M43062 Optimization and Visualisation

Image Segmentation Report
Daniel Cantos

1 Introduction

Image segmentation referse to the ability to differentiate between a foregraound object from its surroundings. This has many widespread applications in image analysis. The task is as follows. Given an image and a few points marked as foreground and background, separate the foreground from the background. That is, determine whether each pixel belongs to either a forground object, or the background. This problem can be turned into many equivalent problems. Here we use a network optimization model to solve this problem.

2 Model

To solve the image segmentation problem, it is converted into a minimum cut problem for a network flow graph. The idea is to represent each pixel as a node with edges connecting to neighboring pixels. The edge weights should be chosen such that small cuts to the graph naturally occur near the boundary of a foreground object. Let I_u be the pixel array of intensities. We begin by assigning each pixel to a node. Let $\mathcal{P}$ be the set of nodes corresponding to the pixels in the image. To this we add two nodes, S and T, to denote the source and target (sink) nodes respectively. The graph is then defined as

$$\mathcal{G} = (\mathcal{P} \cup \{S, T\}, \mathcal{A})$$

Where $\mathcal{A}$ is the set of arcs. An arc (u, v) is in $\mathcal{A}$ if $u, v \in \mathcal{P}$ correspond to neighbouring pixels, or either u or v is the source or sink node. For the purpose of simplicity, we define this graph to be undirected.

Let $C_{u,v}$ denote the capacity of arc (u, v). There are two distinct types of edges in our graph and we need to define their capacities separately.

Case 1: neighboring pixels . We want arc capacities to be small near the boundaries of a foreground object. A useful heuristic is that foreground and background pixels tend to have differing intensities. An argument based on energy shows that the following function is useful to assign arc capacities based on the intensity of neighboring pixles. Define $B_{u,v}$ as

$$B_{u,v} = \exp\left(\frac{-(I_u - I_v)^2}{2\sigma^2}\right)$$

This function punishes pixel intensities whos difference is greater that σ, so we take σ to be the standard deviation of pixel intensity across the entire image. Foreground pixels tend to be on one side of the distribution, while background the other, so differences greater than a standard deviation indicates a boundary, and this is exactly what this function does.

Case 2: arc to source or sink . Let $\mathcal{F}$ be the set of nodes marked as the foregound, and $\mathcal{B}$ be the set of nodes marked as background. We can connect these to the source (sink) directly. In order to push a decent amount of flow through the graph, we take this capacity to be 1 more than the maximum outgoing capacities among the pixel array, that is,

$$K = 1 + \max_{u \in \mathcal{P}} \sum_{(u,v) \in \mathcal{A}} B_{u,v}.$$

Now we connect the non-marked nodes to either source or sink. THis is not necessary, but can improve performance. We'd like to determine an arc capacity between a given pixel and the source or sink node, depending on how likely that pixel is part of the foreground or background. Here we use a simple heuristic. let $\mu(\mathcal{F})$ and $\mu(\mathcal{B})$ be the mean intensities of the foreground and background marked pixels. To estimate the probability a pixel is in the foreground, we construct two histograms, one for the pixels marked foreground, and one for the background. We can then estimate the probability a given pixel is in the foreground by finding the bucket it falls into, and computing

$$Pr(I_u|\text{"fgr"}) = \frac{\text{pixels in bucket}}{\text{pixels in foreground}}.$$

We do similar for the background. Note that these don't depend on I_u, so they may be computed in advance. If a node has low probability of being in the foreground (background), then we assign a large arc capacity to the background (foreground). To do this, define the following functions (called probability terms).

$$\begin{aligned} R_f(u) &= -\ln(Pr(I_u|\text{"fgr"})), \\ R_b(u) &= -\ln(Pr(I_u|\text{"bkg"})). \end{aligned}$$

With both of these cases handled, we can define all the arc capacities as follows

$$C_{u,v} = \begin{cases} B_{u,v} & u, v \in \mathcal{P},\ \ (u,v) \in \mathcal{A} \\ K & u = S, v \in \mathcal{F} \\ K & u = T, v \in \mathcal{B} \\ \lambda R_b(v) & u = S, v \notin \mathcal{F} \cup \mathcal{B} \\ \lambda R_f(v) & u = T, v \notin \mathcal{F} \cup \mathcal{B} \\ 0 & \text{otherwise} \end{cases}$$

With the condition that $C_{u,v} = C_{v,u}$, this completely defines C. The parameter λ determines how significant the probabilities are in determining edge capacities. Setting $\lambda = 0$ effectively says we do not care about these probability terms.

