A Distributed Algorithm for Global Min-Cut

David Flatow
Stanford University
dflatow@stanford.edu

Daniel Penner
Stanford University
dzpenner@stanford.edu

Abstract

In this paper, we present a distributed algorithm to solve the global min-cut problem based on the Karger-Stein sequential algorithm. Like the Karger-Stein algorithm, the algorithm presented is probabilistic, and outputs the correct global minimum cut with probability $1 - \frac{1}{n}$. We present experimental results which suggest that communication costs do not dominate computational complexity as the problem size scales.

1. Introduction

1.1. Problem Definition

Given an undirected weighted graph $G = (V, E)$ with $|V| = n$ and $|E| = m$ and edge weights $w_e \geq 0$ for $e \in E$, we define a **cut** of G to be a subset $S \subseteq V$ of vertices. The **cutsize** of S is defined as $c(S) = \sum_{u \in S} \sum_{v \in S^c \cap N_u} w_{(u,v)}$, where $N_u = \{v \in V : (u, v) \in E\}$ and $S^c = V \setminus S$. We would like to find the cut S with minimal cutsize.

1.2. Karger's Algorithm

Although other approaches had been developed previously, David Karger's randomized algorithm, published in 1993, has proven to be the most enduring published thus far [2]. Each iteration of the algorithm is performed as follows:

function **contract**(G) {
while $|V| > 2$ **do**
  choose $e \in E$ randomly
  $G \leftarrow G \setminus \{e\}$
end while
return G
}

The result of one iteration is a cut of the graph, whose probability of being the minimum cut is $\Omega(n^{-2})$. Therefore, iterating $O(n^2 \log n)$ times gives a lower bound of $1 - \frac{1}{n}$ on the probability of finding the minimum cut, and since each iteration takes $O(n^2)$ operations to complete (in some implementations $O(m)$), we obtain a total runtime of $O(n^4 \log n)$ runtime, or $O(mn^2 \log n)$, depending on the implementation.

1.3. Karger-Stein Algorithm

An alteration to the algorithm proposed in 1996 by Karger and Clifford Stein provided a significant boost in total runtime. The idea is to cut the number of required iterations per trial by an order of magnitude by simulating many trials in each trial by branching recursively. More concretely, we run the contraction algorithm twice in parallel, contracting until G has t vertices, where t is some pre-determined integer, and then recursively call the algorithm on these two graphs of size t, reverting to brute-force computing mincut for the base case of graphs of size $\leq C$ for some pre-determined constant C, and keeping the min of all recursive subcalls as the final answer for the trial. The pseudo-code below describes this process.

function **contract**(G) {
while $|V| > t$ **do**
  choose $e \in E$ randomly

$G \leftarrow G \setminus \{e\}$
end while
return G
}
function **karger_stein**(G) {
if $|V| \leq C$ **then**
return $mincut(G)$
else
return $min\{karger\_stein(contract(G,t)), karger\_stein(contract(G,t))\}$
end if
}

Choosing $t = \lceil n/\sqrt{2} \rceil$, we get a runtime recursion of $T(n) = 2T(n/\sqrt{2}) + O(n^2)$, which by Master theorem yields $T(n) = O(n^2 \log n)$ per trial. To obtain the same probability bound of success as in Karger's algorithm of $1 - 1/n$, we need to perform only $\Omega(\log^2 n)$ trials, so that we get a total time bound of $O(n^2 \log^3 n)$, a significant improvement.

2. Distributed Algorithm

2.1. Model Assumptions

The graph model we assume in this paper is such that vertices can communicate only with their neighbors in the graph, and furthermore that vertices have storage linear and computing capacity polynomial in their degree.

