Balanced Graph Partitioning

Index

Introduction: Basic notations and definitions

G = (V, E)
Input graph

(k,1+ε)-balanced partitioning
A minimum cost partitioning of G into k parts such that each part contains at least (1+ε)(n/k) nodes.

T-Partitioning
A partitioning is a T-partitioning if it only contains subsets of T (they correspond to some node of T).

Coarse T-Partitionings
It is a T-partition with no small subsets. More precisely it does not contain subsets V_{v_t} for which the parent subset V_{v_p} (with v_p a parent of v_t in the tree) contains less than εn/3k nodes

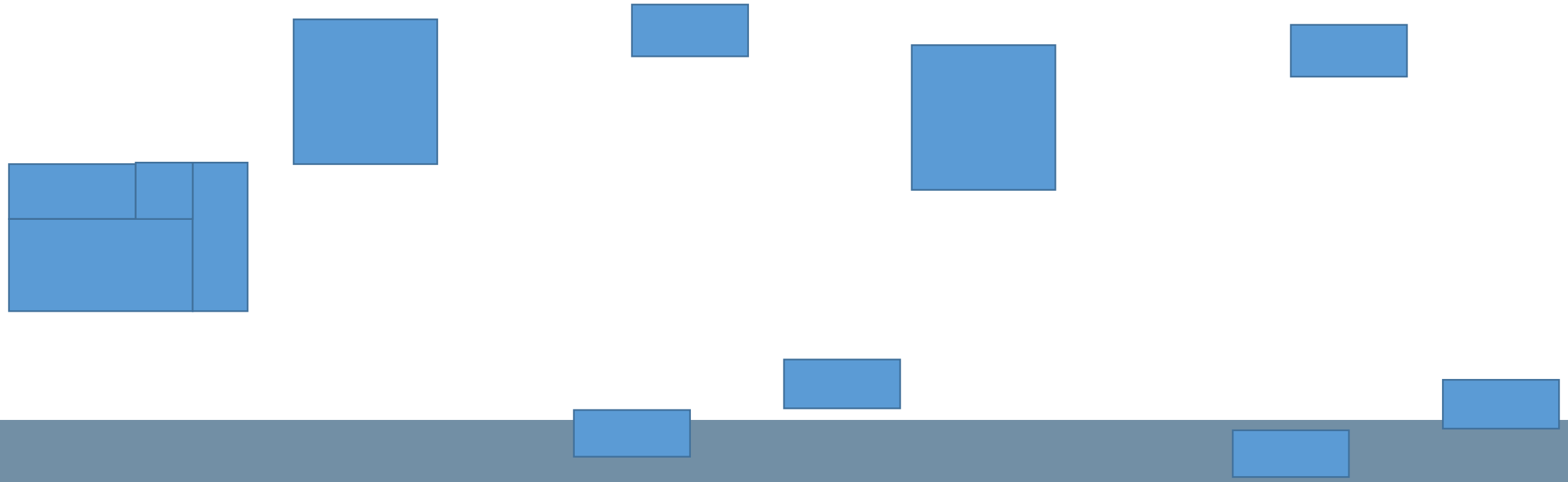
Cost(P)
Denote the capacity of edges cut by the partitioning P

Definition of the general problem

We are given n =3k integers a1,a2,...,an and a threshold S such that:

$$S/4 < a_i < S/2 \qquad \sum_{i=1}^{n} a_i = kS$$

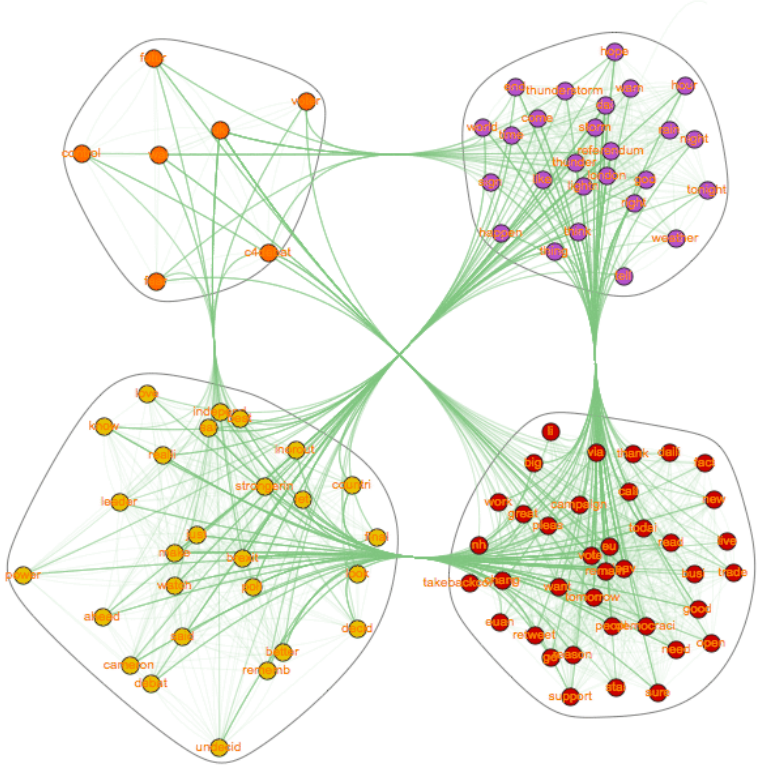
The task is to decide if the numbers can be partitioned into triples such that each triple adds up to S.



