Minimum Cut problem

Luca Zaninotto – 2057074

30 sept 2022

Contents

1 The problem

The problem we want to solve is to find a cut of a graph such that its weight is minimal. In other words, given a graph

$$G = (V, E)$$

where $V = \{1, 2, 3, \dots, n\}$ a set of n vertices and $E = \{(1, 3), (4, 6), \dots\}$ a set of m edges and a function

$$w : E \to \mathbb{R}$$

a *cost* function, that returns the weight of each edge, we want to find

$$p_1, p_2 \subseteq V \mid p_1 \cap p_2 = \emptyset\,,\, p_1 \cup p_2 = V$$

such that

$$\text{cost}(p_1, p_2) = \sum_{u \in p_1, v \in p_2} w((u, v))$$

is minimized.

2 Karger and Stein

Karger and stein works by joining subsets of nodes, until only two remain. Those two are a cut of the graph. More in detail, given a Graph $G = (V, E)$ where $V = \{1, 2, 3, 4, \dots, n\}$ is the set of vertices, the algorithm starts with a partition of V where each node appears by itself

$$\{\{1\}, \{2\}, \{3\}, \dots \{n\}\}$$

Then, let $\mathcal{P}_a(V)$ be the set of partitions of V, a `contraction` procedure joins two subsets and produces a new partition of V.

$$\text{contraction} : \mathcal{P}_a(V) \longrightarrow \mathcal{P}_a(V)$$

This keeps happening, until the partition of V is made of only two sets p_1, p_2. At this stage the sum of the weights that between the two nodes (the cut of the graph) is returned. This works because of the way in which the `contraction` procedure selects the nodes to join in the partitions. The two nodes are selected based on a random selection based on the weight of the nodes. Nodes with an higher weight have an higher probability of being selected.

Doing such procedure, allows to find a minimum cut with probability $\frac{1}{\log(n)}$, therefore by repeating this procedure $\log^2(n)$ times we can find a minimum cut with error probability less than $\frac{1}{n}$.

Karger and Stein implemented as such has a complexity of

$$O(n^2 \log^3(n))$$

2.1 Results

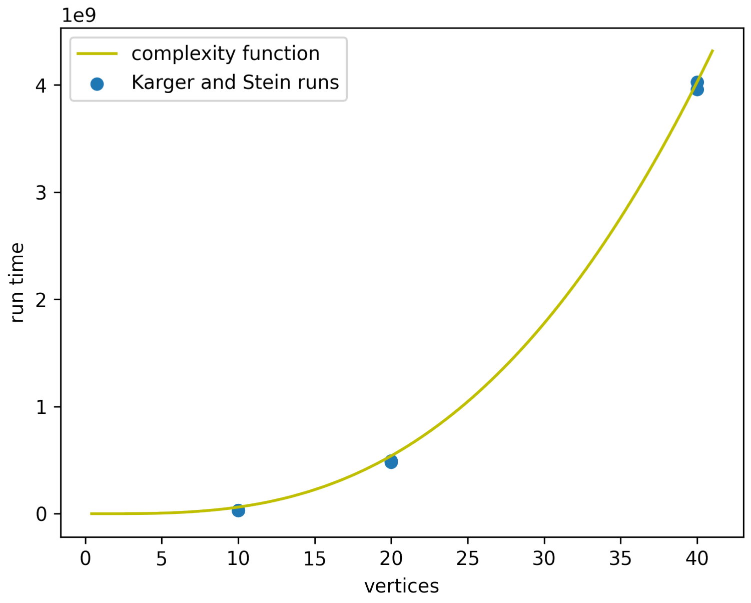

Figure 1: Karger and Stein run times against input size (number of vertices)

The function plotted is $O(mn^2 \log(n))$ that is the overall complexity of the algorithm, the coefficient for the surface (under which all results should fall) is calculated as

$$\max\left(\frac{t}{mn^2 \log(n)}\right)$$

since the time of execution of the algorithm is maximized by this function and the different coefficient depend also on external factors (CPU throttle, memory allocations, etc.) and we're interested the behavior of the algorithm against the input nodes. As we can see in Figure 1 the run times follow the surface of the complexity function.