2.2. Variable Definitions

As before, we are working with a weighted undirected graph $G = (V, E)$, with non-negative edge weights $w_e \geq 0$ for $e \in E$. We define $N_u = \{v \in V : (u, v) \in E\}$ to be the neighborhood of $u \in V$. In the implementation, each vertex $u \in V$ stores the quantities:

1. the vertex id $id(u) \in \{1, \ldots, n\}$
2. the weights $w(u, v)$ for $v \in N_u$
3. the ranks $r(u, v)$ for $v \in N_u$, randomly set at each iteration
4. the maximum rank $max(u) = \max_{v \in N_u} r(u, v)$, which will be updated
5. the status $s(u) = 1$, set to zero if $w(u, v) = 0$ for all $v \in N_u$ at any time
6. the group id $g(u) = id(u)$, which is updated as the graph contracts

2.3. Algorithm Summary

As in the sequential Karger's algorithm, we define a trial below that produces a min-cut with probability bounded below by n^{-2}, so that the same number of trial as before produces a min-cut with probability bounded below by $1 - \frac{1}{n}$.

At the beginning of each trial, we initialize the vertex group id's and edge ranks. Then we loop over the following until there remain two groups (this will in general require $n - 2$ iterations):

First, we find the maximum rank, and broadcast it, along with the id of the maximum rank vertex, over all machines. Then we filter out the edge with max_rank, to contract it, and run connected components algorithm to update the vertex id's according to the contraction. Next, we filter out all edges that become self-loops as a result of the contraction (that is, edges connecting vertices with the same id), by setting their weights to zero.

Finally, once there are two groups remaining, the trial's guess for the global mincut value is computed by summing the weights of the remaining edges, and is broadcast via messages to all the vertices.

3. Runtime and Communication Analysis

In the analysis that follows, we recall that $n = |V|$, $m = |E|$, and we define K to be the number of machines used, and $M = m/K$ to be the number of edges stored per machine.

First we analyze the computation time per trial. In each iteration, we start by finding the max edge rank. This is done by finding the max edge in each machine in parallel ($O(M)$ time), and then finding the max rank over all machines' max ranks ($O(K)$ time), for a total of $O(M + K)$ time. Then, if (u, v) has max rank, the group ID's of u, v are broadcast over all machines. Next, updating vertices' group ID's using the broadcast tuple occurs within each machine, and so computation time is $O(M)$. Finally, we iterate over all edges in each machine and remove those connecting two vertices of the same group ID, and so the computation time is again $O(M)$. So for each iteration we incur $O(M + K)$ computation time, and since there are $n - 2 = O(n)$ iterations per trial, each trial takes $O(n(M + K))$ computation time.

Next we analyze the shuffle size per trial. First, in each iteration, we communicate the max edge rank of each machine to compute the overall max rank, which causes shuffle proportional to the number of machines, $O(K)$, and then we communicate the final max rank back to each machine, which incurs $O(K)$ shuffle as well. Next, we set the weights of all edges which have become self-loops to zero. This occurs within each machine so there is no shuffle. So the total shuffle size for each iteration is $O(K)$, and therefore since there are $n - 2 = O(n)$ iterations per trial, each iteration incurs shuffle size $O(nK)$.

4. Spark Implementation

Our spark implementation takes as input a Graphx graph and returns the global minimum cut value as a double. The operations, map and reduce, implemented in spark are beautifully suited to our algorithm. In particular, outside of the operation to find the max rank edge amongst all edges in the graph, all other operations are done using map. This allows for extreme scalability in terms of distributing the algorithm on a cluster of commodity machines as the communication cost is minimal. In particular, with combiners, our distributed KS-algorithm has a shuffle size of $O(\log^2(n)nK)$.

4.1. Data Structures

The workhorse data structure we use in the spark implementation is an RDD of a `Triplet` data structure that contains the following fields: `srcGroup, dstGroup, rank, weight`.