3 Algorithm

The algorithm used to solve the minimum cut problem is the Golberg-Tarjan preflow-push algorithm. This algorithm solves the maximum flow problem and allows easy comutation of the minimum cut at the end. The idea is simple, for any given node, attempt to push as much flow from it to its neighbours as possible. We define a preflow as any given flow wherby the sum of flow into a node is greater than the sum exiting. Let$x_{u,v}$ denote the flow along arc (u, v), and ex_u denote the excess flow at node u. That is,

$$ex(u) = \sum_{(v,u)\in\mathcal{A}} x_{v,u} - \sum_{(u,v)} x_{u,v}$$

The algorithm then works by selecting a node, and pushing flow out from that node to its neighbours until it has no excess left (this process is called discharging). The algorithm finds a feasible flow once there is no excess left at any node. A push is done by incrementing flow along an arc, and adjusting the residual capacity and excess at both nodes. The maximum amount of flow that can be pushed along an arc can be found as $\max(ex_{u,v}, C_{u,v} - x_{u,v})$.

In order to guide flow towards the sink, for each node, we assign a "distance" label representing the distance from that node to the sink. These are not strictly distances, and are updated as the algorithm progresses. To push flow towards a sink, we want to push flow towards nodes with a smaller diastance label. To do this, we define an arc (u, v) as admissible if the distance label of u is one more than the distance label of v. We then only push flow along an arc if it is admissible. If no admissible arcs exist and there is still excess at a given node, then the distance label needs to be changed. To do this, we find the neighbor with the smallest distance label for which flow can be pushed to, and relabel such that the arc becomes admissible.

The algorithm finishes once there are no nodes with excess left. This also leaves a gap in the distance labels since relabelling causes distance labels to increase. The smallest gap in the distance labels defines an S-T cut, in the graph, where

$$S = \{u \in \mathcal{G} | d(u) > j\}, \quad \text{and} \quad T = \{u \in \mathcal{G} | d(u) < j\}$$

It can be shown this cut provides the minimum cut for the graph.

3.1 Implementation

Pixel intensity is found by converting rgb values into grayscale. If a pixel has colour (r, g, b), then intensity (grayscale) is taken as $I = 0.299*r + 0.587*g + 0.114*b$ The graph was implemented using an adjacency list data structure. Each node contains a list of arcs. Each arc contains a reference to the tail node, head node, capacity, flow and a pointer to the inverse arc (ie: (u, v) contains a pointer to (v, u)).
For discharging operation, the current arc data structure to cycle over neighbours until all excess at a node is discharged. It is implemented as a list containing an index into a nodes list of edges. For the discharge operation, the index is retrieved from the list, then stored again once discharging is complete. The next discharge cycle then continues where it left off
To select nodes for discharging, the relabel-to-front approach was used. All nodes are cycled through, and once a discharge is complete, if a relabel occured, then that node is moved to the front of the list and the procedure is reset. This ensures the node list remains topologically sorted with respect to the admissible arcs. This way, flow tends to be pushed away from the source early, then closer towards the sink as time passes.

