ZMO semestral work report:
Graph cut library for Python in Rust

Jan Lubojacký
České Vysoké Učení Technické
Fakulta Elektrotechnická
lubojjan@fel.cvut.cz

1 Task requirements

Task requirements reformulated from the e-mail conversation.

- Write a library for Python in a low-level language (Rust).
- The algorithm should be able to solve binary segmentation given a foreground and a background example
- The implementation should make use of either a ford-fulkerson based algorithm or message-passing
- Compare the implementation with other solutions in terms of speed and quality

2 Introduction

2.1 Motivation

Inspiration for this project came to me in the form of a tweet.

Mike Conover
@vagabondjack

We build on a hybrid Rust / Python stack. We'll frequently wrap performance-sensitive compute in pyo3 bindings so it's addressable as a Python module within our modeling and ETL pipelines.

Eye-wateringly fast. Highly recommend this pattern.

anton @abacaj · Nov 6, 2023
Is everyone using rust now?

Figure 1: See the tweet

The semestral project in this subject offers enough flexibility to use it as an opportunity to try this pattern out. It turns out it is quite simple to get this going and develop a module for Python that is written in Rust and, as a result, much faster than a Python native module would be.

2.2 Why Rust?

Rust is a compiled systems programming language focused on performance, reliability, and safety. It has been getting a lot of love in recent years, (most loved language in the annual StackOverflow Survey for several years in a row). As well as backup from industry from Mozilla or Google. It is comparable in performance to C/C++ with a focus on being safe [1].

3 Methods

3.1 Max-flow / min-cut problem for image segmentation

For segmentation via graph cuts, we build a graph for the image so that similar pixels are connected with a high-valued edge. And a low value if they are dissimilar. This should form the bottleneck of the network, and as a result, the cut should lead through here, separating dissimilar pixels from each other.

For defining the weights, we use the following pixel affinity equation.

$$\text{Affinity}(x, y) = \exp\left(\frac{(x-y)^2}{2 \cdot \sigma^2}\right) \in [0, 1]$$

Then, we can apply some standard algorithm for solving the max-flow / min-cut problem for a graph G with a set of lower bounds l, upper bounds u, source s, and sink t.

$$\begin{aligned}
&\max \sum_{\{e \in \delta^+(s)\}} f(e) - \sum_{\{e \in \delta^-(s)\}} f(e) \\
&s.t. \\
&\quad \sum_{\{e \in \delta^+(v)\}} f(e) - \sum_{\{e \in \delta^-(v)\}} f(e) = 0 \quad \forall v \in E(G) \setminus \{s, t\} \\
&\quad l(e) \le f(e) \le u(e) \quad \forall e \in E(G)
\end{aligned}$$

e.g., the Ford-Fulkerson algorithm.

3.2 Algorithms

There are two classes of algorithms for computing the max-flow / min-cut problem. Push-relable class, like the Goldberg-Tarjan algorithm, and augmenting paths Ford-Fulkerson style algorithms. For this project, we implement the edmons-karp algorithm, which belongs to the latter class.

3.2.1 Edmons-Karp algorithm

For computing the maximal flow in the network, we use the Edmons Karp algorithm, which, in each iteration, computes the shortest path from source to sink in terms of edges (i.e., via a BFS) and then augments the path by adding the bottleneck of this path to each edge along the way. The algorithm continues until an augmenting path can be found.

The algorithm's running time is $O(VE^2)$.

3.3 Implementation

3.3.1 Bindings from Rust to Python

The pyo3 library provides a simple interface for exposing rust functions to Python. Here is a fleshed-out version of the `SemestralWork/graph_cuts/src/lib.rs` file, which defines the exposed module. It exposes the function `segment` under the module `graph_cuts,` which accepts several arguments and returns the binary segmentation mask as a NumPy array to the Python interpreter.

```rust
use numpy::{IntoPyArray, PyArray2, PyReadonlyArray2};
use pyo3::prelude::*;

#[pyfunction]
fn segment<'py>(
    py: Python<'py>,
    img: PyReadonlyArray2<f64>,
    mask_fg: PyReadonlyArray2<f64>,
    mask_bg: PyReadonlyArray2<f64>,
    sigma: Option<f64>,
    neighborhood_sz: Option<usize>,
) -> &'py PyArray2<usize> {
    ...
    return segmentation_mask.into_pyarray(py);
}

// A Python module implemented in Rust.
#[pymodule]
fn graph_cuts(_py: Python, m: &PyModule) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(segment, m)?)?;
    Ok(())
}
```

3.3.2 Pixel affinity

For simplicity, we consider only grey-scale images, though extending this to RGB images is the matter of a simple extension of the pixel affinity functions so that it can compute the affinity between vectors of $\begin{pmatrix} R \\ G \\ B \end{pmatrix}$ color values.

