_-(z)} f(e) = 0 \tag{2.10}$$

by the law of conservation.

Now, by starting out with $S = \{s\}$, the expression holds according to the definition of $\text{val}(f)$. Therefore we can design any graph cut S, T by starting out this way and adding new nodes one by one to S without the expression (2.3) being violated. □

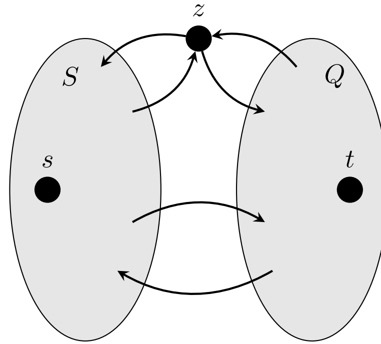


Figure 2.3: The setting in the lemma. By extending the set S with one node z, the set of outgoing edges is altered. All the connections from S to z are lost as outgoing edges, but the edges from z to Q are gained. The incoming edges are affected similarly.

A graph cut being s-t separating is a fundamental property when studying flow networks. Therefore we will from now on always mean that a graph cut S,T has this property.

Corollary 1. *The value of a maximum flow is upper bounded by*

$$\max_f \text{val}(f) \le \min_{S,T} \sum_{e\in\delta_+(S)} c(e) \tag{2.11}$$

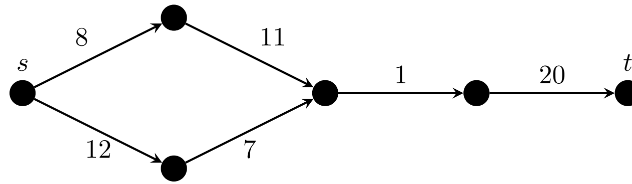


Figure 2.4: It is quite clear that the edge with capacity 1 is the bottleneck in this flow network.

Proof. For any graph cut S, T and any flow f the following holds:

$$\begin{aligned} \mathrm{val}(f) &= \sum_{e \in \delta_+(S)} f(e) - \sum_{e \in \delta_-(S)} f(e) \\ &\leq \sum_{e \in \delta_+(S)} f(e) \\ &\leq \sum_{e \in \delta_+(S)} c(e), \end{aligned} \tag{2.12}$$

since $0 \leq f(e) \leq c(e)$. Thus it must also hold for any maximum flow f and any minimum cut S, T. □

All of this might seem abstract, but there is a lot of intuition behind it. Since we cannot have a flow that exceeds the capacity of all outgoing edges of any S, the outgoing edges of a minimum cut S, T represent the *tightest bottleneck* in the flow network. Consider Figure 2.4. It is obvious that no flow can exceed the value 1 because of the single edge with capacity 1, despite all the other edges having a comparatively high capacity. Thus the graph cut S, T where S consists of all the nodes left of this edge is the minimum cut of this flow network.

One question remains: If we have a maximum flow f, does there exist a cut with exactly that capacity? Also, given a minimum cut S, T, is there a maximum flow with equal value? We let the following theorem summarize this discussion.

Theorem 1 (The Max-flow/Min-cut Theorem)**.** *In a flow network* (G, s, t, c) *the following holds:*

$$\max_f \mathrm{val}(f) = \min_{S,T} \sum_{e \in \delta_+(S)} c(e) \tag{2.13}$$

We will save the proof for now and instead discuss algorithms for finding a maximum flow.

2.2 Max-Flow algorithms

2.2.1 Maximum flows through augmenting paths

Central to this work is the finding of a maximum flow given some flow network. The algorithms we will consider are all based on finding flows through *augmenting paths*. The basic idea is to find paths in the flow network along which we can increase the flow without violating the feasibility condition.

Definition 8 (Path)**.** *A path in a graph* $G = (V, E)$ *is a sequence of edges* $e_1, e_2 \dots e_n$ *where the edge* e_t *ends where* e_{t+1} *begins.*

Definition 9 (Saturated edge)**.** *We say that an edge* e *in a flow network is saturated if* $f(e) = c(e)$*.*

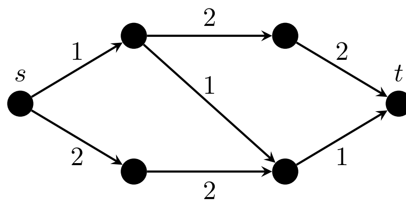
In drawings of flow networks, we will draw saturated edges extra thick from now on.

A naive approach to designing a max-flow algorithm based on this idea would be to look exclusively for s-t paths along unsaturated edges in the network and then increasing the flow along this path as much as possible:

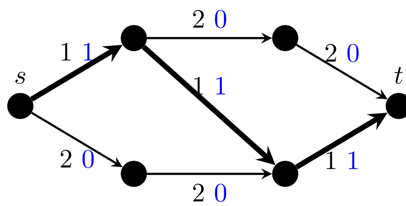
Algorithm 1: The naive algorithm

```
input  : Flow network (G, s, t, c)
output: Maximum flow f
f(e) ← 0 for all edges e in G
while there exists an unsaturated s-t path P in G do
    r ← ∞
    foreach edge e in P do
        r ← min(r, c(e) − f(e))
    foreach edge e in P do
        f(e) ← f(e) + r
return f
```

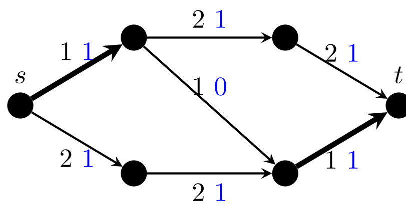
Unfortunately, this does not suffice. This is demonstrated in Figures 2.5a to 2.5c. The first augmentation to the flow ends up blocking all further possible s-t paths even though the flow is suboptimal (Figure 2.5b). The flow we wish to achieve is instead shown in Figure 2.5c.



(a) Example of a flow network on which we test the naive algorithm.



(b) The naive algorithm finds a path along the diagonal edge in the center. After augmenting along this path there no longer exists any path along unsaturated edges from s to t, yet the found flow is still not maximum.



(c) A maximum flow has value 2 in this flow network.

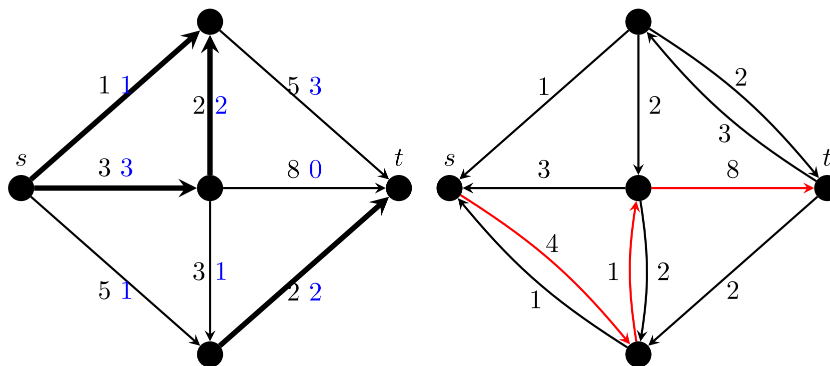
Figure 2.5: A demonstration of why the naive algorithm fails.

We need to augment the notion of an augmenting path. Currently we are only pushing extra flows along unsaturated edges, which may lead to having the "wrong" edges being saturated. The solution to this is to also allow pushing back against the flow on certain edges. By reducing the flow along an edge, we create an excess inflow to its tail node which we can redirect along some other path to t. We define the so-called *residual graph* that encodes all possible ways of pushing flows forward along unsaturated edges or pushing backward against an existing flow.

Definition 10 (Residual graph)**.** *Given a flow network* (G, s, t, c) *and a flow* f *the residual graph* G_f *is a graph* $G_f = (V_f, E_f)$ *together with a map* $r : E_f \to \mathbb{R}$ *called the residual capacity function.* G_f *has the following properties:*

- $V_f = V$.
- $E_f = E_{forward} \cup E_{reverse}$.
- *If* $e = (u, v) \in E$ *and* $f(e) < c(e)$ *then there exists an edge* $e_{forward} = (u, v) \in E_{forward}$ *and* $r(e_{forward}) = c(e) - f(e)$.
- *If* $e = (u, v) \in E$ *and* $f(e) > 0$ *then there exists an edge* $e_{reverse} = (v, u) \in E_{reverse}$ *and* $r(e_{reverse}) = f(e)$.

Figure 2.6a shows a flow network with a flow and Figure 2.6b shows its corresponding residual graph. Even though there is no unsaturated s-t path in the flow network, there is still an s-t path (drawn red) in the residual graph along which we can augment the flow. This involves actually reducing the flow along the middle edge of the path.



(a) A flow network without an unsaturated s-t path.

(b) The residual graph G_f of the flow network in Figure 2.6a. We can see that it contains one s-t path (marked red).

Figure 2.6: Illustration of how the residual graph helps us find augmenting paths.

We can now put this to use in the so called *Ford-Fulkerson* algorithm:

Algorithm 2: Ford-Fulkerson's algorithm

```
input  : Flow network (G, s, t, c)
output: Maximum flow f

f(e) ← 0 for all edges e in G
construct residual graph G_f

while there exists an s-t path P in G_f do
    r_min ← ∞
    foreach edge e_f in P do
        r_min ← min(r_min, r(e_f))
    foreach edge e_f in P do
        if e_f is forward edge then
            f(tail(e_f), head(e_f)) ← f(tail(e_f), head(e_f)) + r_min
        else
            f(head(e_f), tail(e_f)) ← f(head(e_f), tail(e_f)) − r_min
    Update residual graph G_f
return f
```

Theorem 2 (Correctness of the augmenting path algorithm)**.** *If the capacities of the flow network (G, s, t, c) are integer values, then the augmenting path algorithm will find a maximum flow in finite time.*

Proof. If $c(e)$ is integral, the flow's value will always increase by at least 1 after each augmentation step. Since $\mathrm{val}(f)$ is upper bounded by any cut, we can be certain that it the algorithm will reach a maximum value in finite time and we only need to show that the flow is maximum at that point.

