Graphical Models

Alexandre Duval
alexandre.duval@student-cs.fr

Thomas Lamson
thomas.lamson@student-cs.fr

Philibert de Broglie
philibert.de-broglie@student-cs.fr

1 INTRODUCTION

This projects aims at comparing different interactive graph cuts methods for binary image segmentation. This means that for each image being segmented, we try to separate the background from the foreground and help our algorithm to do so by interactively annotating some parts of the image with specific colours to indicate the category of parts of the image. We started with the randomised algorithm named Karger and its extension Karger-Stein, before focusing on more complex and powerful algorithms such as Boykov-Kolmogorov and Push Relabel. In addition to implementing these algorithms, we also created an interactive interface to annotate the image on the fly and display the result of the different algorithms. After the first implementations in Python, we re-coded two of the algorithms in C++ to compare the performances and spent some time on optimising them. We compared our results in terms of accuracy and of running time with the Python Maxflow library.

2 GRAPH CONSTRUCTION

In this section, we describe the way we constructed a graphical model from each input image, taking the user's annotations into account and computing relevant edge weights. Figure 1 shows the structure of such graph, where each pixel is a node and where we add two special nodes: the source and the target.

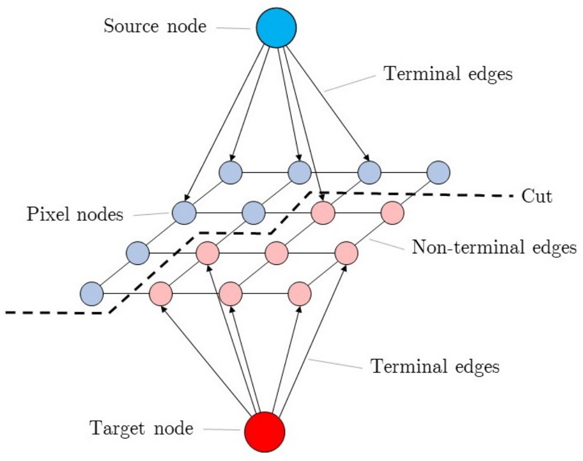


Figure 1: Structure of the graph

First of all, we selected four different images from the web to do binary segmentation. Once downloaded, we created an interactive framework with Pygame allowing us to annotate manually an image with blue for the **foreground** and red for the **background**. In this step, we register the colour and emplacement of the annotations as this is key to compute terminal edges of the graph. As seen in class, the graph contains one vertex per pixel of the image, linked to at most four neighbour pixels (left, right, top, bottom). These undirected edges are called non-terminal edges and are opposed to terminal edges, which additionally link each pixel to the two extra nodes: the source and target. These edges are directed from the source, representing the foreground (blue), to the pixel and from the pixel to the target, representing the background (red). We assign a weight on each edge but compute differently non-terminal and terminal weights. This is because the former gives a measure of similarity between pixels while the latter indicates how likely it is for a pixel to belong to the foreground/background.

Regarding non terminal edges, the computation is very simple. For two neighbours pixels i and j, whose YUV value is indicated by I_i and I_j respectively, weights are given by:

$$w_{ij} = w_{ji} = e^{-\frac{1}{2\sigma^2}||I_i - I_j||^2}$$

This provides weights that are close to a normal distribution and belong to $[0, 1]$, with mean μ and variance σ^2. To get a nicer behaviour, as required for the superpixelisation of the image (see later), we normalise them so that they follow a Gaussian N(0.5, 0.5) distribution using

$$0.5 + \left(\frac{X - \mu}{\sigma\sqrt{2}}\right)$$

Concerning non terminal weights, it is a bit more tricky. We assume that annotated pixels have a probability 1 of being in the annotation class. Their associated weight is therefore set to $w_{iF} = 0$ or ∞, $w_{iB} = \infty$ or 0.