4 Results

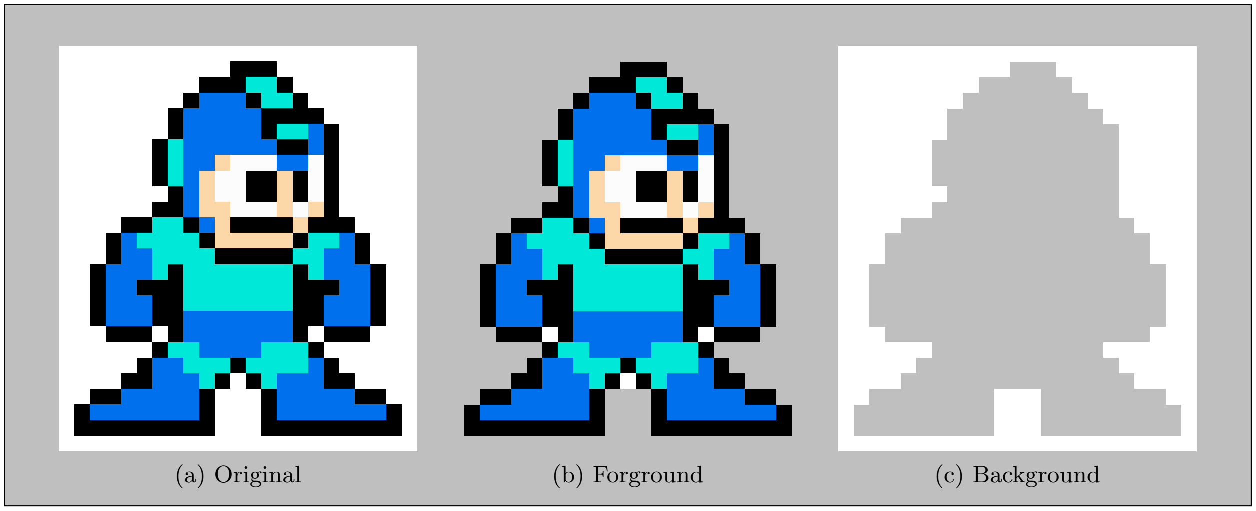


Figure 1: Megaman. Size: 19×26, time to run: 1s

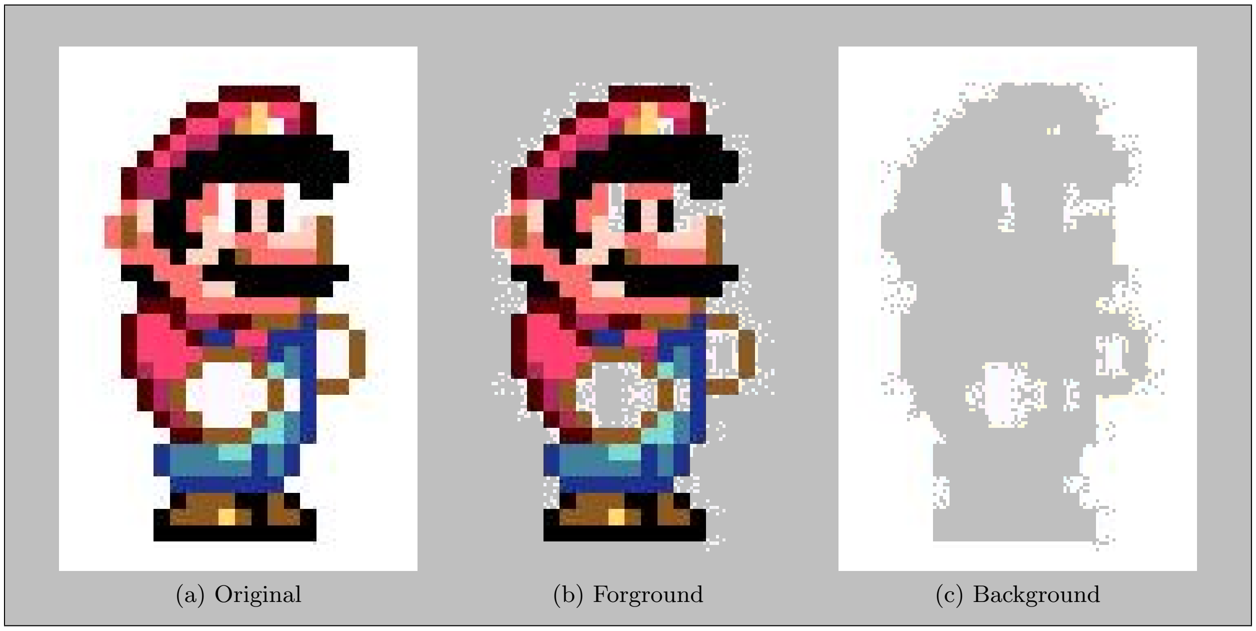
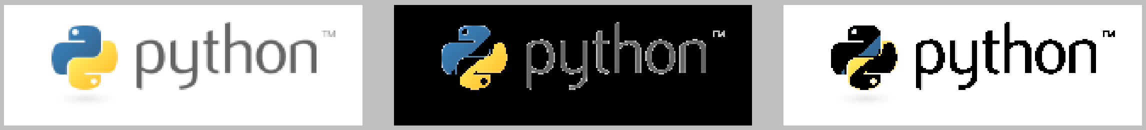


Figure 2: High resolution Mario. Size: 110×161. Time to run: 40s

In figure (2), we see some of the weakness of this model. The image is slightly "dirty" in the sense that the white regions have some noise. The noise spots are picked up as part of the foreground, causing the strange behaviour. We also see that Mario's glove and eyes are also placed in the background since they are white as the background, even though they are part of Mario.

In figure (3), It is unclear why a diagonal slash was taken out of the logo on the left. There is a large instensity difference between the blue and yellow sections of the logo, so a minimum cut will try to cut along this boundary. It may get confused then since both sides have a low probability of being background pixels. This may even be improved by lowering the importance of the foreground



(a) Original (b) Forground (c) Background

Figure 3: High resolution Mario. Size: 211×71. Time to run: 30s

and background probability terms (smalle λ).

Accross other tests, it was noted that the choice of foreground and background points even within a similar looking region can significantly affect the quality of the output

Structure and Usage of the code . The code is organised into three files. The file `annotator.py` contains the annotator code provided in the assignment. The file `graph.py` contains the implementation of the graph datastructure and the preflow push algorithm. The filw `segment.py` contains the routine used to annotate an image then run the segmentation. The routine may be called in a terminal using the following command.

```
python segment.py <filename>
```

The filename argument is optional. It will default to the python logo if unspecified. For this to work, you need a working internet connection.

Images used to test the code can be found in the images folder.

The code contains two hyper parameters. These are located at the top of the main routine in the `segment.py` file. The `l` parameter determines the strength of the probability terms in determining edge capacities between the pixels and the source and sink nodes. The `num_bins` parameter determines the number of bins to use in the histograms. Note that it is not scaled, so more bins result in larger edge capacities. These may be changed to affect the behaviour of the routine.