The algorithm terminates its loop once no s-t path exists. At this point, let Q be the set of all nodes reachable from s in the residual graph G_f. In the flow network, the set Q has the following properties:

1. $t \notin Q$.
2. The edges $\delta_+(Q)$ are all saturated.
3. The edges $\delta_-(Q)$ carry no flow.

(1) follows directly from the definition of Q.

Assume (2) does not hold. Then there must exist an unsaturated edge leaving Q. This means that there exists a forward edge that leaves Q in G_f and thus that edge's head node is reachable from s in G_f. Therefore, that node is in Q which contradicts that the edge in question belongs to $\delta_+(Q)$.

(3) can be proven in the same way.

This means that

$$\mathrm{val}(f) = \sum_{e \in \delta_+(Q)} f(e) - \sum_{e \in \delta_-(Q)} f(e) = \sum_{e \in \delta_+(Q)} c(e), \qquad (2.14)$$

since the edges in $\delta_+(Q)$ are all saturated and the ones in $\delta_-(Q)$ carry no flow. According to lemma 1 there can not exist any flow with a higher value, thus the flow that is returned by the algorithm is maximum.

□

The observant reader will notice that we ended up proving the Max-flow/min cut theorem as a side effect. The set Q from the proof has the exact outgoing capacity as the flow delivered by the algorithm, and by lemma 1, there can not exist any cut with a smaller capacity.

The need for $c(e)$ being integral is quite important, as there has been constructed flow networks with irrational capacities on which the algorithm neither terminates nor converges to the maximum value. See [10] for an example.

Theorem 3 (Complexity of Ford-Fulkerson)**.** *Ford-Fulkerson's augmenting path algorithm runs in $O(|E| \cdot U)$, where $U = \max c(e)$.*

This is bad news, as we would really like the complexity of our algorithms be polynomial in $|V|$ and $|E|$. Luckily, the algorithm is quite generic and leaves a lot of room for specialization. The key is in how we go about finding the augmenting paths.

The Edmonds-Karp algorithm is one such specialized version of the augmenting path algorithm. Their algorithm is designed to always pick the shortest possible path to augment. This is achieved by doing a breadth-first search from s to t in the residual graph between each augmentation. We need a couple of further definitions in order to understand some of the subsequent algorithms that we will consider:

Definition 11 (Subgraph)**.** *Let $G = (V, E)$ be a graph. $H = (V', E')$ is a subgraph of G if*

- *$V' \subset V$ and $E' \subset E$,*
- *If $(u', v') \in E'$, then $u', v' \in V'$.*

Central to the next algorithms is a specific type of subgraph known as a *tree*. We are specifically interested in trees with certain properties coming from the direction of their edges:

Definition 12 (Forward tree)**.** *A graph $T = (V, E)$ is a forward tree if it has a root node $r \in V$ and for every other node $v \in V \setminus \{r\}$ there exists a unique path from r to v.*

We define *backward tree* in the same way, only that the paths are in the opposite direction.

Algorithm 3: Breadth-first search
input : Graph G, node s, node t **output**: Shortest s-t path P, $\emptyset$ if none exists

We leave out the exact implementation here. The only thing important to us regarding breadth-first search is that it constructs a forward search tree, rooted in s, that has the property that for any node in the tree other than s, the path from s to that node is the shortest. We say that the tree is a *breadth-first search tree* and we will make use of this concept later on. Using breadth-first search in

the augmenting path algorithm, we get the algorithm by Edmonds-Karp:

Algorithm 4: Edmonds-Karp's algorithm

```
input  : Flow network (G, s, t, c)
output: Maximum flow f

f(e) ← 0 for all edges e in G
construct residual graph G_f
P ← Breadth-first search(G_f, s, t)

while P ≠ ∅ do
    r_min ← ∞
    foreach edge e_f in P do
        r_min ← min(r_min, r(e_f))
    foreach edge e_f in P do
        if e_f is forward edge then
            f(tail(e_f), head(e_f)) ← f(tail(e_f), head(e_f)) + r_min
        else
            f(head(e_f), tail(e_f)) ← f(head(e_f), tail(e_f)) − r_min
    Update residual graph G_f
    P ← Breadth-first search(G_f, s, t)
return f
```

Theorem 4 (Complexity of Edmonds-Karp)**.** *Edmonds-Karp's algorithm runs in* $O(|V| \cdot |E|^2)$ *time.*

A proof can be found in [3]. One thing to note is that there is no longer any requirement of the capacities being integral, the fact that the paths are always the shortest possible ensures that the algorithm is always able to deliver a maximum flow in finite time.

2.2.2 Boykov-Kolmorogov's algorithm

The advantage of Edmonds-Karp is that the augmenting paths are always as short as possible, which means that we get a theoretical worst-case performance. It also means that the augmentation step goes as fast as possible, using just the minimum of comparisons needed to find the minimum residual capacity and the minimum of edges whose flow is updated.

The main drawback lies in the finding of the paths themselves. Since we increase the flow as much as possible at each augmentation, there is always at least one edge that either gets saturated or loses its flow completely. This means that the residual graph always loses one forward or backward edge. The structure of the residual graph gets altered and the search tree that is built by the breadth-first search is rendered invalid. After each augmentation step, we need to completely rebuild the search tree in order to guarantee that the paths we find are of minimal length. This is what the Boykov-Kolmorogov (BK) algorithm sets out to rectify. Instead of completely discarding the search tree after each augmentation, we spend some effort to repair any part of the tree that is severed by lost edges. The exact description of the algorithm can be found in [1], but we will also reproduce and illustrate it in the rest of this section.

The algorithm maintains some auxiliary data structures throughout:

- The current flow f and the corresponding residual graph G_f.
- One forward tree S rooted in s.
- One backward tree T rooted in t. These trees are disjoint and live as subgraphs in the residual graph G_f.
- A set N called the *free* nodes. These are nodes that do not belong to any tree.
- A set of nodes A called the *active nodes.* These are nodes in the trees that might have neighbouring nodes that are free or belong to the opposite tree.
- A map $p : V \to V \cup \{\emptyset\}$ called the *parent map.* This is used to maintain information on how the trees are connected.
- A set of nodes O called *orphans.*

The algorithm begins with a **growth step** in which one of the trees is expanded. An active node is picked to see if there are neighbouring free nodes that can be added to its tree. All such nodes are added to the tree and marked as active. If there is a neighbouring node that belongs to the opposite tree, we have found a *s*-*t* path and move on to the augmentation step. If all neighbouring nodes belong to the same tree, we mark the current node as inactive and pick a new active node for the growth step.

The **augmentation step** works just like in the previous algorithms. The path is scanned to find the minimum residual capacity and the flow is increased by that amount along the path. This leads to at least one edge on the path being removed from the residual graph. If the edge is between the two trees S and T nothing needs to be done, but if such an edge is inside one of the trees

(as often is the case), a part of the tree gets severed from the root node. If the edge is in S, its head node loses its connection to its parent and gets labelled as an *orphan*. If the edge is in T, the tail node is orphaned instead. The orphaned nodes maintain their S or T label.

The **adoption step** tries to repair the trees by finding a new suitable parent node for every orphan. It is required that

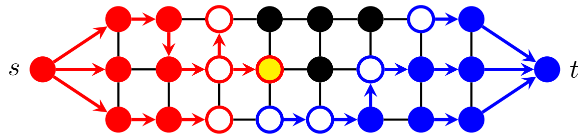
- the parent belongs to the same tree as the orphan,
- there exists a residual edge between the orphan and the potential parent that is oriented the proper way depending on to which tree the two nodes belong and
- there exists an unbroken directed path between the root and the potential parent. The direction of the path is the same as the direction of the tree in question.

If there is no neighbouring node that satisfy these conditions the orphan is marked as a free node and all its children are made orphans. All neighbouring tree nodes are marked active to allow that the newly freed node can be found again by one of the trees once the growth step is resumed.

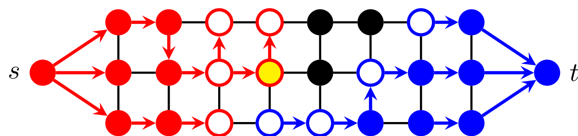
The algorithm terminates once there are no more active nodes. The sets S and T are at that point the minimum cut of the flow network. One complete iteration of the main loop of the BK algorithm is illustrated in Figure 2.7.

The main drawback of BK is the fact that we can not make any assumption of the structure of the trees and thus no assumption on the length of the augmenting paths. Because of this, we can only analyse BK as a generic augmenting path algorithm. There exists no known bound of the worst case performance of the BK algorithm. Empirical studies has shown it to work better than Edmonds-Karp in vision instances (which we will discuss later). Since there is no restriction on the structure of the search trees, it is possible that they grow quite lopsided, resulting in very long augmenting paths.

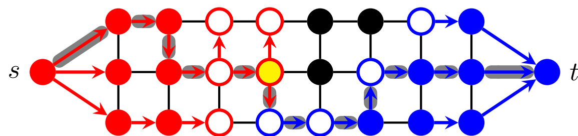
Legend: ● ○ S-nodes, ● ○ T-nodes, ○ ○ active nodes, ● ● orphan nodes, ● free nodes.