3 Stoer and Wagner

Stoer and Wagner works in another way. Instead of relying on random cuts relies on the fact that given a couple of nodes, They either are separated by the minimum cut or are on the same side of the cut. So given a graph $G = (V, E)$, $s, t \in V$, an s, t minimum cut is a cut (S, T) of G s.t.

$$s \in S \vee t \in T$$

$$w(S, T) \text{ is minimum among all } s, t \text{ cuts}$$

So, let (S, T) be a global min-cut for G. For every pair $s, t \in V$ either $s \in S$ and $t \in T$ or s and t are on the same side of the cut.

Stoer and wagner leverages this fact and searches for $s, t \in V$ s.t. (S, T) is a global min cut for G. If that's also a global min-cut its weight is returned, otherwise the global min-cut for $G \backslash \{s, t\}$ (The same graph as G, where the nodes s and t are merged together) is also a global min-cut for G.

3.1 On implementation

In `stoer_wagner.py`, in the function `stMinCut`, around line 21 we can find

```
if v in q:
    keys[v] = keys[v] + graph.weight(u, v)
    q.update_elem(v, (keys[v], v))
```

`q` is an object of type `PriorityQueue` (q = PriorityQueue()). The definition for this object can be found in the file `custom_queue.py`. The reason for using a custom version of a priority queue over the default `heapq` implementation of python are essentially 2:

1. Allows to be syntactically closer to the algorithm definition we gave in class. The `PriorityQueue` object implements in fact a max-Heap structure where objects inside are tuples of type `(key, value)`, therefore in the implementation I don't have to rely on tricks such as decrementing keys instead of incrementing them to make a min-Heap (the default `heapq` in python) work like a max-Heap.

2. Allows to update elements inside. the line displayed above shows `q.update_elem(v, (keys[v], v))` which is an internal function to update an element in the queue without removing ti ad adding it again, therefore resulting in a simpler and more concise definition of the `stMinCut` function.

The `custom_queue` implementation was actually found online on GitHub[1], and was originally used by the developer to implement the Dijkstra algorithm. I added the function `__contains__` in `custom_queue.py` in order to be able to write `if v in q:` in `stoer_wagner.py` at line 19.

3.2 Results

The complexity of this algorithm is

$$O(mn\log(n))$$

where $n = |V|$ and $m = |E|$. so the coefficient for the plot is calculated again as

$$\max\left(\frac{t}{mn\log(n)}\right)$$

[1]github.com/denizetkar/priority-queue

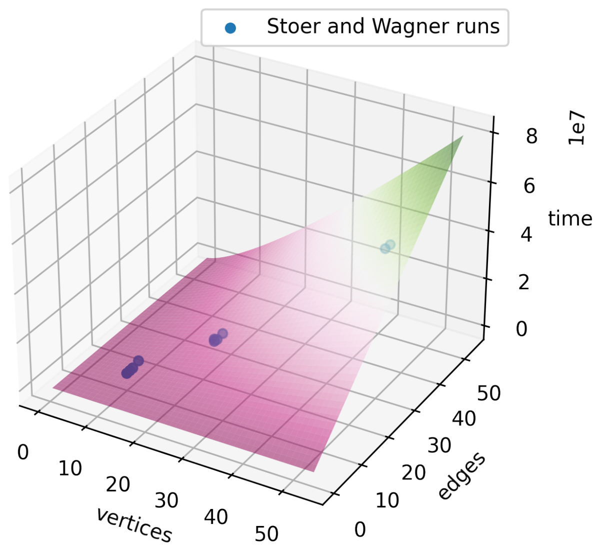

Figure 2: Stoer and Wagner run times against input size (nodes and edges)

By plotting the run times against the input sizes (nodes and vertices) we can see that the plot follows the surface of the complexity function.

4 Hybrid approach

An Hybrid approach consist into merging the two approaches (Karger and Stein and Stoer and Wagner): the algorithm contracts the graph until there are $t = \frac{n}{\sqrt{2}} + 1$ nodes, from there, Stoer and Wagner is run on the contracted graph.