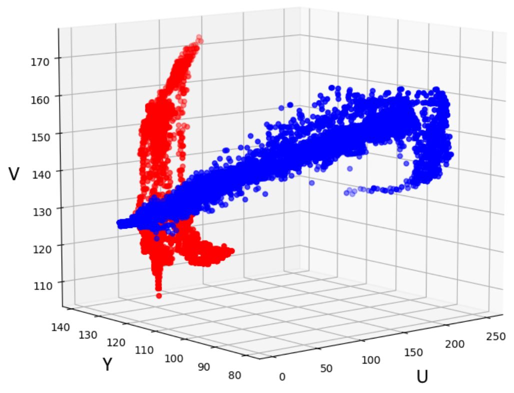


Figure 2: Colour distribution of annotated pixels

For non-annotated pixel i, we first compute an approximation of the colour distribution of each annotated regions through a Gaussian Mixture Model with scipy. Figure 2 shows a 3D representation of the pixel colours in red and blue annotated regions, along Y, U and V axes.

We score pixel i by comparing its score against foreground and background distribution: $P(c_i|F)$ and $P(c_i|B)$. They represent the probability of being of a particular colour given that pixel i is part of the foreground or background. They are then used to compute the probability $P_F(i)$, $P_B(i)$ of a pixel i being in the foreground / background.

$$P_{F(i)} = \frac{P(c_i|F)}{P(c_i|F) + P(c_i|B)} \quad , \quad P_{B(i)} = \frac{P(c_i|B)}{P(c_i|F) + P(c_i|B)}$$

From these probabilities, we deduce the terminal weights as follow:

$$w_{iF} = -\lambda \log P_B(i) \quad , \quad w_{iB} = -\lambda \log P_F(i)$$

3 KARGER

3.1 Theory

Karger [1] is a randomised algorithm that computes the minimum cut of a connected graph. The main idea behind it is to iteratively contract a random edge of the graph, i.e. merge two connected nodes i and j into one and connecting it to all neighbours of i and j while preserving the number of edges or at least their accumulated weights. Only the contracted edge disappears as well as edges that would form self-loops. This process induces the creation of a multi-graph (or the use of a weighted random choice of edge). Indeed, if i and j are both adjacent to a node z, contracting the edge (i, j) produces a new graph with two edges from the new node ij to z (or with an edge whose weight is the sum of the weights of the two original edges). We contract the edges of the graph until two nodes remain, and thus many edges between them (or one weigthed edge).

In a fully-connected graph with n vertices, there are $2^{n-1} - 1$ possible cuts and $\binom{n}{2}$ can be minimal, which is the number of possible pairs of vertices that can remain at the end. A single run of Karger's algorithm gives the result with a success probability of at least $\frac{1}{\binom{n}{2}}$ and a runtime of $O(n^2)$. This is better than picking a cut at random, where the probability that it is minimal equals $\frac{\binom{n}{2}}{2^{n-1} - 1}$. The probability of success can be further increased by doing multiple runs. If we run Karger's algorithm $T = \binom{n}{2} \log(n)$ times, the probability of failure is below $1/n$. So we need to iterate it a large number of times to find the overall minimum cut.

```
Algorithm 1: Karger
  Result: Min Cut value
  Input: graph $G = (V, E)$;
  while $|V| > 2$ do
      Pick an edge $e \in E$ uniformly (or weighted) at random;
      Contract edge $e$ in a single vertex;
      Remove self loops;
  end
```

There are several additional adaptations that are required for our project. The most important one deals with the existence of a source and target node, which we cannot place in the same node partition since they represent different classes. Once this constraint is added, most of the other adaptations relate to the time complexity issue. Running Karger $T = \binom{n}{2} \log(n)$ times increases the complexity to $O(n^2 m \log(m))$ where m is the number of edges. Since our image size is approximately 1000×1000 and each vertex has six edges in most of the cases, the problem becomes nearly intractable if we want a good solution. The optimisation procedure we adopted includes several steps. The first one consists in avoiding the multi-graph approach by enforcing a unique edge between the newly created node and any of its neighbours. The weights are updated accordingly during the contraction step, calculated as the sum of the weights of previously existing edges to that neighbour. This enables to reduce the number of edges in the graph as the algorithm iterates. The second was to move from a *Networkx* based implementation to lists and dictionaries that were carefully optimised to loop as few as possible on the entire list of edges, nodes and weights. As this was still too slow, we developed a more radical framework to reduce the size of the graph to cut with Karger.

