Homework #8 – Graph Partitioning

Yoke Kai Wen, 2020280598

May 6, 2021

1 Introduction

In this assignment, I implemented two traditional strategies of graph partitioning in graph processing systems, edge-cut and vertex-cut. I demonstrate the graph partitioning approaches using C++ code scripts that run on a single machine and partition the input graph into N partitions that represent a cluster of N machines that the large graph is supposed to be placed on.

2 Implementation details overview

I implemented the four graph partitioning algorithms in four scripts: `edgecut.cpp, vertex-cut.cpp, greedycut.cpp, hybridcut.cpp`. To rerun all four algorithms on all three graph datasets `roadNet-PA.graph, synthesized-1b.graph, twitter-2010.graph` with different numbers of partitions `N=2,3,4,8`, run the bash script `run_algs.sh`. (Works on my Ubuntu 20.04 machine).

```
./run_algs.sh
```

2.1 Shuffling input graph edges

Three of the graph partitioning algorithms (edge cut, random vertex cut, hybrid cut) are applied on the original binary graph files `roadNet-PA.graph, synthesized-1b.graph, twitter-2010.graph`. However, since the input edges are in sorted order according to the source nodes, it leads to the greedy heuristic algorithm assigning all the edges to the same partition. To avoid this situation, I first shuffle the input edges randomly using linux shell commands before applying the greedy heuristic algorithm. The `shuf` command works for the smaller graph files, but the twitter file is too large to be sorted in memory, hence I used `terashuf` (https://github.com/alexandres/terashuf), a quasi-shuffle algorithm for shuffling multi-terabyte files with limited memory. Before applying `shuf, terashuf` commands, I first converted the binary graph files into readable text using the `xxd` command, and then converted the shuffled lines back into binary.

```
xxd -p -c 8 synthesized-1b.graph | shuf -| xxd -p -r - synthesized-1b_shuffled.
    graph

xxd -p -c 8 twitter-2010.graph | ./terashuf -| xxd -p -r - twitter-2010_shuffled.
    graph
```

2.2 Directed and undirected graph files

I treated directed and undirected graph files the same in all four partitioning algorithms. Although the provided undirected graph files only included the edges from one direction (i.e. (1,2) only and not both (1,2) and (2,1)), the additional edge is trivial as it can always be assigned to the same partition as the counterpart edge. Hence, for simplicity, I simply ignored the additional edge when computing edge assignment.

2.3 Code structure

Before implementing the algorithms, I first initialise a vector `all_stats` to store the required output statistics (e.g. number of edges, number of vertices in each partition), as well as a vector of sets `A` to store the set of partitions each vertex belongs to. Initialising the vectors involves one full iteration of all the edges to get the maximum vertex ID number, and for hybrid cut, another full iteration is required to get the in-degrees of each vertex. The code fragment for initialising the vectors is shown below.

```
std::vector<unsigned long int> all_stats(N*4, 0); //partition ID, numMaster, totV, numEdges

for (unsigned long int p=0; p<N; p++) {
  all_stats[p*4] = p;
}

//initialise vertex location vector
std::vector<std::set<int>> A;
for (int i=0; i<=maxV; i++) {
  std::set<int> tmp;
  A.push_back(tmp);
}
```

Subsequently, in the algorithms, I only need to go through one full iteration of the input edge list. As I go through the edge list, I increment the edge counts in `all_stats` and assign partitions of each vertex to `A` accordingly. Finally, I loop through all the vertex location sets in `A` to calculate the number of vertices in each partition, as seen in the code fragment below.

```
//update number of vertices in each partition
std::set<int>::iterator it;
for (int i=0; i<=maxV; i++) {
  if (!A[i].empty()) {
    int partition = i%N;
    all_stats[4*partition + 1] ++; //add master vertex
    for (it = A[i].begin(); it != A[i].end(); ++it) {
      int partition = *it;
      all_stats[4*partition + 2] ++; //add vertex
    }
  }
}
```

3 Visualisation of graph partitioning

Figure 1 visualises each of the four partitioning algorithms on 3 partitions of the `small-5.graph`, and the detailed explanations of how each algorithm works will be explained in subsequent sections.