To study the complexity we can take look at the hybrid procedure:

```
def hybrid(graph):
    n = graph.n_vertices
    t = int(np.ceil((n * np.log(n)/ (n-1))))
    amin = np.Inf
    d_time = 0
    for i in range(t):
        cut, d_time = hybrid_iteration(graph)
        if cut < amin:
            amin = cut
            d_time = perf_counter_ns()
    return amin, d_time
```

where the hybrid iteration is described as

```
def hybrid_iteration(graph):
    t = np.ceil(graph.n_vertices / np.sqrt(2) + 1)
    g = contract(graph, t)
    return stoer_wagner(g)
```

one `hybrid_iteration` consists in a contraction of the given graph, then from that contraction we run the Stoer and Wagner algorithm. Since the contraction is just a single loop running $t = \frac{n}{\sqrt{2}} + 1$ times, `contract(graph,t)` is $O(n)$. Stoer and Wagner on the resulting graph is $O(mn\log(n))$ by hypothesis. The whole procedure (`hybrid(graph)`) is just a matter of running the `hybrid_iteration` $\frac{n}{n-1}\log(n)$ times. Therefore the whole procedure has complexity

$$O(mn\log^2(n))$$

4.1 Results

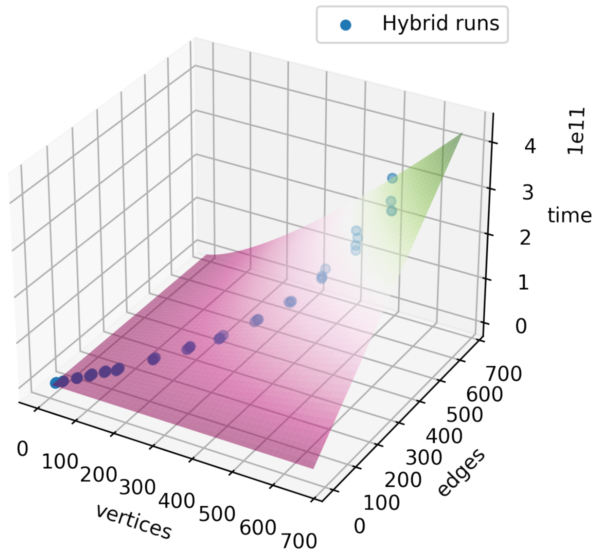

Figure 3: Hybrid run times against input size (nodes and edges)

By plotting the run times against the input size (nodes and edges) we can see that they follow the surface induced by the complexity function of the algorithm.

5 Conclusions

5.1 On efficiency

By purely relying on the complexity analysis of the algorithms we should be able to see how the Karger and stein algorithm should perform worse than Stoer and Wagner and the Hybrid approach, which have instead comparable complexities. By plotting the run time of each algorithm:

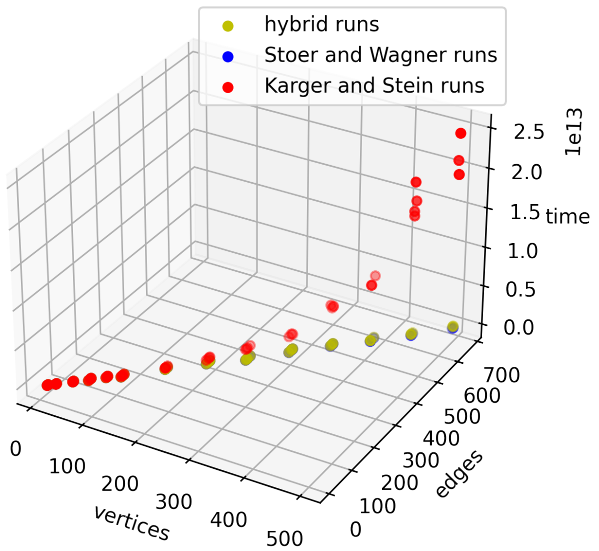

Figure 4: Runtime comparison between the three algorithms

By looking closely to the graph we can see how the results meet our expectations: Karger and Stein perform generally worse than Stoer and Wagner, that performs in a similar way to our approach.

5.2 On discovery time

Discovery time tells us another story. Even though Karger and Stein performs badly compare to the other two algorithms, the discovery time for the graphs in the dataset is not that worse compared to the discovery time of the hybrid algorithm. Stoer and Wagner outperforms the two even from this point of view, showing how is a generally faster algorithm.