Graph application

Investigate the (k,1+ε)-balanced partitioning problem, that is the problem of finding a minimum cost partitioning of G into k parts such that each part contains maximum(1+ε)(n/k) nodes

$0 \leq \boldsymbol{\varepsilon} \leq 1$
k = number of final partitions
n = |V|



Real use case example:

Input: a set of Data or Processes

Goal: assign them in a balanced way for parallelization

s.t. minimizing the communication

Algorithm: General Overview

1) Recursively decomposition of the graph

2) In the following we remove all nodes from T that correspond to sets of size smaller than $\varepsilon n/3k$, and we mark all nodes that correspond to sets larger than $(1 + \varepsilon)(n/k)$ as invalid.

3) Use dynamic programming to define the T-partitioning and thus the approximated solution.

Appriximated Ratio

$$\mathrm{cost}(\mathrm{OPT}''_k) \leq O(\log^2 n/\epsilon^4) \cdot \mathrm{cost}(\mathrm{OPT}_k)$$

The cost is based on the number of edges that are cutted during the partitioting, related to the optimal solution

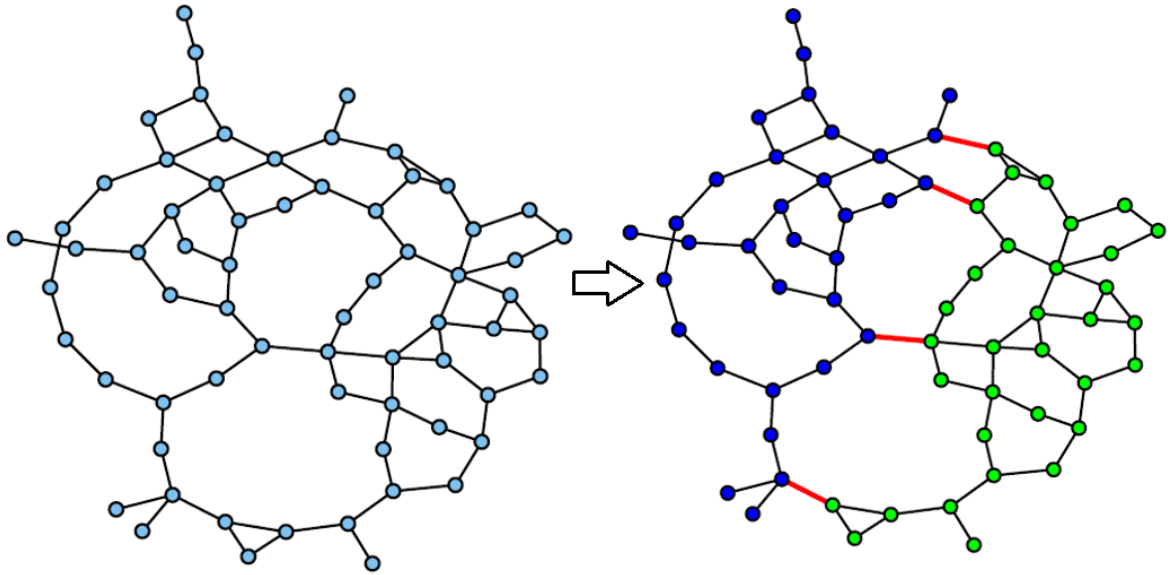
OPT’’ is the sub-optimum solution obteined using this algorithm.

OPT’’ is the optimal partition

Decomposing the graph (1)

In the first pre-processing phase the graph is decomposed via a recursive decomposition scheme.