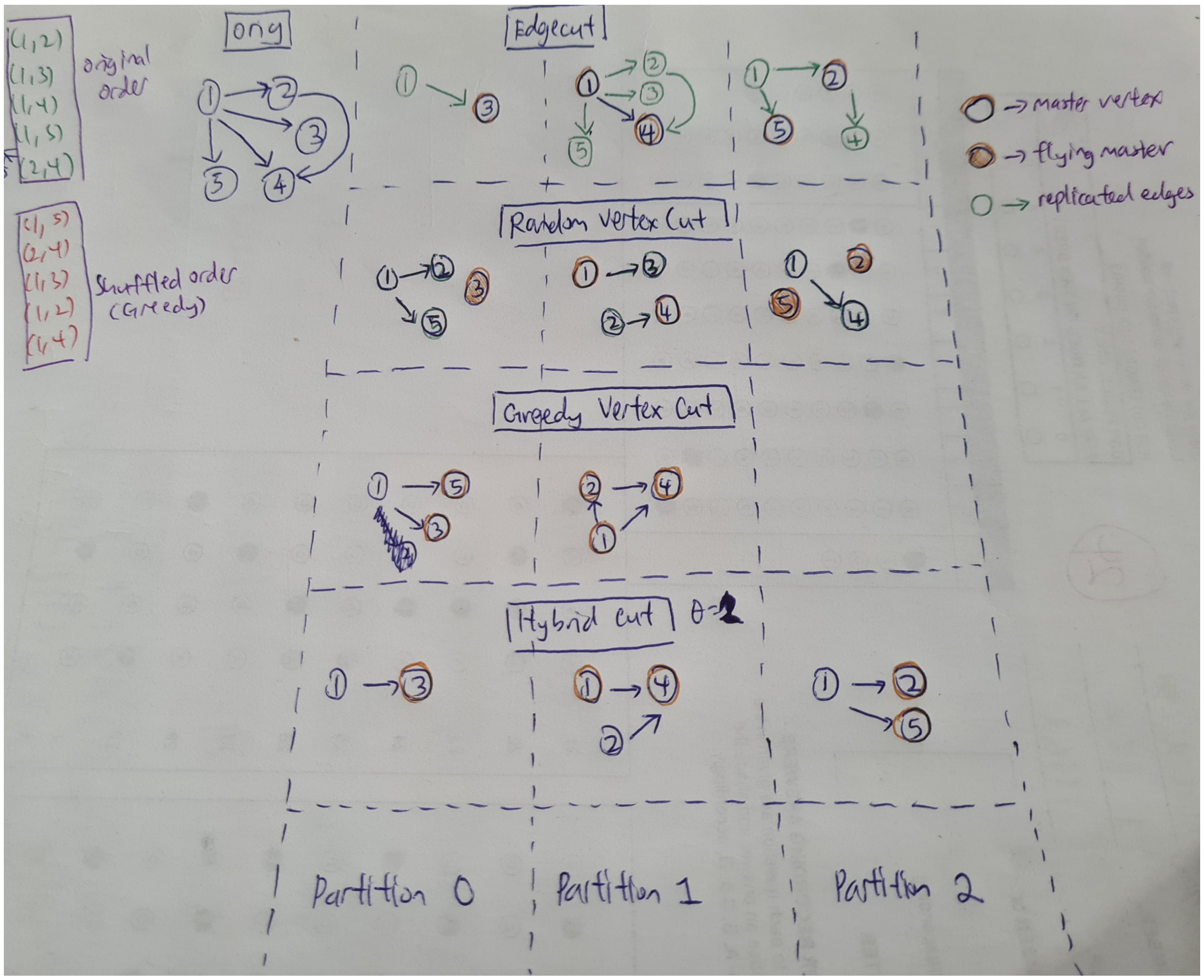

Figure 1: Illustration of edge cut, random vertex cut, greedy vertex cut, and hybrid vertex cut on small graph

4 Edge Cut

Edge cut is a very simple partitioning algorithm. Firstly, vertices are evenly assigned to partitions randomly (vertexID % #partitions), known as master vertices. Then, all the edges associated with each master vertex in each partition are then added to the respective partition, leading to replicated edges and vertices (known as mirror vertices). This process is visualised in Figure 1 for edge cut. The code fragment below shows the edge assignment process into each partition.

```
int srcLoc = src%N;
int dstLoc = dst%N;

A[src].insert(srcLoc);
A[dst].insert(dstLoc);

if (srcLoc == dstLoc) { //edge gets inserted into partition srcLoc=dstLoc
  all_stats[5*srcLoc + 4] ++;
}
```

```
else { //edge gets inserted into both partitions srcLoc and dstLoc
  all_stats[5*srcLoc + 4] ++;
  all_stats[5*dstLoc + 4] ++;

    //replicated edges
  all_stats[5*srcLoc + 3] ++;
  all_stats[5*dstLoc + 3] ++;

  A[src].insert(dstLoc);
  A[dst].insert(srcLoc);
}
```