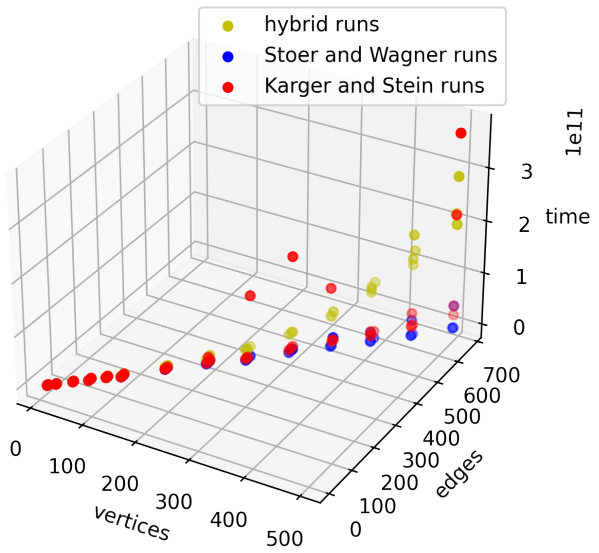

Figure 5: Discovery time comparison between the three algorithms

A Appendix

A.1 On running the script

One of the included file of the project is `requirements.txt`, it contains the dependencies for the project, and is intended to be give to pip in order to download and install them. Is also good practice to create a virtual environment to run the project, in order to avoid dependency conflicts with packages installed in the main system. To do so we can run in a shell

```
python3 -m venv venv
```

A `venv` folder will be created, with scripts to activate the environment depending on the system shell. E.g. on bash on Linux

```
. venv/bin/activate
```

or on windows shell

```
venv\Scripts\activate.bat
```

or Power-Shell

```
venv\Scripts\Activate.ps1
```

From this point on a variety of scripts are available for the project. The main script is in the `main.py` file. can be run with

```
python src/main.py --help
```

to show all the available options. Similarly `plot_d_times.py` and `plot_run_times.py` can be run in order to plot discovery times and run times of precedent runs.

A.2 Structure

Files in the `proj/src` folder are organized as follows:

`graph.py` contains the implementation of a class `Grap`. It represents a graph, and internally consists on the adjacency matrix of the graph built starting by a list of nodes and edges. it implements methods useful for the algorithms (such as `merge_vertices()` and `cut_weight()`);

`stoer_wagner.py` contains the implementation of the Stoer and Wagner algorithm, along with all the functions seeded by the algorithm as subroutines (e.g. `stMinCut`);

`karger_stein.py` contains the implementation of the Karger and Stein algorithm, along with all the functions seeded by the algorithm as subroutines (e.g. `rec_contract`, `contract`);

`hybrid.py` contains the hybrid approach implementation and all the auxiliary functions needed (e.g. `hybrid_iteration`);

`test.py` contains the functions to test each single algorithm and a general `mesure_run_time` function to collect results of tests run in parallel on a given collection of inputs, in order to run multiple test in parallel and waste as little time as possible (each test runs on a dedicated processor, and does not exchange data with anything outside itself, therefore can return reliable time measurements). Each test is run only one time, since the computation of the min cut of each graph is long enough to have reliable data with just one run;

`rapresentation.py` contains functions to plot, print, read ad write csv files of data, relies on the common tuple rapresentation of a single run: `(name, time, discovery time, #vertices, #edges, solution)`;

`custom_queue.py` contains the custom max heap implementation as described in 3.1;

`files.py` contains and exports a simple list with all the files in the `dataset` folder written manually, ideally it could read the directory and return all the files in it but was out of the scope of this project;

`file_parser.py` contains a parser function that reads a file in the dataset and returns a graph based on its contents;

`plot_*.py` are scripts to plot the collected results.

A.3 Collecting results

Each run of a test function (`test_sw`, `test_ks`, `test_hy`) writes the partial results in a file, respectively `stoer_wagner`, `karger_stein` and `hybrid`, in order to use the data also later. The `plot_d_time.py` `plot_run_times.py` scripts read these files and produced the graphs found in this document for the discovery times and run times analysis (5.2 and 5.1).

A.4 Given results

Inside of the `project` folder, three files (`stoer_wagner`, `karger_stein` and `hybrid`) represent a run carried on the 15 oct 2022, and contains the results collected to make this document.