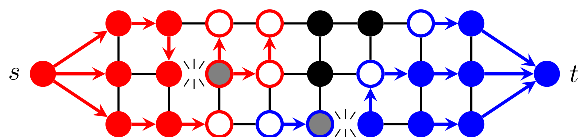
(a) The growth step begins by picking one active node (marked yellow) for expansion.



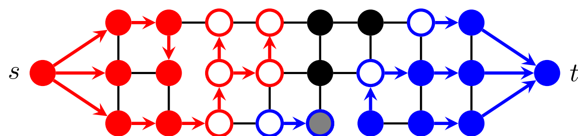
(b) By scanning the active node's neighbours, a free node is found and added to the tree.



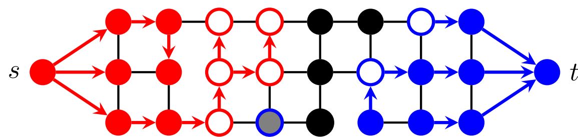
(c) A further scan finds a T-node, which means we have found an augmenting path.



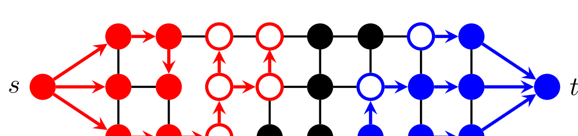
(d) After augmenting along the path, two edges are lost from the residual graph, resulting in two orphans.



(e) A new parent is found for the S-orphan.



(f) No parent can be found for the T-orphan, thus it is made free and its children are made orphans.



(g) No parent can be found for the newly orphaned node, so it is also freed. The algorithm can now resume the growth step.

Figure 2.7: A step by step illustration of one iteration of the BK algorithm's main loop.

2.2.3 Incremental Breadth First Search

Incremental Breadth First Search (IBFS) is an extension of the BK algorithm. As BK set out to always maintain the search trees (however arbitrarily), IBFS also maintains the trees *and* make sure that they are breadth first trees. This means that we always have an upper bound of the lengths of the augmenting paths that we find, without having to perform a new breadth first search after each augmentation. This also leads to a theoretical worst case performance.

The setup is similar to BK. IBFS maintains the following:

- The current flow f and the corresponding residual graph G_f.
- Two directed trees S and T and a set of free nodes N as before.
- The same parent map $p : V \to V \cup \{\emptyset\}$ as before.
- A set of nodes O_s and O_t called S- and T-orphans respectively.
- A distance label d_s or d_t for every node in a tree. It marks the number of edges between the root and the node in question.
- Two maximum distances D_s and D_t.

The S-tree is grown by scanning all nodes whose distance d_s is equal to the current max distance D_s. Any free node $u \in N$ found is added to the tree and gets the distance $d_s(u) = D_s + 1$. If a T-node is found, the growth step is interrupted by the augmentation step. After the growth step, if there are no nodes with distance $D_s + 1$ the algorithm terminates, otherwise D_s is incremented and the growth step begins anew. Growing the T-tree is done symmetrically.

Augmenting the path creates S and T orphans, just like in the BK-algorithm.

Assume we are trying to adopt an S-orphan. We start by looking at potential parents u that fulfil the following:

- There exists a residual edge (u, v).
- $d_s(v) = d_s(u) + 1$.

If such a node is found, we set $p(v) = u$ and remove v from O_s (see Figure 2.8). Otherwise we perform a **relabel step** on v by looking for the potential parent u that minimizes $d_s(u)$. If none such exists or $d_s(u) = D_s + 1$ we free v and make all its children S-orphans. Otherwise we set $p(v) = u$ and $d_s(v) = d_s(u) + 1$. All children of v are made S-orphans. If $d_s(v) = D_s + 1$ then we make v inactive, in case it was active previously. An illustration of the relabel operation can be seen in Figure 2.9.

Processing the T-orphans is done in the same way, only we free v if $d_t(u) \geq D_t$, since we do not wish to grow T at the growth stage of S.

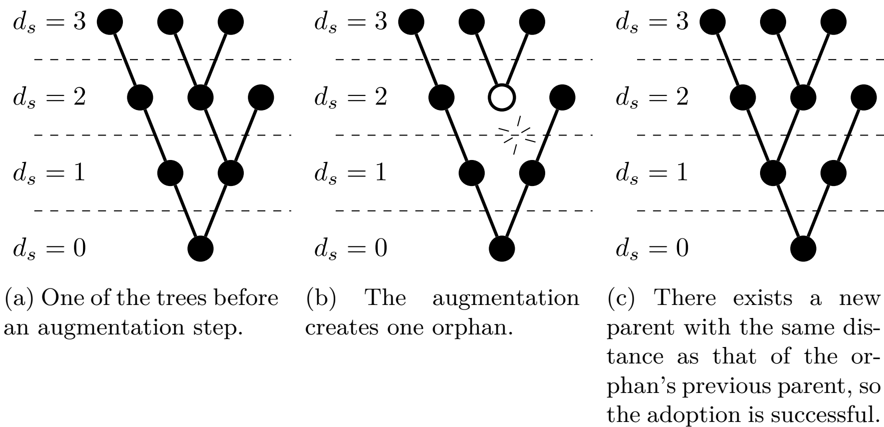
The basic idea of the adoption stage is to find a parent that does not require us to alter the distance label of the orphan. This way there is no need to alter the labels of the orphan's children in order to keep the distance labelling valid. Otherwise, we try to re-attach the orphan as close to the root of the tree as possible. This means that the distance label of all the adopted node's children is made invalid, so we need to make them orphans and process them in the same

way. It is likely that they will be re-adopted by their previous parent as long as it does not mean that their new distance exceeds the maximum distance of the tree.

Empirical testing of IBFS versus BK has shown that in the vast majority of cases, IBFS outperforms BK with a speed increase of about 20 to 50% on a variety of vision instances [6].

We finish our discussion of max-flow algorithms by looking at the complexity of IBFS:

Theorem 5 (Complexity of IBFS)**.** *Incremental Breadth First Search runs in* $O(|V|^2 \cdot |E|)$ *time.*

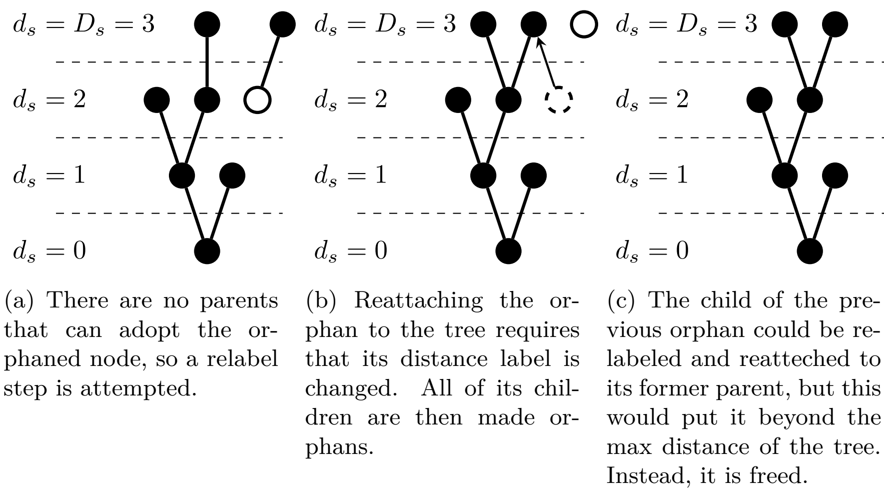


(a) One of the trees before an augmentation step.

(b) The augmentation creates one orphan.

(c) There exists a new parent with the same distance as that of the orphan's previous parent, so the adoption is successful.

Figure 2.8: An illustration of the IBFS adoption stage.



(a) There are no parents that can adopt the orphaned node, so a relabel step is attempted.

(b) Reattaching the orphan to the tree requires that its distance label is changed. All of its children are then made orphans.

(c) The child of the previous orphan could be relabeled and reatteched to its former parent, but this would put it beyond the max distance of the tree. Instead, it is freed.

Figure 2.9: How the relabel step works. It is assumed that we are in the growth stage of T meaning that we do not relabel S-nodes to the distance $D_s + 1$.

2.3 Image segmentation

2.3.1 Introduction

A common problem in computer vision is that of segmenting an image into several parts that all have some desired characteristic. This is often used as an initial step in automatically analysing or processing images as it allows for isolating certain regions of interest and excluding parts of the image that are irrelevant for the application.

There are a multitude of methods for segmenting images. The method of active contours [2], for instance, can be described as a deformable spline that is iteratively changed in order to minimize an energy function. The energy function is small when the spline follows edges in the image and when the spline conforms to a model of the sought segment, such as having a smooth curvature. A disadvantage to active contours is a chance of stopping in a local minimum. For certain applications there is a need for segmentations that are globally optimal, hence there are certain methods that guarantee such segmentations, possibly at the cost of having high computational complexity.

In general, image segmentation can be described as assigning a label to each pixel in such a way that an energy function is minimized. The energy function is a map from the set of all possible labellings to $\mathbb{R}$ and is minimized when the segmentation best conforms to a segmentation model that describes the desired properties of the segments.