3.2 Superpixelisation

We decided to reduce the size of the graph without changing the size of the original image. To do so, we created a superpixelisation algorithm, which replaces a connected group of nodes (pixels) by a single node, and updates the weights in a relevant way.

We started by dividing our image into a grid and placing randomly a certain number of seeds per grid sub-region, so as to obtain the desired number of superpixels in the end. This ensures a smooth but random distribution of seeds. Each seed is then grown so as to form a superpixel of the image. We grow progressively all seeds, 'capturing' a seed's one-hoop neighbours with a probability defined by the normalised weight w_{ij} between the seed i and its neighbour j, if it does not belong to any other superpixel yet. This is why we created a Gaussian on the weights in the first part. After each seed is grown as described, we have superpixels (seed and some captured neighbours), and we repeat the process. At each iteration, we keep track of the new neighbourhood of the superpixels and compute the weights for the next iteration. Whenever no pixels can be captured anymore, we terminate the algorithm.

As a result, we construct a new graph where each superpixel is a node and weights between nodes are determined by the sum of the weights between two superpixels. Finally, the terminal weights of a superpixel is the sum of the terminal edges of each pixel inside it. To decrease even further the computational power required, we directly contract the source/target nodes with the superpixels that contains an annotation. Hence, the final graph contains even less nodes.

As a second improvement, we changed the weights used by the superpixelisation algorithm to use the result of a canny edge detection. This way, edges are much harder to cross when growing the superpixels, and they respect more the contrasted areas of the image. Figure 3 shows a result of a superpixelisation with ten thousands superpixels.

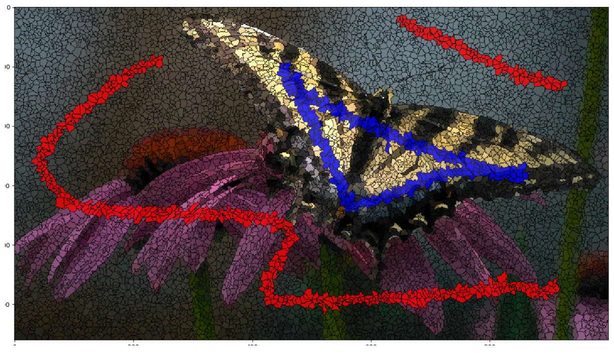


Figure 3: Superpixelisation result

3.3 Karger-Stein extension

Another possible improvement induces the use of an extension of the Karger algorithm, named Karger Stein, which is more performant. According to [2], this algorithm returns a global min cut with probability $O(\frac{1}{\log n})$ in running time $O(n^2 \log^2 n)$. Although a single iteration is longer than the Karger algorithm, it needs fewer iterations to achieve a probability of failure lower than $O(\frac{1}{n})$, requiring a running time of $O(n^2 \log^3 n)$, which is an order of magnitude improvement.

The main idea behind it is that contracting an edge may destroy the minimum cut. So to solve this problem, it applies a series of t random edge contractions to G, like in Karger, but two times and in parallel to obtain graphs G_1 and G_2 from G. It then repeats the process on both G_1 and G_2, until your graph has fewer than 6 nodes. In this case, a classic Karger is applied to end up with a two vertices graph. You therefore have a sort of tree with root G, a mincut value at each leaf and 2^n children at step s where n is the number of graphs in the parent layer. You assign the minimum value across all leaves to be the mincut and retrieve the corresponding partition.