4.2. Code

```
sc.setCheckpointDir(checkpointDir)

case class Triplet (
        srcGroup: Long,
        dstGroup: Long,
        weight: Double,
        rank: Long
)

type MyGraph = Graph[Long, Triplet]

// generate random edge ranks
def genRandRank(): Long = { //
  val rand = new Random()
  rand.nextLong() // TODO make this number of node
}

// calculate the minimum cut for a set of edges
def calc_min_cut_value(edges: RDD[Triplet]): Double = {
  math.min(edges.map(e =>
                        if (e.srcGroup > e.dstGroup) e.weight
                        else 0.0)
                                .reduce(_ + _),
           edges.map(e =>
                if (e.srcGroup < e.dstGroup) e.weight
                else 0.0)
                        .reduce(_ + _))
}

def contract(edges: RDD[Triplet]): RDD[Triplet] = {
    val max_rank = edges.reduce((a, b) => if (a.rank > b.rank) a else b)
    val oldGroup = math.min(max_rank.srcGroup, max_rank.dstGroup)
        val newGroup = math.max(max_rank.srcGroup, max_rank.dstGroup)
    edges.map(e => if (e.dstGroup == oldGroup) e.copy(dstGroup=newGroup)
                   else if (e.srcGroup == oldGroup) e.copy(srcGroup=newGroup)
                   else e)
         .filter(e => (e.srcGroup != e.dstGroup))
}

def mincut(orig_edges: RDD[Triplet], numVertices:Long): Double = {

  var components = numVertices
   // first stage contractions
  while (components > 2){
    edges = contract(edges)
    components -= 1
    if ((components % 500) == 0){
        edges.checkpoint()
      }
```

```
  }
  calc_min_cut_value(edges)
}

def fast_mincut(orig_edges: RDD[Triplet],
                numVertices:Long): Double = {

  if (numVertices < 6) {
      mincut(orig_edges, numVertices)
  } else {
    val t =  math.ceil(1 + numVertices / math.sqrt(2)).asInstanceOf(Integer)
    var edges_one: RDD[Triplet] = orig_edges.map(e =>
                    e.copy(rank=genRandRank()))
    var edges_two: RDD[Triplet] = orig_edges.map(e =>
                    e.copy(rank=genRandRank()))
    edges_one.cache()
    edges_two.cache()
    var components = numVertices

    // first stage contractions
    while (components > t){
      edges_one = contract(edges_one)
      edges_two = contract(edges_two)
      components -= 1
      if ((components % 500) == 0){
        edges_one.checkpoint()
        edges_two.checkpoint()
      }
    }
    math.min(fast_mincut(edges_one, components), fast_mincut(edges_two, components))
  }
}

def min_cut(g: MyGraph, trials:Long): Double = {
  var min_cut_value: Double = Double.PositiveInfinity
  val begin: Long = 1
  val n = g.numVertices
  val edges = g.edges.map(e => e.attr)
  edges.cache()
  begin to trials foreach { _ =>
    val cut = fast_mincut(edges, t, sub_trials, n)
    min_cut_value = math.min(min_cut_value, cut)
  }
  min_cut_value
}

def run(): Double = {
        val numVertices: Integer = 2000
        val numEdges: Integer = numVertices * (numVertices - 10)
  val trials = math.pow(math.log(numVertices), 2)
        val g: MyGraph = GraphGenerators.logNormalGraph(sc,
                numVertices=numVertices, numEParts=32)
      .mapVertices((id, _) => id)
```

```
        .mapTriplets(e => Triplet(e.srcGroup, e.dstGroup, e.attr, 0))
        .subgraph(epred = triplet => (triplet.srcId != triplet.dstId))
          g.partitionBy(PartitionStrategy.EdgePartition2D)
          min_cut(g, trials)
}
run()
```

4.3. Discussion

We quickly abandon using the graphx data structure within the implementation of MinCut to avoid any overhead associated with maintaining the graph structure. For example, when filtering vertices graphx removes edges to ensure the graph remained valid. Secondly, we since our implementation really only performed operations on triplets we saved greatly by only operating on RDDs of triplets rather than the entire vertex-cut data structure.