An example of this is binary segmentation, where the image is split into two segments. Let P be the set of pixels in the image and let the binary vector $\mathbf{x} = (x_1, x_2, \dots) \in \{0,1\}^P$ indicate the segmentation, that is, $x_p = 0$ if the pixel p belongs to the first segment and $x_p = 1$ if it belongs to the other one. In this setting it is common to use energy functions of the form

$$E(\mathbf{x}) = \sum_{p \in P} D_p(x_p) + \sum_{\substack{p \in P \\ q \in \mathcal{N}_p}} V_{pq}(x_p, x_q). \tag{2.15}$$

The set $\mathcal{N}_p$ denotes all pixels in a neighbourhood around the pixel p, for instance the four pixels above, below, left and right of p. The terms $D_p(x_p)$ are called *data terms* and can be thought of as a penalty for assigning the pixel p to the segment x_p. The terms $V_{pq}(x_p, x_q)$ are called *regularization terms*. These terms are used to encode requirements on the structure and cohesion of the segments. As an example, consider the following regularization function:

$$V_{pq}(x_p, x_q) = \begin{cases} 0, & x_p = x_q \\ \mu, & x_p \neq x_q. \end{cases} \tag{2.16}$$

If $\mu > 0$, this function can be seen as a penalty for discontinuities of the labelling, that is, it penalizes having a large boundary between the segments. The global optimum of the energy function becomes the best trade-off between assigning every pixel to its own data term minimizer and having a short, smooth boundary between the segments. Figure 2.10 shows how the parameter μ affects the segmentation of the famous MIT cameraman image.

Certain energy functions of this form have the remarkable property that an optimal solution can be found by constructing a special flow network where the

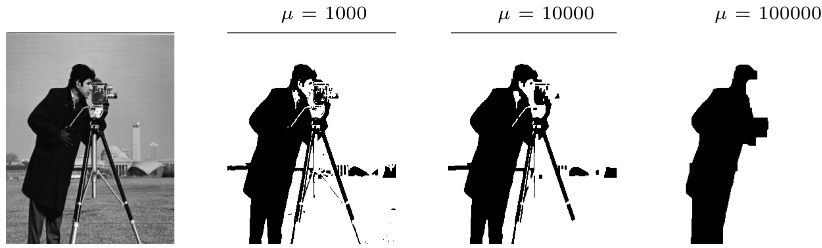


Figure 2.10: Segmentation using different values of μ.

minimum cut corresponds to the minimizer of the energy function. To illustrate this, consider an image with two pixels p_1 and p_2. Assume we are trying to minimize the energy function (2.15), with the regularization term as described in equation (2.16). This allows us to build the flow network in Figure 2.11a.

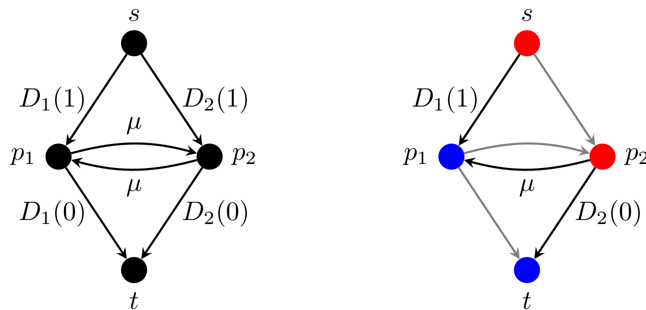
If S, T is a cut in this graph, we let

$$x_{p_n} = \begin{cases} 0, & p_n \in S \\ 1, & p_n \in T \end{cases} \tag{2.17}$$

for $n \in \{1, 2\}$. We see that the following holds:

$$\sum_{e \in \delta_+(S)} c(e) = E(\mathbf{x}). \tag{2.18}$$

This is demonstrated in Figure 2.11b. For bigger images, we get a grid-like graph with a double μ-link between every pair of neighbouring nodes and one s- and t-link with capacities corresponding to the data terms as in Figure 2.11a. As we know by now, the minimizer of the left hand side can be quickly found using algorithms like BK and IBFS.



(a) We can construct a flow network that allows us to find the minimizer of the energy function.

(b) The value of the cut is exactly that of the energy function if we let the nodes in $S \setminus \{s\}$ represent the binary values x_p that are zero.

Figure 2.11: Demonstration of how flow networks can be used to minimize energy functions for image segmentation.

2.3.2 The general setting

Let $\mathbf{x} \in \{0,1\}^N$ be a binary vector of N variables. We will now consider energy functions of the form

$$E(\mathbf{x}) = \sum_i E_i(x_i) + \sum_{j<i} E_{ij}(x_i, x_j). \tag{2.19}$$

Definition 13 (Graph-representability)**.** *An energy function of the form (2.19) is said to be graph-representable if there exists a flow network (G, s, t, c) where the node set of G is of the form $V_G = \{s, t, v_1, \ldots v_N\}$ and the following holds:*

$$\sum_{e \in \delta_+(S)} c(e) = E(\mathbf{x}) + d, \tag{2.20}$$

where d is some constant, S, T is an s-t separating graph cut and

$$v_i \in S \Leftrightarrow x_i = 0. \tag{2.21}$$

It is quite clear that minimizers of energy functions with this property can be found using max-flow/min-cut algorithms. In general, though, there is no guarantee of there being polynomial-time algorithms for finding the global minimum of these kinds of energy functions:

Theorem 6. *Finding the minimizer of (2.19) is an NP-Hard optimization problem.*

So the question remains: What energy functions can be minimized via graph cuts? In the article [9], that very question is posed and answered:

Theorem 7. *An energy function E is graph representable if and only if the following holds:*

$$E_{ij}(0,0) + E_{ij}(1,1) \leq E_{ij}(0,1) + E_{ij}(1,0), \tag{2.22}$$

for all $j < i$.

This property is called *submodularity* and is an important concept also for higher order energy functions, though we will not deal with such functions in this work.

Of course, if we have a submodular energy function, we also need to know how to construct the flow network that allows us to do the minimization. The basic idea is to construct a graph $G = (V, E)$ where $V = \{s, t, v_1, \ldots v_N\}$ and for every $j < i$ where E_{ij} is not identically zero, we have a pair of oppositely oriented edges between v_i and v_j. For every node v_i there are also two edges (s, v_i) and (v_i, t). Finally the capacity for each edge is calculated by a *reparametrization step* that ensures that every capacity is non-negative while still ensuring that the flow network represents the energy function. This is described in detail in [8].

2.3.3 Multi-region energy

As we saw in Section 2.3.1, we can use graph cuts to split an image into two segments. This works in quite an intuitive fashion, as the structure of the graph closely mimics the structure of the image and the energy function. However, since a graph cut only splits the graph into two parts, it becomes harder to see how one can use graph cuts to divide an image into three or more segments optimally.

We reproduce the so called Multi-region framework from [4]: We have a set of pixels $\mathcal{P}$ and a set of labels $\mathcal{L}$. A pixel p can be assigned any combination of labels. If p is assigned label i we say that p belongs to *region* i. We can represent all possible labellings with the binary vector $\mathbf{x} \in \{0,1\}^{\mathcal{P}\times\mathcal{L}}$. We index this vector as $\mathbf{x} = (x_p^i)$ where $x_p^i = 1$ is interpreted as the pixel p being assigned label i. We also introduce the notation for subvectors of $\mathbf{x}$: The vector $\mathbf{x}_p$ is the vector of length $|\mathcal{L}|$ that describes the combination of labels that are given the pixel p and $\mathbf{x}^i$ is the vector that describes what pixels belong to region i. It is important to note that a pixel can be labelled as belonging to multiple regions at the same time, though as we will see, we will be able to forbid or enforce certain combinations of labels.

Using this, we will design energy functions that can describe certain geometric interactions between regions, for instance forcing one region to be contained within another or forcing two regions to be disjoint. This way, we will be able to use models more expressive and robust than what simple binary segmentation would allow.

Multi-region energy functions take on the following form:

$$E(\mathbf{x}) = \sum_{p\in\mathcal{P}} D_p(\mathbf{x}_p) + \sum_{i\in\mathcal{L}} V^i(\mathbf{x}^i) + \sum_{\substack{i,j\in\mathcal{L}\\ i\neq j}} W^{ij}(\mathbf{x}^i,\mathbf{x}^j). \tag{2.23}$$

The functions V^i are the same regularization terms that we saw in section 2.3.1, only now we have different regularization functions for each layer:

$$V^i(\mathbf{x}^i) = \sum_{\substack{p\in\mathcal{P}\\ q\in\mathcal{N}_p^i}} V_{pq}(x_p^i, x_q^j), \tag{2.24}$$

where $\mathcal{N}_p^i$ describes the connectivity of the pixel p in the region i.

A new concept is that of the *interaction terms* $W^{ij}(\mathbf{x}^i,\mathbf{x}^j)$, which are defined as

$$W^{ij}(\mathbf{x}^i,\mathbf{x}^j) = \sum_{(p,q)\in\mathcal{N}^{ij}} W_{pq}^{ij}(x_p^i, x_q^j). \tag{2.25}$$

These allow us to create interactions between regions. As an example, consider the interaction $W_{pp}^{ij}(0,1) = \infty$, that is, an infinite penalty for assigning a pixel to label j but not label i. This way, the interaction forces the region j to be contained in region i. If we let $W_{pq}^{ij}(0,1) = \infty$ for all pixels q within some radius of p, we force j to be inside i with some minimum distance between their boundaries.

In [4], three main interactions are introduced:

i contains j		
x_p^i	x_q^j	W_{pq}^{ij}
0	0	0
1	1	0
0	1	∞
1	0	0

i excludes j		
x_p^i	x_q^j	W_{pq}^{ij}
0	0	0
1	1	∞
0	1	0
1	0	0

i attracts j		
x_p^i	x_q^j	W_{pp}^{ij}
0	0	0
1	1	0
0	1	0
1	0	α

The attraction interaction reminds a bit of the inclusion interaction in the sense that it penalizes having one region without the other, only not as strictly. This creates a spring like attraction of j towards i.