5 Random Vertex Cut

In random vertex cut, edges are evenly assigned to partitions in a random manner (edgeID % #partitions). Master vertices are still assigned evenly and randomly to partitions (vertexID % #partitions), with mirror vertices being the extra ones associated with the edges in the partitions. This leads to the existence of flying masters in some partitions, i.e. master vertices that are not associated with any edge in that partition. This is visualised in Figure 1. The code fragment below demonstrates the edge assignment process into each partition, where `count` represents the edge count.

```
count++;

int partition = count%N;
all_stats[4*partition + 3] ++;
A[src].insert(partition);
A[dst].insert(partition);
```

6 Greedy Vertex Cut

The greedy vertex cut uses a greedy heuristic algorithm to allocate edges more efficiently than random vertex cut. Given that an edge is represented as `(u,v)` and `A(v)` represents the set of partitions that vertex v resides in, the edge placement rules are:

1. If `A(u)` and `A(v)` intersect, then the edge should be assigned to a partition in the intersection.
2. If `A(u)` and `A(v)` are not empty and do not intersect, then the edge should be assigned to one of the partitions containing `u` or `v` with the least assigned edges.
3. If only one of the two vertices has been assigned, then choose the partition from the assigned vertex.
4. If neither vertex has been assigned, then assign the edge to the least loaded partition.

The master vertices are then randomly picked from one of the vertex replicas.
Note that the greedy heuristic is only effective if the input edge list are shuffled, otherwise if the edge list is already sorted according to src/dst vertex, then all the edges are going to be assigned to the same partition. Figure 1 visualises the greedy cut when the sequence of input edges are [(1,5), (2,4), (1,3), (1,2), (1,4)]. The code fragment below demonstrates the greedy vertex cut algorithm.

```cpp
std::vector<int> v_intersection;
std::set_intersection(A[src].begin(), A[src].end(), A[dst].begin(), A[dst].end(),
    std::back_inserter(v_intersection));

//4 cases for greedy heuristic
//#case 1
if (v_intersection.size() > 0) {
  int partition = v_intersection[0]; //arbitrary partition in intersection
  all_stats[4*partition + 3] ++; //add edge to partition
  //no need update A since vertices already in partition
}
//#case 2
else if (!A[src].empty() && !A[dst].empty()) { //if both sets not empty, and no
    intersection
  unsigned long int min = ULONG_MAX;
  int minPart = 0;
  int minNode = src;

  std::set<int>::iterator it;
  for (it = A[src].begin(); it != A[src].end(); ++it) {
    int partition = *it;
    int numEdges = all_stats[4*partition + 3];
    if (numEdges < min) {
      min = numEdges;
      minPart = partition;
    }
  }

  for (it = A[dst].begin(); it != A[dst].end(); ++it) {
    int partition = *it;
    int numEdges = all_stats[4*partition + 3];
    if (numEdges < min) {
      min = numEdges;
      minPart = partition;
      minNode = dst;
    }
  }

  all_stats[4*minPart + 3] ++; //add edge to partition
  if (minNode == src) {
    A[dst].insert(minPart);
  }
  else {
    A[src].insert(minPart);
  }
}
//#case 3
else if (A[src].empty() && A[dst].empty()) { //if both sets empty
  unsigned long int min = ULONG_MAX;
  int minPart = 0;
  int minNode = src;

  for (int p=0; p<N; p++) {
    unsigned long int numEdges = all_stats[4*p + 3];
    if (numEdges < min) {
      min = numEdges;
      minPart = p;
    }
  }

  all_stats[4*minPart + 3] ++; //add edge to partition
  A[src].insert(minPart);
  A[dst].insert(minPart);
}
//#case 4
else { // one empty, one not empty
  if (!A[src].empty() ) {
    int partition = *(A[src].begin());
    all_stats[4*partition + 3] ++; //add edge to partition
    A[dst].insert(partition);
  }
  else {
    int partition = *(A[dst].begin());
    all_stats[4*partition + 3] ++; //add edge to partition
    A[src].insert(partition);
  }
}
```

