

# Big Data HW8

Tiago Antunes

April 2021

## 1 Problem Definition

The given problem can be formulated as the following: given a graph  $G$  and a number of machines  $p$ , create  $p$  partitions of the graph that try to distribute the load as much as possible (edges) while also lowering the replication factor of the vertices. 4 solutions were implemented for this homework: edge-cut, vertex-cut, greedy vertex-cut, hybrid vertex-cut.

## 2 How to run the code

To run the code, it is required to have Go installed (version 1.16). Then to run it, just use

```
go run cut.go <edge|vertex|hybrid|greedy> <p> <file> [threshold]
```

The input file should be the file input graph in binary format with little-endian encoding for 4 bytes ids. The threshold parameter should be used for the hybrid cut. The default threshold is 3.

To run the greedy cut on a graph called "example.graph" with 3 machines just do *go run cut.go greedy 3 example.graph*

## 3 Code structure

The input is fed through a go channel for all the cut techniques. A thread is dedicated to reading the file and feeding the pairs into the channel, closing it in the end, easing the task of reading the file. Then, the main thread executes the cut while getting the data from this channel.

Each cut has an associated function that is selected with a switch command in the main function. This function then will execute the whole logic of the cut technique. A general type, **Partition**, is used to represent the partition and contain its information. This type includes counters for the algorithms to use freely, and 2 maps to include the mirror and the masters. The logic of the algorithms can then freely manipulate an array of type Partition to save the information.

Each cut has a main loop for each edge. The logic for each cut technique is:

Type	#Machines	Time
Edge	2	12m6.433s
	4	13m22.580s
	8	14m7.557s
	48	17m32.627s
Vertex	2	12m10.116s
	4	12m13.724s
	8	12m32.874s
	48	12m41.922s
Hybrid	2	11m52.545s
	4	12m34.444s
	8	12m24.732s
	48	13m51.232s
Greedy	2	20m31.258s
	4	18m39.782s
	8	22m5.591s
	48	38m41.562s

Table 1: Time taken to run with different number of machines

- Edge
  1. Get the hash of each vertex
  2. Put the edge in both partitions of the given hashes, and put the vertices as master nodes of that partition
  3. If the hashes are the same, lower the count of the edge by one
  4. If they're different, add the corresponding mirror counts on the partitions and put the vertices as mirror nodes
- Vertex
  1. Keep a counter of the index of the next edge
  2. Put the edge in the hash of the counter
  3. Put each vertex in its own hash and in the hash of the edge
- Hybrid
  1. Maintain a set of the vertices that are considered high degree, and a map of vertex id to vector of vertices, to keep track of the edges for low degree vertices
  2. If the destination of the edge is a high degree vertex, then put the edge in the partition of the hash of the source and continue to the next edge
  3. If not, the put the edge in the partition of the destination and put the source node in the map to keep track of it

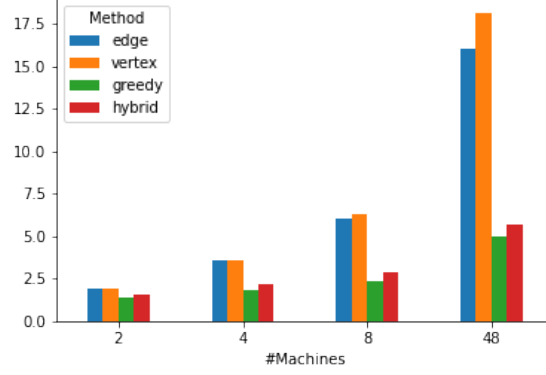


Figure 1: Replication factor for different number of machines across all methods

Type	Complexity
Edge	$O(E)$
Vertex	
Hybrid	
Greedy	$O(E * P)$

Table 2: Complexity of the implementation of edge graph cut technique. P corresponds to the number of partitions used.

4. Check if the target edge exceeds the threshold
  5. If yes, then perform the logic to move the edges to the sources hash.
- Greedy
    1. Find the machines where each vertex is replicated
    2. Select a function according to the rules of the paper (Rules 1-4) which will decide which machines to consider to place the edge
    3. Select the machine that is within the ones given by the selected function and that contains the least amount of edges
    4. Keep track of the mirror and master information of each partition

Hybrid was done in a different way than the described in the paper and in a single pass so that it performs faster (since the graph doesn't fit in memory for the Twitter dataset), therefore the extra logic.

## 4 Results and analysis

A visualisation of the results is available in Table 3

The different types were run to different number of machines (2,4,8 and 48). Hybrid cut was run with a threshold of value 100. To analyse each cut’s performance, the replication factor  $RF$  was calculated given the formula

$$RG = \frac{N_{Master} + N_{Mirror}}{N_{Master}}$$

Given the results in Figure 1 and Table 1, we can take the following conclusions:

- All the cuts have a sub-linear growth given the number of machines to split the graph across
- Although there is a sub-linear growth, Edge-cut and Vertex-cut both have a very high replication rate, reaching values above 15 for 48 machines. On average, each node is replicated across 15 machines, which is about a third of the machines, and this will result in a lot of network overhead when performing graph computation on these partitions.
- Both greedy and hybrid performed much better, obtaining a replication factor close to 5 for 48 machines, about a third less than Edge and Vertex cuts. This result is much more acceptable and will result in better performance of the graph computation
- In terms of partitioning time performance, only greedy performed worse due to its increased complexity, and took almost double the amount of time that was required for the other cuts.
  - Edge and Hybrid both have an increase in time compared to vertex, but that’s due to the way the code is written. Vertex and Greedy had to be optimized for the code to run with 48 partitions, and that optimization resulted in less memory usage, as well as a somewhat increased processing performance
  - Both Edge and Hybrid can have this optimization, but I didn’t find it relevant for this work
- Hybrid cut performs well in terms of time to partition and replication factor, but requires a threshold to be tuned for it to perform well, with the values varying by a lot depending on this threshold, which results in it not being very useful. The time to partition compared to greedy is not relevant enough to compensate for the amount of times one needs to run it just to get a good result.

The results of each algorithm are consistent with what was expected. Hybrid is faster than Greedy but needs to be fine tuned to achieve decent results, and if further work is conducted to find what threshold value should be used, then it is a good algorithm. Overall, Greedy is the best choice, even if it takes longer to split. One thing to be noted is that, because we are doing greedy with a global view of the dataset since it’s a single machine, Greedy will then optimize


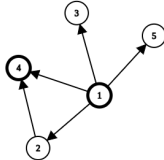
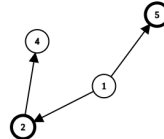
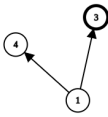
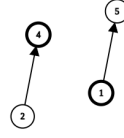
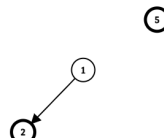

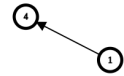
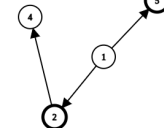
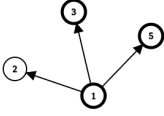
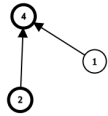
	Partition 0	Partition 1	Partition 2	Rep. Factor
Edge				2.2
Vertex				2
Hybrid				1.6
Greedy				1.4

Table 3: Partitioning visualisation on a shuffled version of small-5 graph

the function to lower the replication or vertices. The small graph was given in a sorted way, and this resulted in an execution of greedy placing all the vertices in the same partition, achieving a replication factor of 1, and so for Table 3 a shuffled version was used. In terms of edges, in the big graphs, there was a good load balance on how they were distributed across all the results, being the vertex replication factor the metric used to analyse the results instead.