We note one problem however; the exclusion interaction, while very useful, is *supermodular* meaning that we can not minimize the energy function with a graph cut. We can solve this by doing a *label flip* of i. The meaning of the variable x_p^i is changed as $x_p^i = 1$ signifying the pixel p *not* belonging to region i and vice versa. We can interpret this as modelling the complement of region i and replacing the exclusion interaction with an inclusion interaction between j and the complement of i, making the energy function submodular once again.

There may still be cases in which there is no possible way of achieving submodularity through label flipping. Every exclusion interaction needs to be between two regions, out of which one is flipped. Any submodular interaction has to be between two regions that are both either flipped or unflipped. This may lead to situations where a label needs to be both flipped and unflipped – we have a so-called "frustrated cycle." In this case we are unable to use graph cuts for minimization.

Chapter 3

Implementation Aspects

Now that we have established all that we need to understand about flow networks and image segmentation, the next question is how we go about actually putting these techniques to use. We need to implement a way of representing these kinds of flow networks algorithmically, flow networks that may contain millions of nodes and even more edges. It is clear that this causes very high demands on the amount of memory needed for the algorithm. As we will see, we can use some clever techniques to reduce the memory requirements without sacrificing speed.

3.1 A generic implementation

A common way of representing graphs is by adjacency lists. For every node we store a list of neighbouring nodes and incident edges. These lists are often implemented as arrays of pointers:

```
struct node {
    node** neighbours;
    edge** incident;

    //node data
    ...
}

struct edge {
    node* head;
    node* tail;

    //edge data
    ...
}
```

To save memory, we could leave out the edge `struct`s and just store all the edge data in the edges' tail nodes. This would leave us with every edge being represented by a single pointer between two nodes.

Imagine that we are using this graph implementation to solve a max-flow problem for segmenting an image with one million pixels. Every node has four

outgoing edges, which on a 64 bit computer results in using 32 bytes of memory for each node. In total we get about 30 megabytes' worth of pointers only to represent the connectivity of the graph. That is quite a lot of data, but do we actually need it?

3.2 Using the grid structure

We remind ourselves what the graphs for energy minimization look like: For every variable x_i we have a node v_i in the graph. For every two variables x_i, x_j that interact (E_{ij} is not identically zero), there are a pair of oppositely oriented edges between v_i and v_j. The thing to note is that in image segmentation, x_i and x_j interact only if their corresponding pixels are neighbours in the image. This means that for an $M \times N$ image, we can arrange the nodes into an $M \times N$ grid with double edges between every two neighbours in the grid. See Figure 3.1. In this case it is redundant to store any pointers to adjacent nodes as we are always able to compute the index of the neighbours by adding a known offset to the index of the node in question. If the nodes are indexed in a left $\rightarrow$ right, up $\rightarrow$ down order, we can access the left neighbour of a node just by subtracting 1 from its index and the neighbour below can be accessed by adding M to the index.

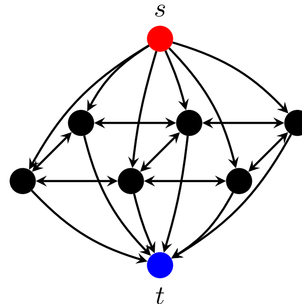
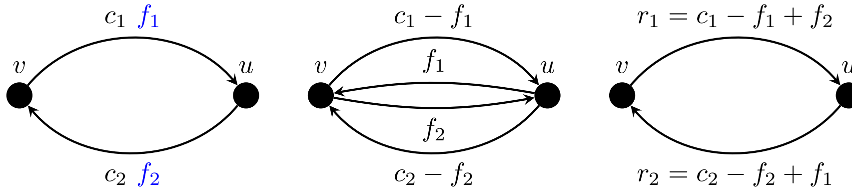


Figure 3.1: Graphs for image segmentation end up having a grid-like structure. The double headed arrows represent two oppositely oriented edges between the same two nodes.

In [7] implementation details for a max-flow solver for binary segmentation are given. Their implementation, called GridCut, draws advantage from the grid structure and also makes some optimizations for increased cache efficiency. These techniques are reproduced in the following sections.

3.2.1 Compact residual representation

In flow networks, the residual graph is constructed by replacing each edge with a pair of oppositely oriented edges that represent either the possibility of pushing additional flow along the edge in the flow network, or diminishing the flow against the direction of the edge. Since the flow networks for segmentation already contain such pairs of opposite edges (Figure 3.2a), the residual graph will contain two pairs of parallel edges for every two neighbouring nodes. This



(a) The flow network has a pair of sibling edges, each with is own capacity and current flow.

(b) The resulting residual graph will in this case contain two pairs of parallel edges.

(c) We can merge these parallel edges back into a pair of residual sibling edges.

Figure 3.2: How we can achieve a compact residual representation by merging edges.

means that we always have two possibilities of augmenting the flow between neighbouring nodes: Either pushing additional flow along one of the edges or pushing back against the other edge. In the residual graph, these two possibilities are represented as two parallel edges with two residual capacities $c_1 - f_1$ and f_2 respectively (see Figure 3.2b). We can merge these two edges into one with the residual capacity $r_1 = c_1 - f_1 + f_2$. When we augment along this edge, the amount of flow we are augmenting with is subtracted from this residual capacity and added to the oppositely oriented residual edge. This compact representation of the residual graph is shown in Figure 3.2c.

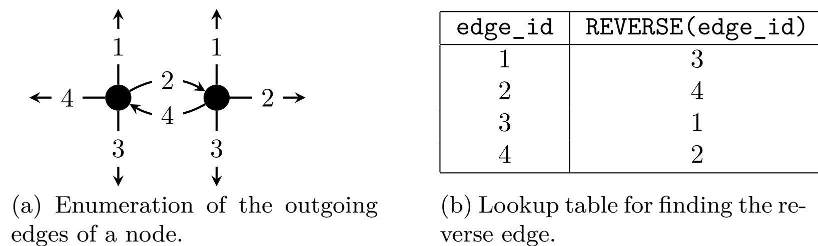
However, by doing this we lose information on the exact flow along each edge as well as the original capacities. The max flow algorithm will still be able to deliver a minimum cut and the value of the maximum flow, but the exact distribution of the flow on each edge is lost. This is a perfectly fine trade-off, as we are only interested in the minimum cut and possibly the value of the maximum flow.

We can combine this representation with the simplifications that the grid structure allows us. We start by enumerating the edges that leave a node. Since all nodes have the same local neighbourhood, this enumeration can be applied to every node consistently (Figure 3.3a). In every node, the residual capacity of every outgoing edge is stored. When we augment along an edge, some of its residual capacity is transferred to its sibling edge. We can quickly access this edge by using a lookup table that maps each edge index to the index of its corresponding reverse direction (see Table 3.3b).

This yields a concise representation of the residual graph between nodes in the grid, but we also need to address how to represent links to the terminal nodes. The following result is of use for us:

Theorem 8. *In augmenting path algorithms, the flow maintained by the algorithm is never diminished on the terminal links.*

Proof. Assume the flow on an edge e leaving s is diminished at some stage in the algorithm. This implies that the augmenting path at some point returned to s via the reverse edge of e before continuing toward t. This is impossible since



edge_id	REVERSE(edge_id)
1	3
2	4
3	1
4	2

(a) Enumeration of the outgoing edges of a node.

(b) Lookup table for finding the reverse edge.

Figure 3.3: Illustration of how we can treat all nodes consistently by using an enumeration of all outgoing edges.

graph search algorithms always deliver direct paths if such exist. The statement for t is proven in the same way. □

Since every node v in the grid has a s-link and a t-link, the algorithm can start by looking at every possible path of the form $s \to v \to t$ and augment along it. At least one of these terminal links will get saturated, and by the theorem they will remain saturated. If we perform this augmentation before the algorithm has even started, we only need to store the residual capacity $|c(s,v) - c(v,t)|$ and to which of the two terminal links (s,v) and (v,t) this residual capacity belongs. This means that there is no need to explicitly store the s- and t-nodes, all the information necessary for the computation is now stored in the grid nodes.

3.2.2 Cache friendly indexing of the nodes

Reading from and writing to memory is a time consuming procedure. To speed things up, all modern CPUs use a caching strategy. In addition to the large and slow RAM-memory, there is also a hierarchy of smaller and faster cache memories close to the CPU. When a program needs to read a value from memory, the lowest level cache is first checked to see if it contains the data in question. If not, the same thing is done on the next cache that is larger but a bit slower than the lower level cache. This is repeated until the value is found. When this happens, the value is transferred to the CPU. In addition to this, the value and some of its neighbouring values are transferred to the lower level caches. The reason for this is that a read of a value from memory is often followed by a read of its neighbouring values, for instance when looping over an array or when the values represent different variables of the same stack-frame. By doing this, the chance of being able to find the data in a fast, low-level cache is increased. This is called getting *cache hits*. For a thorough explanation of modern day memory and cache strategies, we refer to [5].