To solve the max flow problem, it is easier to have the weights in the network defined as integers (though algorithms for computing max flow with floating point weights also exist), so we discretize the affinity values to the range of $[0, 1, ..., 100]$ by

$$\text{round}(100 \cdot \text{Affinity}(x, y))$$

The choice of the sigma parameter is arbitrary, and the optimal value can differ depending on the image we are trying to segment. If the sigma is low, only very similar pixels will be connected, and we might expect lower-quality segmentation if the texture we are trying to segment is not smooth. On the other hand, a very large sigma will miss smooth edges between similar pixels. We find sigma values around 20-30 reasonable for our testing images.

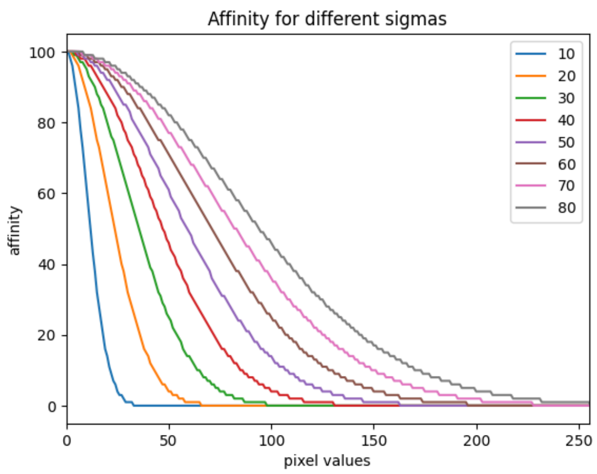


Figure 2: Affinities for different sigma values

3.3.3 Neighbourhood size

When building a graph, we can choose between connecting each pixel to its four (up, left, right, down) neighbors in a 4-neighborhood or diagonal neighbors in an 8-neighborhood. We did not observe a significant difference in either speed or quality between these two methods, but our tests were not extensive. It is configurable via an optional parameter for the main segment function.

3.4 Library description

3.4.1 Project structure

```
├── examples # examples in Python
│   ├── examples.ipynb # jupyter notebook with example usage
│   ├── graph_cuts_pymaxflow.py # segmentation with pymaxflowwe
│   ├── requirements.txt
│   ├── test.py # example invocations, used for benchmarks
│   └── test_imgs # some test images
│       ├── llama.jpeg
│       ├── lungs_ct_huge.jpeg
│       ├── lungs_ct_larger.jpg
│       └── lungs_ct_smaller.jpeg
└── graph_cuts # rust library
    ├── Cargo.lock # rust configs
    ├── Cargo.toml # rust configs
    ├── graph_cuts.pyi # python type & argument hints
    ├── pyproject.toml # rust configs
    └── src # source dir
        ├── edmons_karp.rs # edmons karp algorithm
        ├── graph.rs # data structures to hold the graph
        ├── lib.rs # main entry point for the library
        ├── main.rs # tests in Rust
        └── utils.rs # utility functions (e.g. building the graph)
```

3.4.2 Compiling from source

- Download rust from here, the version used for this project was rustc 1.74.0.
- Optionally run the example in Rust from main.rs to make sure everything works

```
cd SemestralWork/graph_cuts
cargo run
```

- Create a python virtual environment (conda etc.) and install prereqs

```
conda env create -yn most python=3.11
conda activate most
cd SemestralWork
pip install -r requirements.txt
```

- Now you should be able to compile the library locally with

```
cd graph_cuts
maturin' develop --release
```

3.4.3 Example usage

The library features a segment function that accepts the image, foreground mask, and background mask, all in the form of NumPy arrays and optional arguments for sigma and neighborhood size. It can be called from Python like this.

```
import graph_cuts
...
graph_cuts.segment(img, fg_mask, bg_mask, sigma=20.0, neighbourhood_sz=4)
```

A complete example can be found in the `SemestralWork/examples/test.py` file or the `SemestralWork/examples/examples.ipynb`.

4 Results

4.1 Benchmarks, speed comparison

We compare our solution with a Python library for solving the max-flow / min-cut problem PyMaxflow, which is a Python wrapper around the algorithm implemented by V. Kolmogorov in C++ [2]. A function that follows the same API as the one from the rust library is implemented inside `SemestralWork/examples/graph_cuts_pymaxflow.py.`

In [2], an algorithm that works in the augmenting path principle is described. Asymptotically, it is $O(V^2 E|C|)$, (where $|C|$ is the size of the min-cut). However, experimental results show it is up to 26 times faster than Dinic's algorithm, with the difference being larger for larger resolutions. Dinic's algorithm is another max-flow/min-cut algorithm that should be faster than Edmons-Karp for the graphs in this problem because it runs in $O(V^2 E)$.