5. Experiments

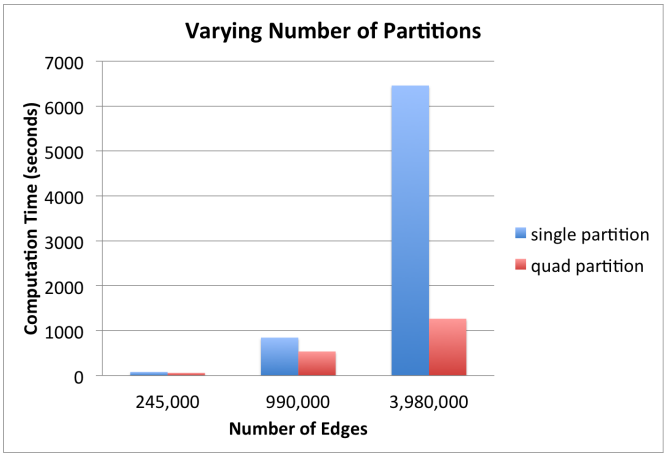
We ran computational experiments in a distributed utilizing the infrastructure of AWS EC2 instances by way of the DataBricks environment. Our experiments fell into two classes. First, we ran experiments to determine the effect of increasing the number of partitions into which the data was distributed. Second, we experimented with distributing the algorithm across 8 machines and recorded the performance differences (in terms of computation time).

5.1. Varying Number of Partitions

To get a handle on the impact of increasing the number of partitions, thus allowing computations to be parallelized within a machine, we experimented with one and four partitions per machine for equally sized graphs. We ran this experiment on graphs with (vertex, edge) sizes:

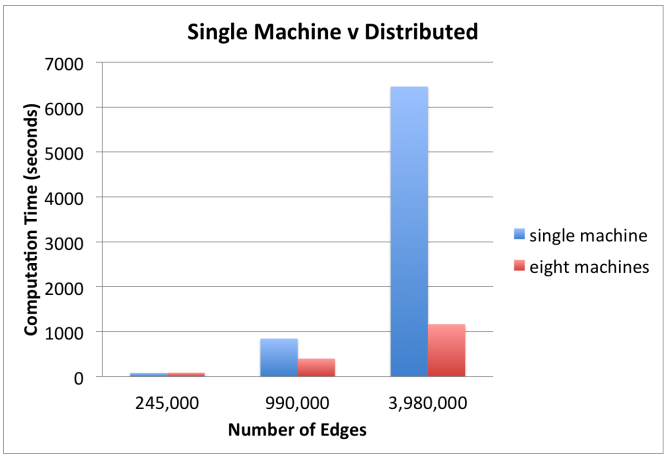
1. Vertices $= 500$, Edges $= 245,000$
2. Vertices $= 1000$, Edges $= 990,000$
3. Vertices $= 2000$, Edges $= 3,980,000$

The following results show that by utilizing multiple cores on a single we can achieve achieve proportionately faster processing time. This should not come as a huge surprise to the reader. Rather, this experiment stands to show that our algorithm is not adversely impacted by partitioning.



5.2. Varying Number of Workers

The most important experiments we ran were comparing the total run time of the algorithm on a single machine cluster to that of an eight machine cluster. Quite interestingly, we found that going from one machine to eight machines had roughly same impact as going from one partition per machine to four. Thus we only lose a factor of 2 due most likely to communication costs when comparing performance on K processors within the same machine to K machines on a network. This has important implications.



6. Conclusion

Our experiments detailed above suggest that as we scale the processing power used to perform the algorithm, the loss attributed to adding machines instead of adding processors is a constant factor. Thus, the communication costs do not dominate the algorithm's complexity as we scale up the graph and, accordingly, the number of machines. Furthermore, since the algorithm's complexity is a function of m, it takes advantage of a graph's sparsity, and should perform much well on sparse graphs.

References

[1] Shine, S; Murali Krishnan, K. Leveraging social media for scalable object detection. *Pattern Recognition 45*, 2012.

[2] Karger, D. Global Min-cuts in RNC and Other Ramifications of a Simple Mincut Algorithm. *Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1993.