In augmenting path algorithms, it is often the case that accessing a node is often followed by accesses to its neighbouring nodes, for instance when growing an active node in BK or IBFS. If we try to ensure that nodes that are close to each other in the graph are stored close to each other in the memory, we will be able to increase the performance of the algorithm by taking advantage of the cache functionality. One way of doing this is by using a *blocked layout* of the nodes. The grid is divided into blocks of size 8×8 and the nodes are indexed

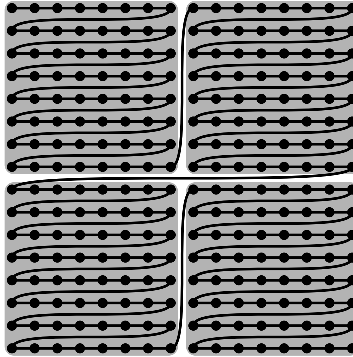


Figure 3.4: The grid is divided into 8×8 blocks and the nodes are then indexed accordingly.

increasingly in a left $\rightarrow$ right, up $\rightarrow$ down fashion within the block. See Figure 3.4.

In order to calculate the index-offsets to the neighbours of one node we now need to determine whether or not the node lies on the boundary of a block. For arbitrary block sizes, this would require performing several modulo and division operations on the node's index in order to determine its coordinates in the grid. However, since the block size is a power of two, we can perform a couple of quick bitwise operations on the node's index to determine if it is on a boundary or not. In the 8×8 case for instance, if the three least significant bits of the index are all zero, the node lies on the right boundary of a block, and if the next three bits are all equal to one, the node is also on the bottom boundary.

3.2.3 Structure splitting

When we run an algorithm like IBFS, the algorithm will enter several different stages in which some information is needed but other information is unnecessary. For instance, consider the augmentation step of IBFS. The first goal is to determine the minimum residual capacity on the augmenting path. We do this by traversing the trees back to their roots, which is done by repeatedly applying the parent map maintained by the algorithm. While this is done we also look at the residual capacity between each node and compare it to the smallest residual capacity currently found. After this, we repeat the traversal and update the residual capacities as we go.

As we can see, this stage of the algorithm only requires access to the residual capacities and the parent map whereas the entire algorithm maintains further data structures like the distance labelling and the $\{\texttt{S}, \texttt{T}, \texttt{FREE}\}$-labelling.

If we were to store this data as an array of node-`struct`s, all these different data fields are stored in an interlaced fashion in the memory. For the augmentation step, this means decreased cache performance as the unnecessary data will also be transferred to the cache.

We can solve this by storing a separate array, indexed by the node indices, for each data field. This way the problem of the data being interlaced is resolved.

3.2.4 Handling the grid boundary

As we have seen, by assuming that the neighbourhood structure is the same for every node, we can gain a lot in terms of memory requirements and cache performance. We have comfortably neglected the fact that this assumption does not hold for nodes on the grid boundary. By treating them as interior nodes with valid outgoing edges in each direction we run the risk of trying to access non-existent nodes outside of the grid, resulting in segfaults or undefined behaviour.

We can prevent this by padding the grid with a layer of dummy nodes, all with zero outgoing and incoming residual capacity. During the growth step, these nodes may be accessed but they will never be added to the trees because of their residual capacities. This allows us to treat the boundary nodes as regular nodes without worrying about things going wrong.

Another issue is that the blocked layout requires the side lengths of the grid to be divisible by 8. We can pad the right and left boundary of the grid with a further layer of dummy nodes until this is satisfied. While this requires us to store some redundant data in the memory, it is negligible even for small grids.

Chapter 4

The Multi-region Implementation

The article [7] introduces several optimizations for solving max-flow problems on grid graphs, which we saw in the previous section. The solver they have written, called GridCut, is available online. The goal of this project has been to write a similar solver, only for multi-region segmentation. As we will see, this becomes different from solving binary segmentations, but many of the optimizations previously discussed can be adapted also for this problem setting. This section will discuss how this multi-region solver developed in this project works and how we can adapt the optimizations to work in this expanded setting. A further difference between the multi-region solver is that it uses IBFS where as GridCut uses the BK algorithm.

4.1 The multi-region grid

In 2D multi-region segmentations with $|\mathcal{L}|$ labels, we get a 3D grid of nodes that is $|\mathcal{L}|$ layers thick. Furthermore, since the interaction terms are between any two layers, there is most likely no way of ordering the layers so that each node has just one upward edge and one downward edge to the neighbouring layers. Even worse, the region interactions are not restricted to be just interior to one pixel but within a neighbourhood of every pixel. This results in "slanted" edges between layers.

We divide the interaction terms into *interior* and *exterior* interactions:

$$W^{ij}(\mathbf{x}^i, \mathbf{x}^j) = W^{ij}_{pp}(x^i_p, x^j_p) + \sum_{(p,q)\in\mathcal{N}^{ij}\setminus(p,p)} W^{ij}_{pq}(x^i_p, x^j_q). \tag{4.1}$$

For this implementation I have chosen to just support interior interactions as it makes the graph still resemble a 3D grid, albeit one that is completely connected along each column of nodes corresponding to each pixel. With exterior interactions things become more difficult as the implementation would need to support the ability to dynamically define the interaction neighbourhoods at run-time. Figure 4.1 shows what the grids supported by the implementation look like.

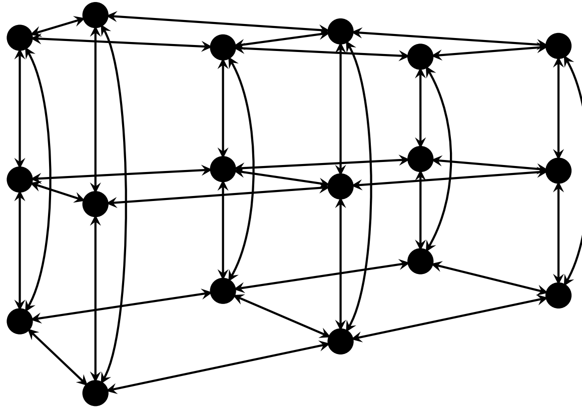


Figure 4.1: What a multi-region flow network looks like. This is for a 2×3 pixel image that is to be segmented using three regions. The double headed arrows represent two oppositely oriented edges between two nodes. Every node has a s- and t-link, but for the sake of clarity these links and the terminal nodes have been left out from this illustration. Note how we get a completely connected graph for each column in the grid.

4.2 Node enumeration in 3D

In [7], where the various memory optimization previously discussed are introduced, the one optimization with the most impact on performance is that of the blocked layout for improved cache-friendliness. It is thus in our interests to adapt this also for multi-region grids. While [7] includes such optimizations for 3D-grids by using $8 \times 8 \times 8$ blocks for dividing the grid, this is done under the assumption that the grid extends reasonably far along each direction. For multi-region grids however, it is not unreasonable to assume that there are just 5 layers or fewer, making the $8 \times 8 \times 8$ block division introduce far too many dummy nodes. Instead, if there are $|\mathcal{L}|$ layers, we divide the grid into blocks of size $8 \times 8 \times |\mathcal{L}|$.

4.3 Handling the layer connectivity

The transition to multi-region grids also introduces some new things to consider when calculating the indices of a node's neighbours. In order to calculate the indices within a layer, the strategy of using bitwise operations to determine the offsets can be carried over directly. There is though still the task of calculating the offsets to the inter-layer neighbours. We also need to find a way of enumerating the outgoing edges in a way that allows us to apply the same `REVERSE` lookup table without needing to consider to which layer the node belongs. This is achieved by using a cyclic arrangement of the layers. The neighbour u that lies n layers below a node v is accessed by moving down n layers. In order to return to v from u we move $|\mathcal{L}|-n$ layers further downwards, looping back around to v. This way, the same `REVERSE`-table can be used with all nodes. While this allows us to take a position agnostic approach when scanning a node, we still

need to calculate to which layer it belongs in order to find the correct offsets. This requires some divisions and modulo operations, but it is outweighed by the benefits of the resulting simplicity of the implementation, especially since the divisions can easily be replaced by bit shifts in this case.

4.4 IBFS variant: the rank-relabel step

For the multi-region implementation, a slight variation of the IBFS-algorithm is used. The standard way of handling orphans is a two pass process: First, the neighbours are scanned to see if an adoption is possible. If not, the neighbours are re-scanned to find a possible relabelling that minimizes the distance increase of the orphan. If a relabelling is done, all the children of the relabelled node are orphaned. If the implementation has no tracking of child nodes then a further neighbour scan is needed to determine the children of the node.

In order for the algorithm to be correct, it is important that the smallest possible relabelling is chosen, otherwise the invariant that the search trees are BFS-trees is violated. There may however be cases where there are multiple choices of a new parent with the same minimum distance increase. In this case it may be better to choose one over the other. For instance, if one of the potential parents has an unbroken path back to its root when the other has not, the rooted parent is the better choice as this guarantees that the relabelled node is not re-orphaned again, should its new parent end up being relabelled before the current orphan processing step is over.

In general, in the case of a tie between two or more potential parents we can use some heuristic to rank these potential parents in order to find one that reduces the chances of the node being re-orphaned.

Definition 14. *If P_o is the set of all potential parents of an orphan o, a parent ranking function $k : P_o \to \mathbb{Z}$ is a function that satisfies the following:*

$$d(p_1) < d(p_2) \Rightarrow k(p_1) < k(p_2), \tag{4.2}$$

where $p_1, p_2 \in P_o$ and d is the IBFS distance labelling.

We can then use such a ranking function in order to find a new parent in a single pass: We scan all neighbours of an orphan o. If a neighbour is a potential parent its rank is calculated and compared with the currently smallest found rank. If the rank is smaller than that, the parent is marked as the currently best and the scan continues. Once all nodes are scanned, the smallest ranking parent is chosen or the orphan is freed if no parent was found.