7 Hybrid Vertex Cut

The hybrid vertex cut uses differentiated partitioning of low-degree and high-degree vertices, where degree here refers to the in-degree of vertices. The algorithm first looks at the dst vertex of the edge. If the in-degree of the dst vertex is below a threshold, the low cut is adopted, where the edge is assigned to the partition (dst % #partitions). Otherwise, the high cut is adopted, and the edge is assigned to the partition (src % #partitions). Figure 1 illustrates the hybrid vertex cut for threshold=2 (i.e. all the vertices in the small graph are low-degree). The code fragment below demonstrates the hybrid vertex cut algorithm. Before the algorithm is executed, the in-degree of each vertex is computed. The threshold is set to 3, 13, 100 for roadNet, synthetic-1b, and twitter-2010 graphs respectively, by observing the distribution of in-degrees in each graph and trial and error with different thresholds to get optimal replication factor (see next section).

```
int srcLoc = src%N;
int dstLoc = dst%N;

//differentiated edge assignment based on dst in-degree
if (inDeg[dst] <= thresh) {
  A[src].insert(dstLoc);
  A[dst].insert(dstLoc);
  all_stats[4*dstLoc + 3] ++; //add edge
}
else {
  A[src].insert(srcLoc);
  A[dst].insert(srcLoc);
  all_stats[4*srcLoc + 3] ++; //add edge
}
```

8 Results and analysis of partition algorithms

8.1 Edge cut vs vertex cut

Only one algorithm is edge cut based, while the other three are based on vertex cuts. The problem with edge cuts is that many edges are replicated, while vertex cuts avoid this problem by assigning edges evenly across partitions.

8.2 Replication factor

However, vertex cuts face the problem of having too many vertices replicated across partitions. This is measured with `rep factor = (total number of vertices in all partitions) / (total number of master vertices)`, which should be minimised.

8.3 Different implementations of vertex cut

Different implementations of vertex cuts result in different replication factors. The most basic implementation is the random vertex cut. The hybrid vertex cut improves on this by attempting to reduce the replication factor for low-degree vertices. The greedy heuristic vertex cut attempts to optimise the edge allocation by coordinating the different partitions which communicate to find out the locations of each vertex in each partition, and use this information to decide where to allocate the next edge.
Figures 2, 3 and 4 show the replication factor against number of partitions graphs for the three datasets, RoadNet, Synthetic-1b, and Twitter-2010 respectively.

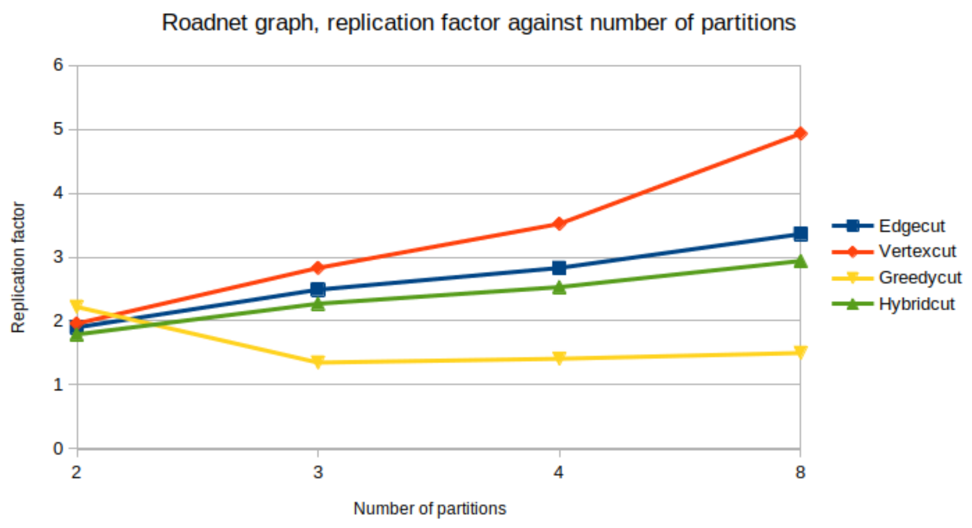

Figure 2: Replication factor against number of partitions for RoadNet graph

From Figure 2, we see that the HybridCut did not perform very well and is much worse than GreedyCut. This is because the vertices in RoadNet have very small in-degrees (less than 5) and the in-degrees are also evenly distributed, hence HybridCut, which is supposed to work better on skewed graphs, did not demonstrate its advantage here.