This means we use a separation algorithm that gets a part of the graph as input, and outputs two non-empty subsets of this part

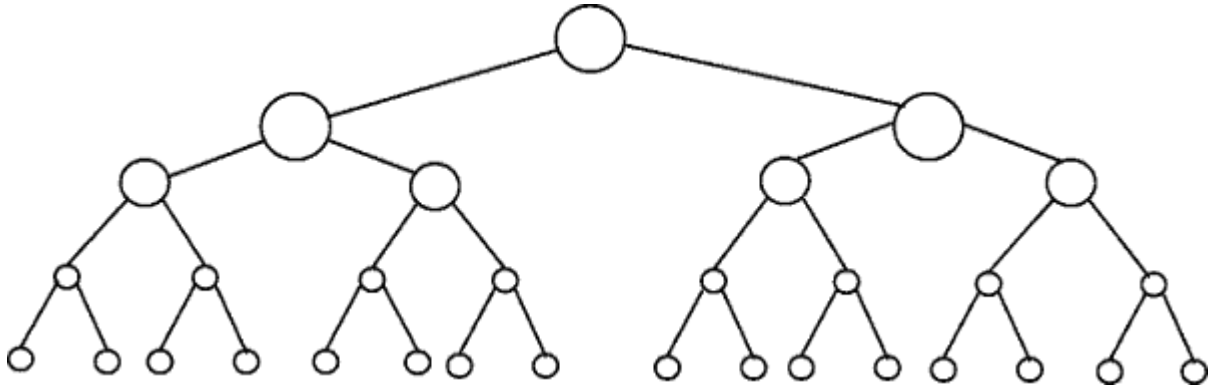


Decomposing the graph (2)

First we have to define ε’ as: $\epsilon' := \frac{\epsilon/3}{1+\epsilon/3}$

In each divide step the algorithm tries to find a minimum ε’/2-balanced separator.

It is a partition of an input set V into two parts with size more than (ε’/2)·|V|



Algorithm: Decomposing the graph (3)

```
listOfNodes = subTree.data.nodes()
nodeSource = random.choice(listOfNodes)
listOfNodes.remove(nodeSource)
nodeDestination = random.choice(listOfNodes)

graphNew = self.fromNetworkXtoiGraph(subTree)
```

It is mandatory to define a source node and a destination node for the cutting function. After that, the graph is converted from a NetworkX data-structure to a igraph data-structure

Finally the new partition are created according to the ”Push-relabel algorithm”. There will be also a loop in order to check the dimensions of the two partitions created.

```
mc = graphNew.mincut(int(nodeSource),int(nodeDestination))
partition = mc.partition
```

Pruning (1)

$$\frac{\epsilon n}{3k}$$

Small nodes are eliminated:

They cannot be part of a coarse T-partitioning. The elimination of these nodes derives from the passage from OPT'_k to OPT''_k

$$(1+\epsilon)\frac{n}{k}$$

Big nodes are eliminated:

They cannot fit in any partition. The goal of the algorithm, indeed, is to have partitions smaller than this size. It derives from the passage from OPT_k to OPT'_k

$$\mathrm{height}(T_i) \leq \log(\frac{1+\epsilon}{\epsilon/3}) / \log(\frac{1}{1+\epsilon'}) = O(\frac{1}{\epsilon}\log(\frac{1}{\epsilon}))$$

Pruning (2)

We begin the pruning directly in the graph decomposition.

In this way nodes that are too small are not even calculated.

```
if (len(partition[0]) > threshold):
    self.division(leftChild, threshold)
if (len(partition[1]) > threshold):
    self.division(rightChild, threshold)
```

We eliminate the biggest nodes by scanning the tree and save in a list the root of the trees that are small enough (number of nodes below the threshold) $(1+\epsilon)\frac{n}{k}$

```
def removeBigNodes(self,root,threshold,listTrees):
    if(root.data.number_of_nodes()<threshold):
        listTrees.append(root)
        return
    else:
        self.removeBigNodes(root.left,threshold,listTrees)
        self.removeBigNodes(root.right, threshold, listTrees)
```

Sub-Trees height

The height of sub-trees is independent from the number of nodes

$$\text{height}(T_i) \leq \log(\frac{1+\epsilon}{\epsilon/3})/\log(\frac{1}{1+\epsilon'}) = O(\frac{1}{\epsilon}\log(\frac{1}{\epsilon}))$$

Number of nodes of each tree:

$$2^{O(1/\epsilon \cdot \log(1/\epsilon))}$$

By considering a complete tree. This happen when the graph is decomposed in a perfect binary way(e.g. balanced partition)

Number of partitions for each tree:

$$2^{2^{O(1/\epsilon \cdot \log(1/\epsilon))}}$$

Since each node can be substitutes with its sons, the number of partition is equal to 2 up to the number of nodes

Partitions

- Using dynamic programming we calculate the cost for each partition in each tree.
- The cost of a partition is equal to the cost of the previous partition plus the cost for substituting one node of it with the two sons.
- We save the final result in an array of arrays.
- By combining the partitions for each tree we can generate all the possible partitions of the graph.