This way we can use heuristics like the one discussed previously to choose the better of several parents in case of a tie. We can achieve the same heuristic by using the ranking function

$$k(p) = 2 \cdot d(p) + \mathbb{I}(p \text{ is not rooted}), \tag{4.3}$$

where $\mathbb{I}$ is the indicator function that is equal to 1 if its statement is true and 0 if not. Other heuristics could take into consideration whether the potential parent is a descendant of the orphan or how long the line of ancestors of the parent is.

While this method seems promising, a couple of brief experiments with different heuristics has shown that there is not much to gain over using the greedy heuristic $k(p) = d(p)$, though it might warrant a more thorough study of how it affects the adoption dynamics of the algorithm. The biggest benefit of the rank-relabel step is the need of just a single scan of an orphan's neighbours.

As the multi-region implementation has no storage of the children of a node, we can also use this scanning step to build a list of all children of o in case the relabelling requires us to make these nodes orphans. We may also choose to terminate the scan prematurely should an adoptive parent be found (as opposed to a parent requiring a relabel).

Chapter 5

Results and Evaluation

5.1 A multi-region example

While the main focus of this thesis is the implementation of a max-flow solver for multi-region instances, we will just briefly take a look at an example showing the robustness of the multi-region formulation. Figure 5.1 shows a multi-region segmentation of a colour picture. It uses three regions, with two inclusion interactions. We also try using the same model on a version of the picture with added Gaussian noise and see that it is still able to deliver a good quality segmentation. It is worth noting that this kind of segmentation can not be achieved by a graph cut solver for binary segmentation, as it requires support for a multi-region grid.

The model used is based on the squared distance in the RGB-colour space. We have four target colours $\mathbf{c}_{\text{sky}}, \mathbf{c}_{\text{red}}, \mathbf{c}_{\text{black}}$ and $\mathbf{c}_{\text{window}}$, where each are vectors with three integral elements in the range $[0, 255]$ representing the red, green and blue components of the target colour. This allows us to create four squared distances of the form $d_{\text{sky}}(p) = (\mathbf{c}_p - \mathbf{c}_{\text{sky}})^T(\mathbf{c}_p - \mathbf{c}_{\text{sky}})$, where $\mathbf{c}_p$ is the colour of the pixel p (repeated for each target colour). Using these distances, we create the data terms for three regions.

$$\text{Region 1:} \begin{cases} S(p) = \min(d_{\text{sky}}(p), d_{\text{black}}(p), d_{\text{window}}(p)) \\ T(p) = d_{\text{red}}(p) \end{cases}$$

$$\text{Region 2:} \begin{cases} S(p) = \min(d_{\text{black}}(p), d_{\text{window}}(p)) \\ T(p) = \min(d_{\text{sky}}(p), d_{\text{red}}(p)) \end{cases}$$

$$\text{Region 3:} \begin{cases} S(p) = d_{\text{window}}(p) \\ T(p) = \min(d_{\text{sky}}(p), d_{\text{red}}(p), d_{\text{black}}(p)) \end{cases}$$

We then force the two inclusion interactions region 3 $\subset$ region 2 and region 2 $\subset$ region 1. Regions 1 and 2 are regularized with $\mu = 50$. Region 3, which represents the windows of the building, is regularized with $\mu = 40$ since it is a bit more detailed and requires a lower regularization in order to not disappear completely.

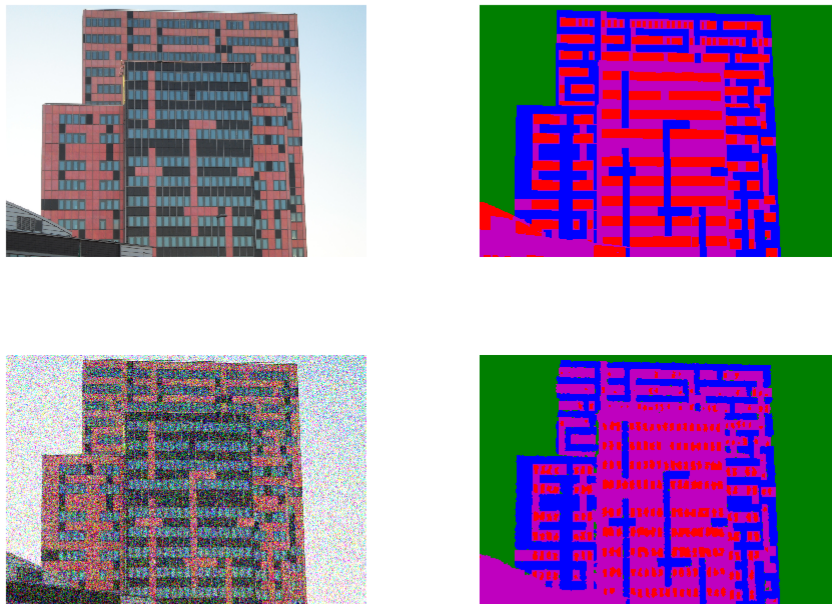


Figure 5.1: A three region segmentation of a colour photograph. We see that the regularization allows us to reject disturbances even in very noisy conditions and still deliver a satisfactory segmentation. The segmentations were calculated in about 4 seconds, including setting up the starting capacities for every edge in the flow network. The grid for this segmentation contains about 3400000 nodes.

5.2 Speed comparison with other solvers

To evaluate the speed of the multi-region implementation, called MRGraph, we use it to do a binary segmentation of an image and compare it with two other available solvers – GridCut as provided by [7] and the generic IBFS implementation provided by [6]. We evaluate them by segmenting four different grayscale images using data terms based on the squared distance between intensities. We create additional problem instances by varying the regularization factor μ. By repeatedly running each problem instance ten times, we get a median value of the time needed for the implementations. All implementations were timed from the function call for computing the max-flow to the point of finishing the computation. The time for constructing the graphs is not included. The median times with respect to μ for the four images can be seen in Figures 5.2 to 5.5.

It is clear that increasing μ makes the computation take longer. The reason for this is that the increased capacity requires far more augmentations to saturate the edges in the grid. Once the regularization is large enough, the only reasonable choices for a minimum cut are the two when either all s- or t-links are chosen. In this case, all augmentations result in the node attached to a terminal node being orphaned; no edge in the grid can be severed. This means that we get a large branch of the tree whose nodes all run the risk of being orphaned by the relabel step if we are using IBFS. In the case when we are using BK, this instead results in the entire branch getting an increased distance from its

terminal node, eventually resulting in very long paths.

It is quite obvious that MRGraph and Gridcut suffer a great deal when the regularization is made very large. It should be noted that for regularizations this large, the resulting segmentation becomes one single segment spanning the entire image. Such a large regularization has no practical value, so we can conclude that MRGraph and GridCut work very well in practice. GridCut consistently outperforms MRGraph which can be explained by MRGraph being a more generalized solver, with support for segmentations as discussed in Section 5.1.

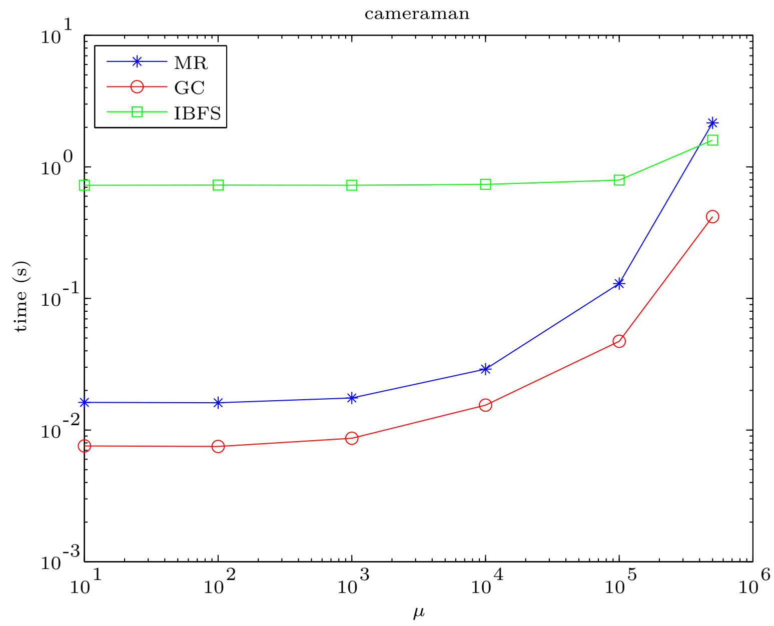


Figure 5.2: Median execution time of the solvers on the cameraman image. 262144 pixels.

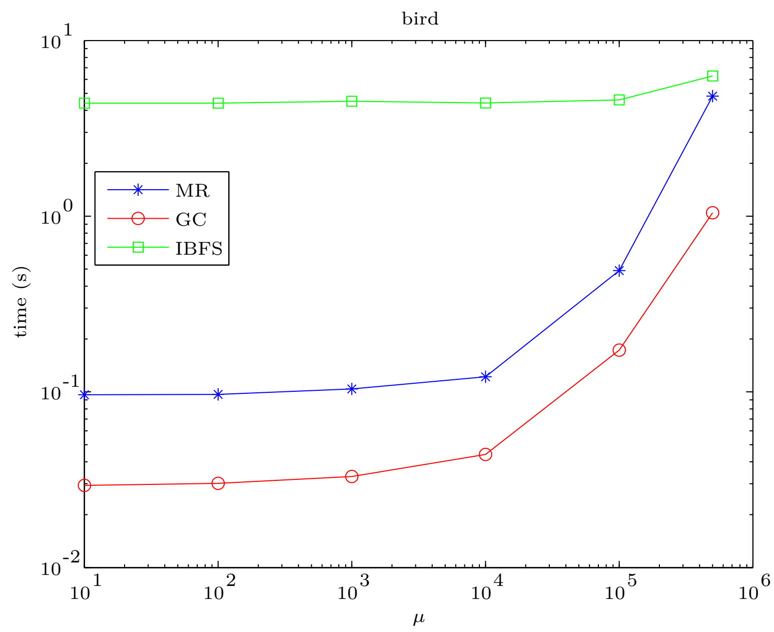


Figure 5.3: Median execution time of the solvers on the bird image. 1329120 pixels.