Algorithm 2: Karger Stein

```
Procedure: min-cut(G);
Result: Min Cut value
Input: graph $G = (V, E)$;
if $|V| \leq 6$ then
    Apply Karger mincut ;
else
    $t \leftarrow \left\lceil 1 + \frac{|V|}{2} \right\rceil$
    $G_1 \leftarrow contract(G, t)$
    $G_2 \leftarrow contract(G, t)$
Result: min (min-cut(G1), min-cut(G2))
```

3.4 Results

Once we have run the Karger algorithm on the superpixelled images, we can retrieve the segmented image from the best cut found. We plot the image in a way that allows us to see what parts belong to the background and what parts belong to the foreground. We mainly judge our algorithm intuitively by observing the segmentation output, as well as using the min cut value. If we had a bit more time, we could have used a dataset and a scoring function. However, given the scope of our project, we deemed this to be unnecessary.

The results of Karger are pretty bad, even with superpixels. Given the probabilities of success with graphs such as the ones we use in image segmentation, it is not so surprising. Random approach alone might not be enough for this kind of problem. Figures 4 and 5 show two of our results.

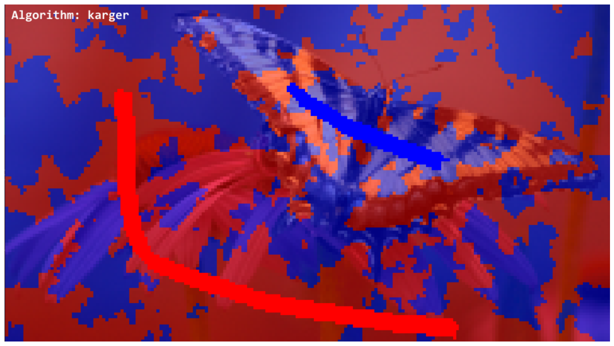


Figure 4: Karger result 1

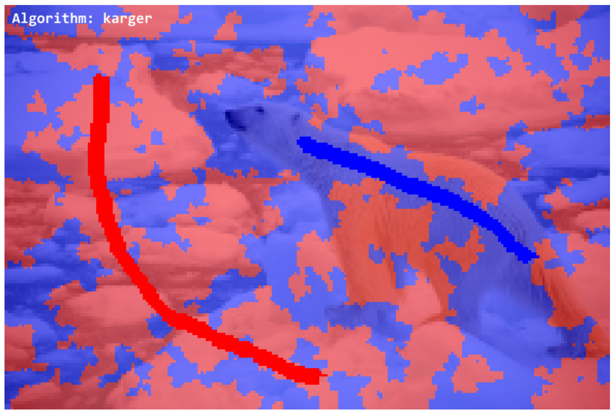


Figure 5: Karger result 2

4 BOYKOV KOLMOGOROV

For this algorithm, [3] [4] define two search trees S and T, non overlapping, such as S and T take respectively the source and target vertices as root. In S, all edges from parent to children are non saturated while in S, all edges from children to parents must be non saturated. Nodes are said to be active when they are on the outer border of the tree, meaning that they can allow it to grow by linking to their neighbours. On the contrary, they are said to be passive when they are internal nodes and consequently cannot help the tree to grow.

This algorithm uses the principle of augmenting path, and especially, it detects when there exists a path form the source node to the target node, using both nodes in S and in T. Note that in the case of image segmentation, we never expand the target tree as only expanding the source tree is enough (very shallow graph). There are three key phases that succeed each other iteratively.

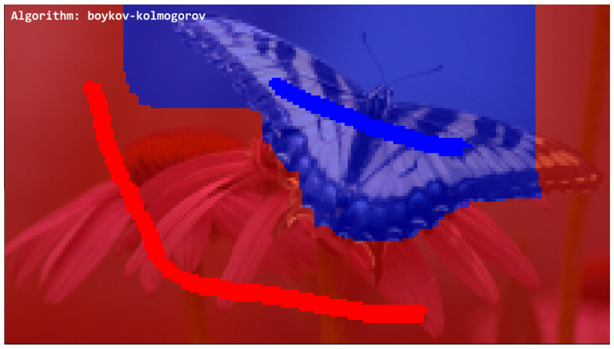
Firstly, the growth stage grows tree S until a path to the target is found.