- P1 = [(a,b,c),(d,e)]
- P2 = [(a,b),(c),(d,e)]
- Cost(P2) = cost(P1) + cost for separate c form (a,b)

$$O(2^{2^{O(1/\epsilon \cdot \log(1/\epsilon))}})$$

def createPossiblePartitions()

Calculate recursively the cost of each partition of a tree; then proceed with the next tree:

```
def createPossiblePartitions(self,listNodes,listTreesCosts,cost):
    listTreesCosts.append(cost)
    for i in range(len(listNodes)):
        if(listNodes[i].left !=None):
            father = listNodes[i]
            listNodes.pop(i)
            listNodes.insert(i,father.left)
            listNodes.insert(i+1, father.right)
            newCost = self.calculateCost(father.data,father.left,father.right)
            self.createPossiblePartitions(listNodes,listTreesCosts,cost+newCost)
            listNodes.pop(i)
            listNodes.pop(i)
            listNodes.insert(i,father)
```

Each node is substituted with its sons until no more combination can be created

G - Vector

Let t denote the number of (1+/2)-powers in the interval (n 3k,(1+)n k)

$$r_s := (1+\epsilon/2)^{\lceil \log_{1+\epsilon/2} \frac{\epsilon n}{3k} \rceil} \cdot (1+\epsilon/2)^{s-1}$$

Nodes are converted into pieces of fixed size. In this way the complexity for the calculation of the bin packing problem is drastically reduced $O(p^{2s})$

$\frac{1+\epsilon/2}{\epsilon/3} k = O(k/\epsilon)$

Since the smallest size is $\frac{\epsilon n}{3k}$, than, the number of pieces is bounded by (k/epsilon) Moreover, having fixed the possible sizes, this number is bounded by (t+1) so, the overall complexity is $O((\frac{k}{\epsilon})^{t+1})$

def getGVector():

```
def getGVector(self):
    vector_gValue = []
    val = math.pow((1 + self.epsilon/2),
                   math.floor(math.log(self.epsilon*n/(3*k),
                                       (1+epsilon/2))))
    while(val < ((1+epsilon)*(n/k))):
        vector_gValue.append(val)
        val *= (1 + epsilon / 2)
    return vector_gValue
```

The possible fixed size are calculated

This array is calculated only once

```
for el in tempPartition:
    partition.append(el)
    numberOfNodes = len(el)
    num = math.ceil(math.log(numberOfNodes,1 + epsilon/2))
    num = math.pow(1 + epsilon/2,num)
    i = bisect.bisect_left(GVector, num)
    if i == len(GVector):
        i -= 1
    currentG[i] += 1
    packer.items.append(Item('A', round(GVector[i])))
```

A G-Vector is created for each partition

A piece of that size is added to the packer that will solve the bin packing problem

Partition evalutation: method 1

For the evaluation of the different partitions with have two ways:

Generate all the possible partitions by combining the cost vector for each tree. Save the result in an array and order it. Scan the ordered array and find the feasible partition at lowest cost.

```
possiblePartitionsCost = np.array(list(itertools.product(*partitions)))
possiblePartitionsCostMerged = possiblePartitionsCost.sum(axis=1)
return (np.argsort(PossiblePartitionsCostMerged)),partitions
```

- It is very fast(almost immediate)
- Not always the final array can be stored in a RAM (depends on the initial parameters)

Partition evalutation: method 2

Scan the trees in a recursive way and store in a dictionary the key(numberOfTree,vector G) combined with its result.

```
if((index,tuple(newVectorG)) in gDictionary):
    cost, minP = gDictionary[(index,tuple(newVectorG))]
else:
    cost, minP = self.getBestPartitionPaper(listTrees,
                                            possiblePartitionsCost, (vectorG + currentG),
                                            index + 1, GVector, packer, gDictionary)
    gDictionary[(index,tuple(newVectorG))] = cost,minP
```

Before invoking the function the algorithm checks if any occurrence has already called the function with the same parameter. In this case the results are already known and the recursion stops.