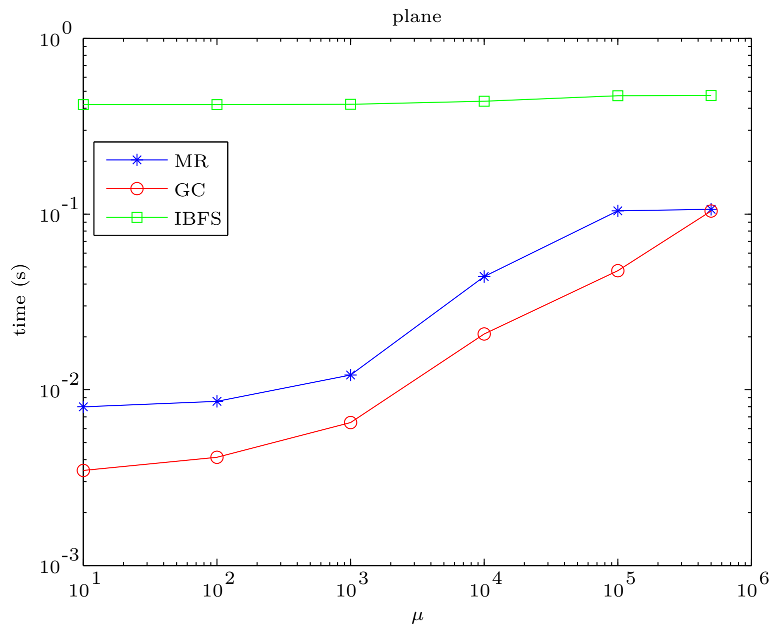


Figure 5.4: Median execution time of the solvers on the plane image. 154401 pixels.

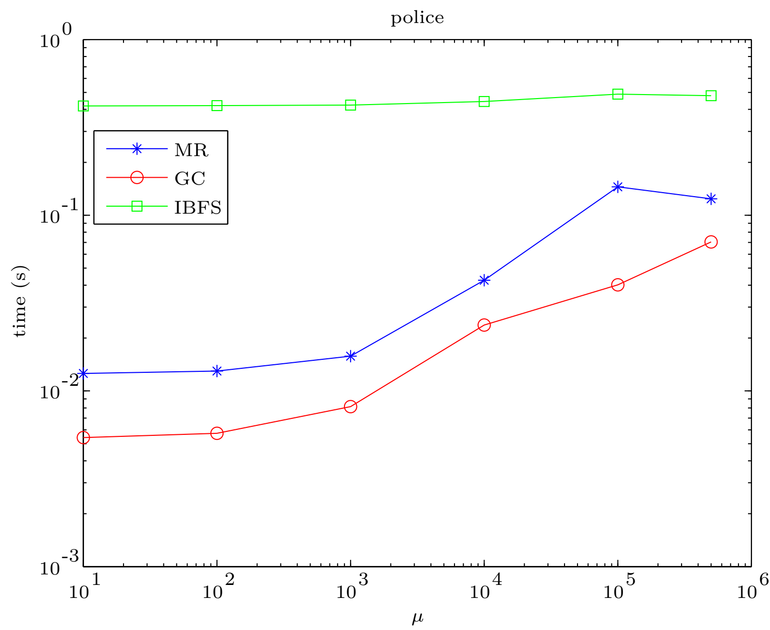
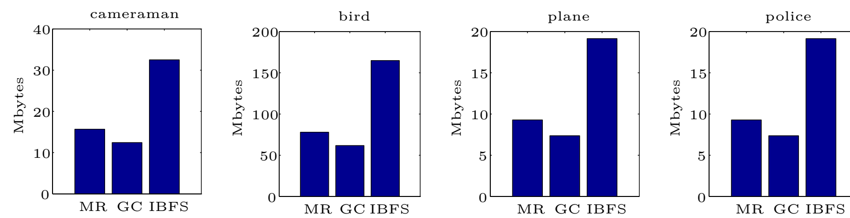


Figure 5.5: Median execution time of the solvers on the cameraman image. 154401 pixels.

5.3 Memory comparison with other solvers

We also estimate the memory required for each solver. GridCut and IBFS allocate memory using `malloc` and `calloc` in their constructors respectively, making it easy to extract the size needed for their allocation. MRGraph allocates several different arrays on the heap when constructed. Once this is done, the sum of the sizes of these arrays are used for the memory estimate. The results of these experiments are shown in Figure 5.6. We note that both MRGraph and GridCut manage to keep a small memory profile. What causes IBFS to use so much more memory is its reliance on pointers for representing the connectivity. Had we used larger connectivities then there would be far more edges in the graph, resulting in many more pointers for IBFS to maintain. The result would be an even smaller memory footprint of MRGraph and GridCut in comparison with IBFS. The experiment was done on a 32-bit computer.



(a) Cameraman image. (b) Bird image. (c) Plane image. (d) Police image.

Figure 5.6: Memory usage of the solvers for the different images. The specialized implementations, MRGraph and GridCut, manage to keep a low memory profile in comparison to the generic IBFS implementation. Had larger connectivities been used, like 8-connected neighbourhoods, this difference would have been even more pronounced.

Chapter 6

Conclusion

This thesis has shown how to solve multi-region segmentations using graph cut techniques. It has introduced a solver for such instances that draws advantage from the regular structure of the resulting graphs. Finally, in the previous section we compared the new solver against two other available solvers on binary segmentation instances. The important thing to note is how the computational burden increases as the regularization is increased. This causes a sort of worst case scenario; only the terminal links are severed by the augmentations, causing many potential relabellings of IBFS, or causing large branches of the search trees to get an increased distance from the terminal nodes when using BK.

I consider this to be worthy of further study. What exactly happens when the regularization is increased? Do we get large propagations of nodes being relabelled along every severed branch, or is it eventually stopped by an adoption? How does the order in which we process the orphans affect this? Can we improve things with a cheap but effective ranking heuristic for the rank-relabel step? And for the BK algorithm, how long do the augmenting paths get? Do the trees grow lopsided, with a vast mixture of short and long branches, all still active? By augmenting the solvers with a stats-gathering friend class, we could gain further insights of the dynamics of these algorithms.

A further area of development is the MRGraph solver itself, that was written as part of this project. MRGraph never manages to beat GridCut. One possible reason is the fact that it is a more general solver with support for neighbourhoods that are not known at compile time. When an MRGraph object is created, the number of layers (regions) are passed as a parameter to its constructor. If MRGraph is to be used for binary segmentation, all its functionality for handling the layer connectivity becomes redundant and may cause suboptimal performance. We could solve this by using templates to ensure the connectivity being known at compile time.

Another thing in need of development is support for different intra-layer connectivities. Currently MRGraph only supports 4-connected neighbourhoods (above, below, left and right), but it should also support 8-connected ones (diagonal connections). There should also be support for 3D multi-region segmentations and their related connectivities (6-connected grids and 26-connected grids). GridCut supports these possibilities, but it does so by providing one class definition for each connectivity. This causes vast code duplication and maintenance difficulties. I would suggest either to use specialized templates for

the connectivity or to have a superclass with the implementation and virtual functions for determining the connectivity. Then we could have different subclasses, each with their own implementation for determining the neighbours of a node. For MRGraph to use a strategy like this would require some refactorization, however.

The source code for MRGraph can be found at `github.com/MartinRykfors/MultiRegion`. For those interested, there are a couple of examples included showing how the class is used. The class interface closely mimics that of GridCut (available at `gridcut.com`).

Bibliography

[1] Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(9):1124–1137, September 2004.

[2] Vicent Caselles, Ron Kimmel, and Guillermo Sapiro. Geodesic active contours. *International Journal of Computer Vision*, 22(1):61–79, February 1997.

[3] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[4] Andrew Delong and Yuri Boykov. Globally optimal segmentation of multi-region objects. In *ICCV*, pages 285–292. IEEE, 2009.

[5] Ulrich Drepper. What every programmer should know about memory. `http://www.akkadia.org/drepper/cpumemory.pdf`, 2007.

[6] Andrew V. Goldberg, Sagi Hed, Haim Kaplan, Robert E. Tarjan, and Renato F. Werneck. Maximum flows by incremental breadth-first search. In *Proceedings of the 19th European conference on Algorithms*, ESA'11, pages 457–468, Berlin, Heidelberg, 2011. Springer-Verlag.

[7] Ondřej Jamriška, Daniel Sýkora, and Alexander Hornung. Cache-efficient graph cuts on structured grids. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 3673–3680, 2012.

[8] Vladimir Kolmogorov and Carsten Rother. Minimizing nonsubmodular functions with graph cuts-a review. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(7):1274–1279, July 2007.

[9] Vladimir Kolmorogov and Ramin Zabih. What energy functions can be minimized via graph cuts? *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2004.

[10] Uri Zwick. The smallest networks on which the ford-fulkerson maximum flow procedure may fail to terminate. *Theoretical Computer Science*, 148, 1995.