So from this, it seems like we are comparing a crippled turtle to a race car, and this is indeed what we observe from our results. Since our algorithm is quadratic in terms of edges, it also takes a big hit in performance with increasing sigma, whereas the algorithm developed in [2] is linear in E so it can handle it much better. Also, similarly to [2], we observe that the algorithms are only comparable in speed for small graphs.

image size [px×px]	time to segment pymaxflow [s]	time to segment ours [s]
200 × 200, sigma=10	0.34±0.01	0.45±0.01
200 × 200, sigma=15	0.33±0.01	2.54±0.03
200 × 200, sigma=20	0.36±0.04	6.65±0.14
200 × 200, sigma=30	0.37±0.03	14.18±0.83
400 × 400, sigma=10	1.89±0.13	81.72±3.49

4.2 Example & quality comparision

The `SemestralWork/examples/examples.ipynb` shows some examples of using the library; it contains a function that allows the user to mark foreground and background via a GUI.

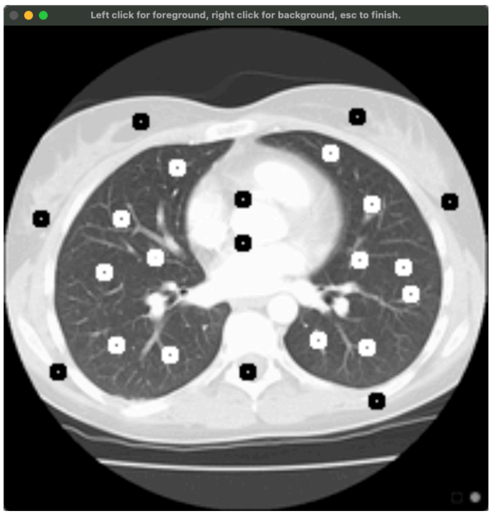
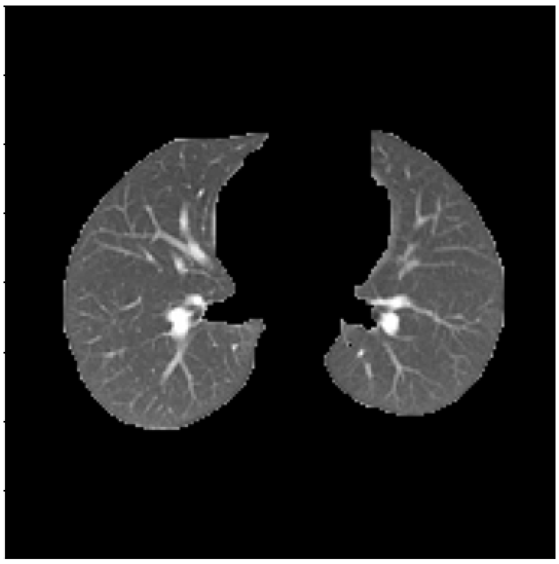
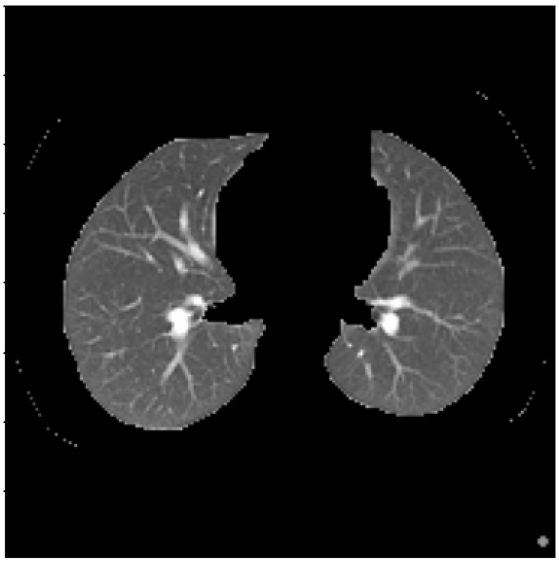

Left click for foreground, right click for background, esc to finish.

Figure 3: Selection of foreground (white) and background (black) seeds via GUI
image credits [3]

The resulting segmentation looks like this

 Segmented image of a lungs CT scan	 Segmented image of a lungs CT scan with pymaxflow

With pymaxflow, the results are a little worse; it struggles with the sharp edge around the image. Our best guess for this behavior is that in [2], they mention that the augmenting path found by the algorithm is not always the shortest one. However, these artifacts could be easily corrected with some morphological operations.

Bibliography

[1] A. Fawcett, “Rust vs C++: An in-depth language comparison”. [Online]. Available: https://www.educative.io/blog/rust-vs-cpp

[2] Y. Boykov and V. Kolmogorov, “An experimental comparison of min-cut/max- flow algorithms for energy minimization in vision”, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 9, p. 1124, 2004, doi: 10.1109/tpami.2004.60.

[3] Radiopaedia, “CT (lung window)”. [Online]. Available: https://radiopaedia.org/images/157288?lang=gb