Feasibility study for the partitions

At the end of the recursion (when all the sub-tress has been scanned) the algorithm verify the feasibility of the vector G; it returns zero if it is feasible infinite otherwise.

```
if(index == len(listTrees)):
    packer.items = []
    for el in range(len(vectorG)):
        for j in range(int(vectorG[el])):
            packer.items.append(Item('A', round(GVector[el])))
    vectorG = tuple(vectorG)
```

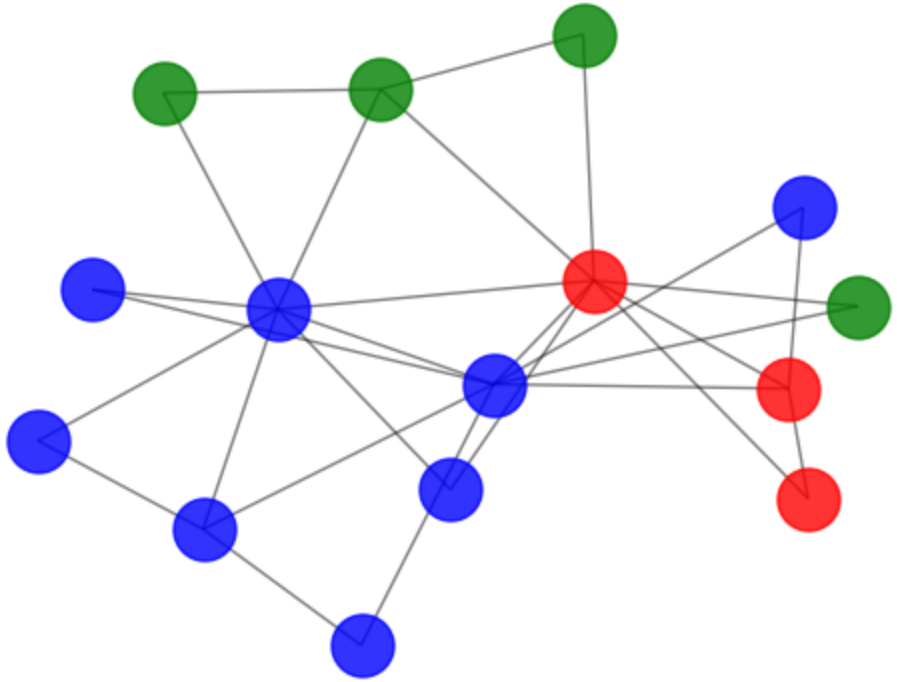
Going back in the recursion the algorithm sum the returned value with the cost of each T_i partition and calculate the one with the least cost.

```
costs.append(cost+possiblePartitionsCost[index][i])

min = np.argmin(costs)
```

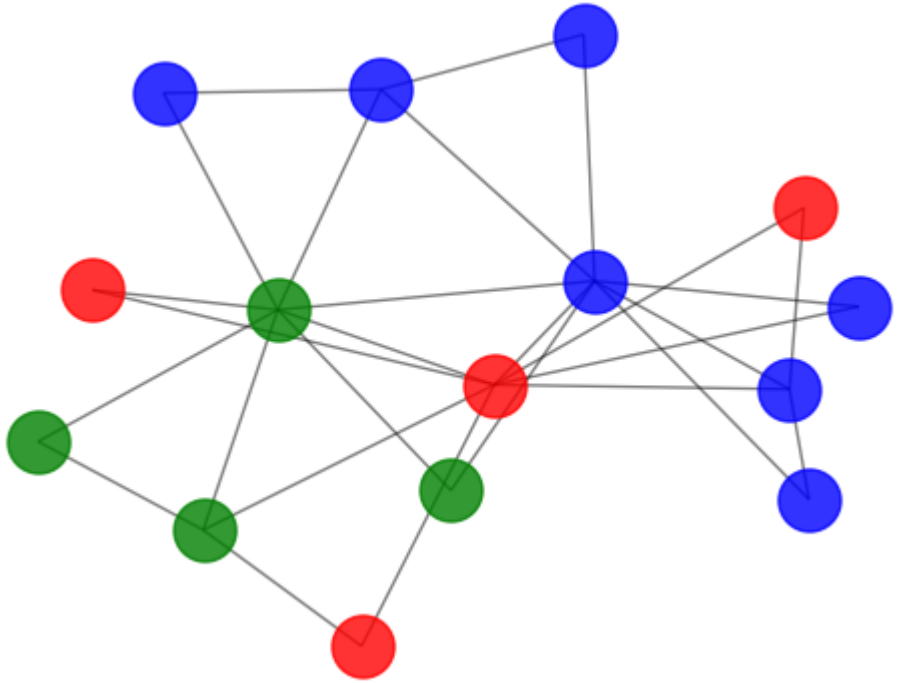
Graphical comparison (1)

Best partition (OPT)



Cost(OPT) = 11

Our algorithm (OPT’’)