Secondly, the augmenting stage. The maximum flow that can pass in the found path (which is the minimum of the capacity of the edges along that path) is added to all edges. Some edges (at least one) become saturated and are no longer valid in the tree. This produces orphans nodes (node along the path whose edge linking to parents are now saturated). As a result, the trees T and S have broken down into a collection of trees, or in other terms, a forest.

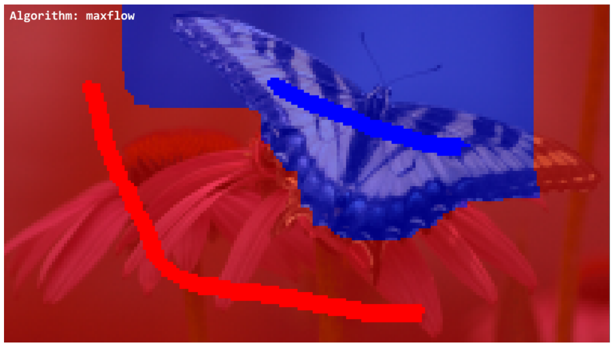
Thirdly, the adoption stage, where trees S and T are restored from orphans. More precisely, at this stage, we try to find a new parent for each orphan. A new parent should belong to the same set, S or T, as the orphan did and should be directly connected to it through a non-saturated edge. If there is no qualified parent, we remove the orphan from S or T and make it a free node. We also declare all its former children as new orphans to be adopted. The stage terminates when no orphans are left and, thus, the search tree structures of S and T are restored.

After the adoption stage is completed, the algorithm returns to the growth stage and the process is repeated. It terminates when the search trees S and T can not grow and the trees are separated only by saturated edges: no valid path can be found. This implies that a maximum flow is achieved and the corresponding minimum cut can be determined by S and T and the saturated edges' capacities at their frontier.

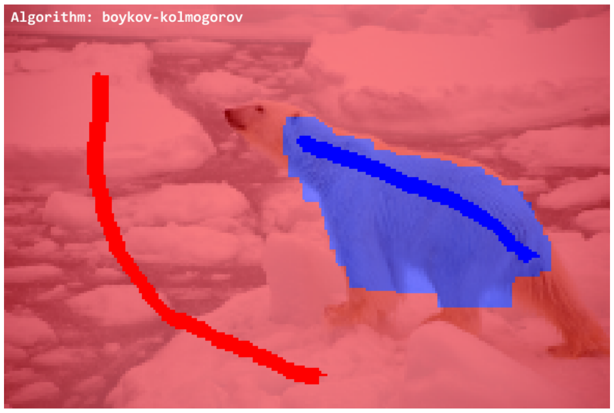
We implemented this algorithm in Python and validated it on small pictures. We don't use superpixels at that point and we compare the results with those of Maxflow library. As it was very slow to run on Python but led to good results on small pictures, we created a C++ extension to our project and re-coded this algorithm so that we could use a compiled version of it in our Python code. Then, we obtained much better speeds and the qualitative results are exactly the same as with Maxflow library, which validates that we apply the algorithm correctly.