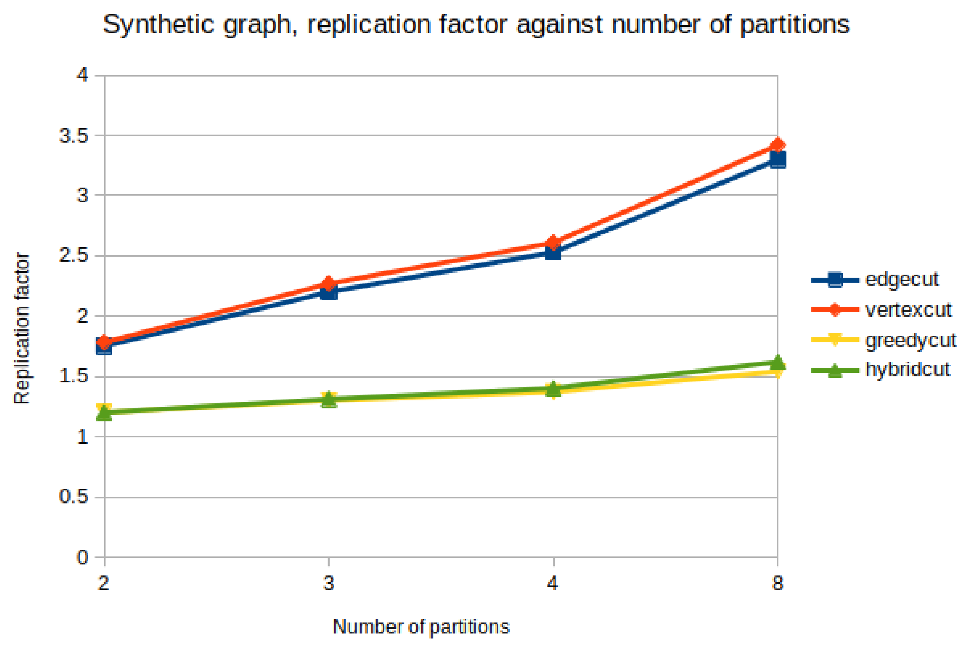

Figure 3: Replication factor against number of partitions for synthesized graph

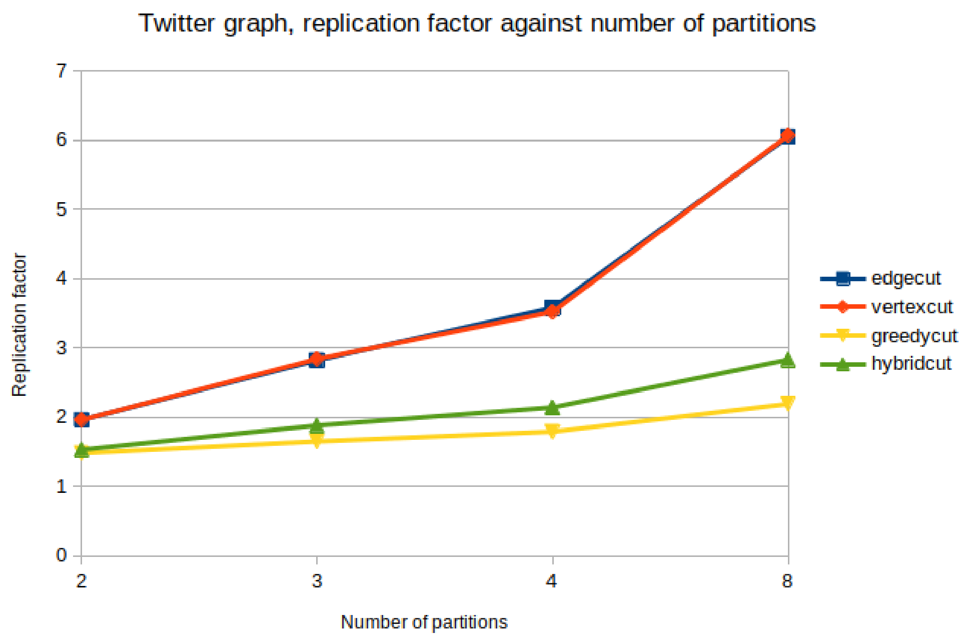

Figure 4: Replication factor against number of partitions for twitter graph

Figures 3 and 4 showed similar trends in performance, since they are larger graphs with greater complexity and high-degree vertices and greater skewness. The basic algorithms edgecut and vertexcut performed significantly worse than the other improved versions of vertex cut, with greedycut performing the best out of all the algorithms. This makes sense as greedycut uses global information to decide edge allocation, and reduces replication factor at the expense of communication cost. To be fair, the hybridcut also requires global information in the form of node degree, however the extent of communication required is much smaller than that required from greedycut, so hybridcut can also be seen as a good tradeoff between replication factor and communication cost.