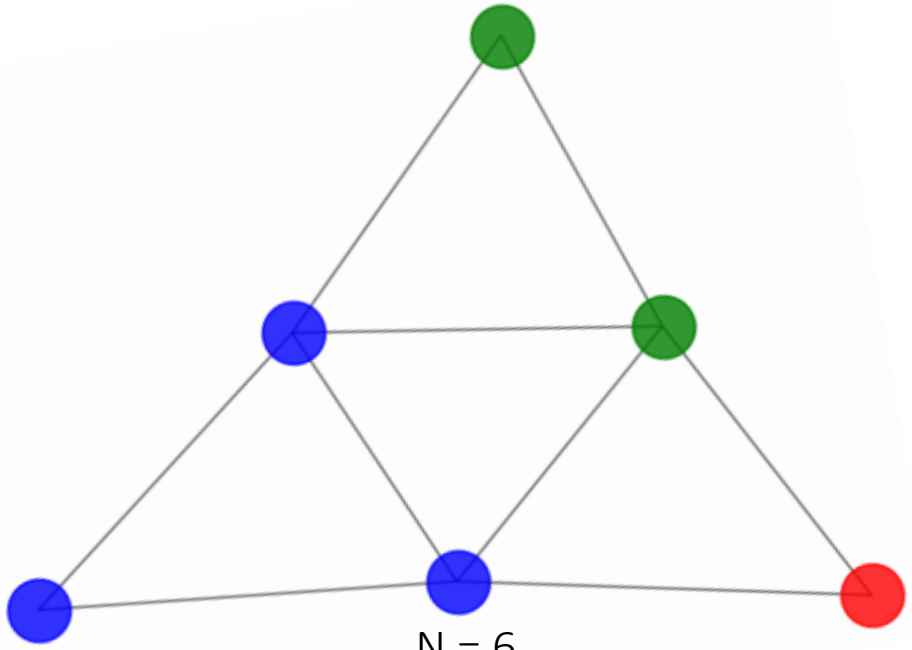
Cost(OPT’’) = 12

$$\mathrm{cost}(\mathrm{OPT}''_k) \leq O(\log^2 n/\epsilon^4) \cdot \mathrm{cost}(\mathrm{OPT}_k)$$

Graphical comparison (2)