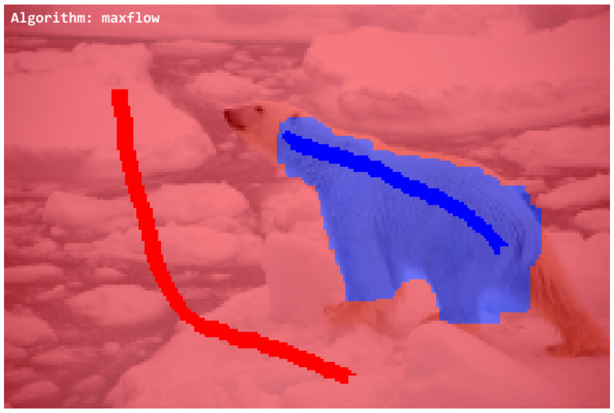
(a) Boykov-Kolmogorov C++ result 1



(b) Maxflow library result 1



(c) Boykov-Kolmogorov C++ result 2



(d) Maxflow library result 2

Figure 6: Compared results

5 PUSH RELABEL

This algorithm [5] was briefly discussed in lecture so we will not delve too much into its theory. To provide a brief intuition of it, the push relabel algorithm is a more efficient version of the Ford Fulkerson algorithm, which also builds on residual graph to compute the maximal flow. However, instead of looking at the whole residual graph to find an augmenting path, we work on each vertex one at a time.
Let's think of nodes as joints and edges as water pipes. The source sends water to adjacent nodes till the target (or sink node). Each node has two main components: the height and the excess flow (difference between input and output flow), which we allow temporarily. Each edge is attributed a flow and a capacity. We initialise the algorithm by setting all height (i.e. label) and flow to 0 except for the source node, whose height equals the number of vertices of the graph and whose incident edges have $flow = capacity$. To make it simple, two operations are applied: push and relabel. If the excess flow of the targeted node is strictly positive and has neighbours with a lower height value, push the minimum value between the excess flow (for the targeted node) and the corresponding edge capacity. If the excess flow is strictly positive but push is not possible, relabel nodes (increase height) until push becomes possible. The repetitive execution of these two operations basically constitute the algorithm. For more details, refer to the code as we implemented this version from scratch.

Similarly to the first part, we use the identitical framework to compute the weights, both terminal and non terminal. We can still use the superpixel-approach but the complexity is lower here [6], (of magnitude $O(V^2E)$) and it is not as essential.

6 CONCLUSION

In this project, we were particularly interested in the way a randomised algorithm like Karger could perform an image segmentation task. We therefore spent quite a lot of time implementing it from scratch, as we tried to adapt it as smoothly as possible to image segmentation. In particular, we were interested in ways to make it run faster while being as relevant or more. This led us to try some approaches like a personalised superpixelisation of the image and consequently tune the graph construction phase, including weights derivation. However the results weren't promising so we decided to implement two other algorithms, from scratch as well, and use a baseline obtained using the PyMaxFlow library in Python. The results for Boykov Kolmogorov were more than encouraging. If we had more time, we would have liked to see how these approaches generalise to multi class labelling, as we briefly mentioned in class.

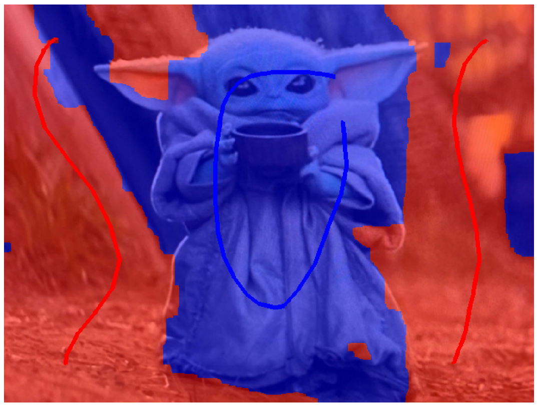


Figure 7: We are still very young to the field!

BIBLIOGRAPHY

[1] David R Karger. Global min-cuts in rnc, and other ramifications of a simple min-cut algorithm. In *SODA*, volume 93, pages 21–30, 1993.

[2] David R Karger and Clifford Stein. A new approach to the minimum cut problem. *Journal of the ACM (JACM)*, 43(4):601–640, 1996.

[3] 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, 2004.

[4] Yuri Y Boykov and M-P Jolly. Interactive graph cuts for optimal boundary & region segmentation of objects in nd images. In *Proceedings eighth IEEE international conference on computer vision. ICCV 2001*, volume 1, pages 105–112. IEEE, 2001.

[5] Boris V Cherkassky and Andrew V Goldberg. On implementing the push—relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.

[6] Bala G Chandran and Dorit S Hochbaum. A computational study of the pseudoflow and push-relabel algorithms for the maximum flow problem. *Operations research*, 57(2):358–376, 2009.