The final graph is partitioned in three different pieces of size below to

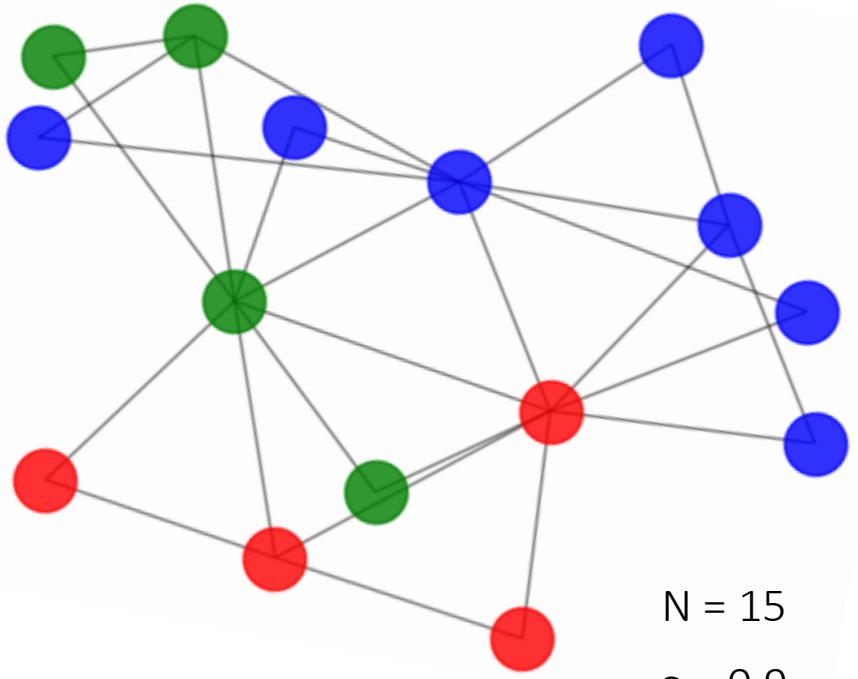
$$(1+\epsilon)\frac{n}{k}$$



N = 6

ε = 0.9

$(1+\epsilon)\frac{n}{k}$ = 3.8

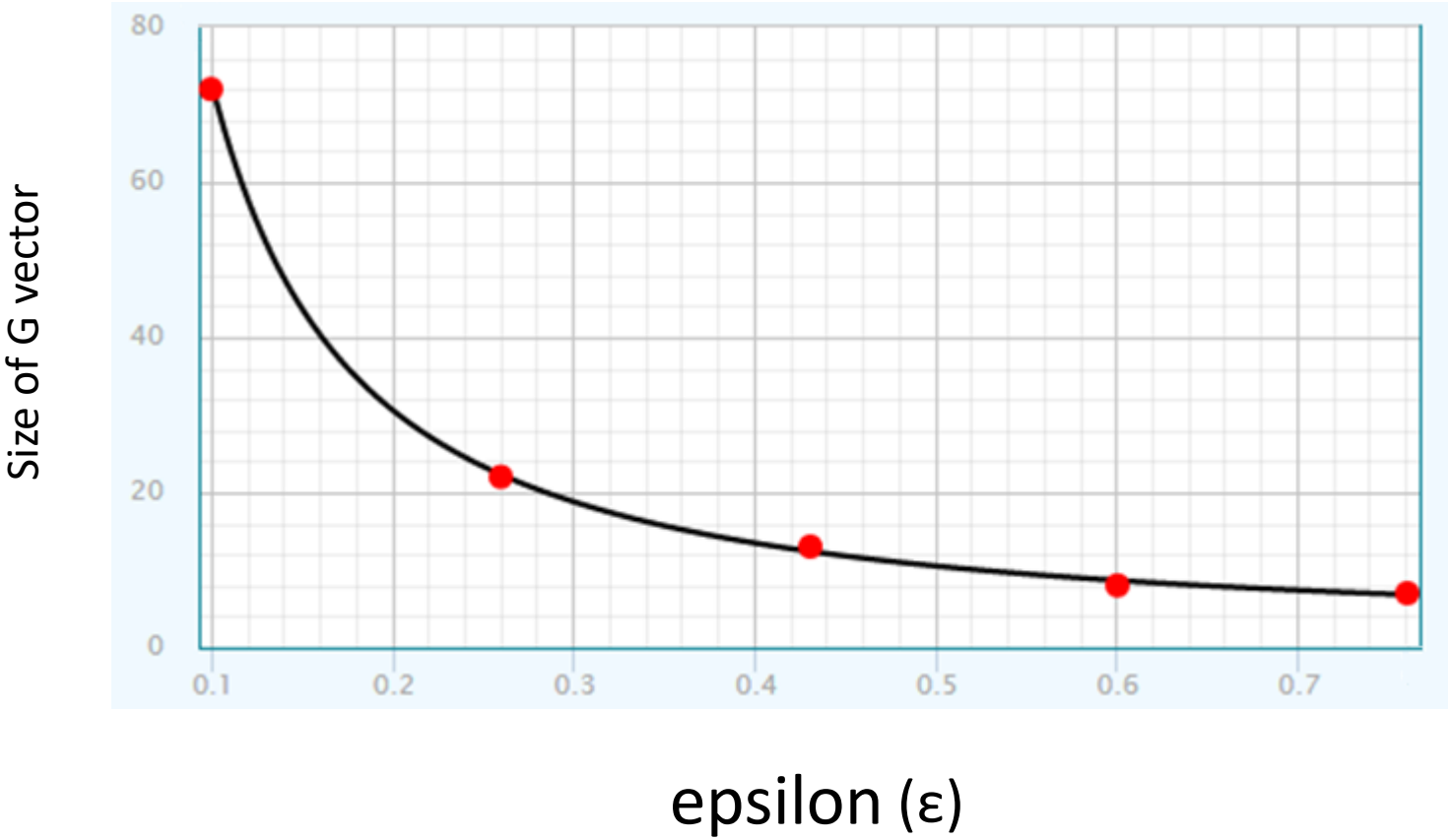


N = 15

ε = 0.9

$(1+\epsilon)\frac{n}{k}$ = 9.5

Size G Vector – Epsilon (ε)



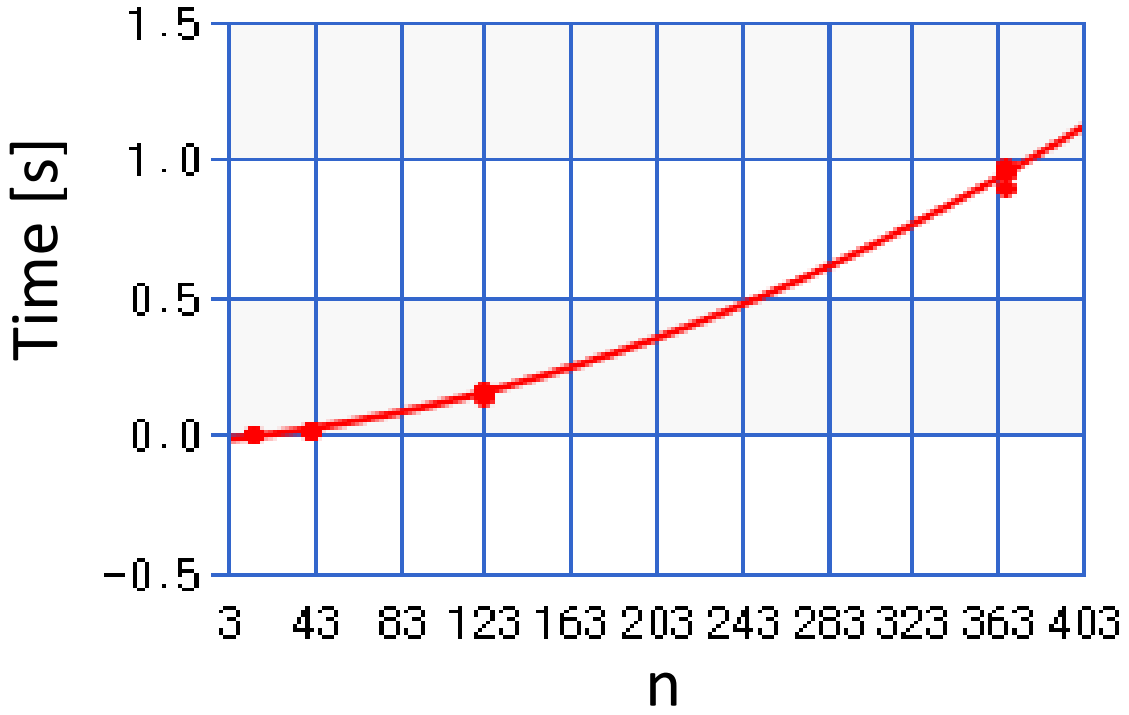
Theoretical size of G vector:

$$O(\frac{1}{\epsilon}\log(\frac{1}{\epsilon}))$$

The size is independent from both n and k, and depends only on epsilon (ε)

This a because the size of G vector is the number of (1+ε/2)-powers in the interval ((ε*n)/3k,(1+ ε)n k)

Time – Number of Nodes



K = n / 2, ε = 0.5

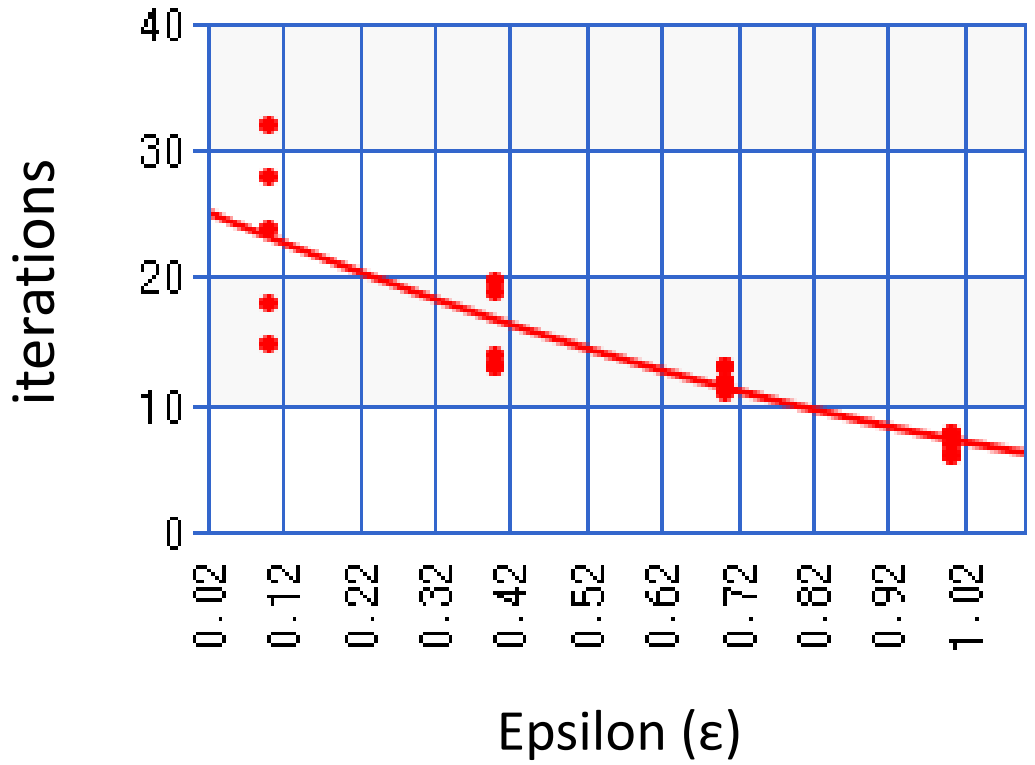
Weight of the edges = 1

O(partition) = n^2

Increase n means that a larger amount of sub trees is generated and as a consequence the overall time of execution increases

In fact independent vector G configurations are saved for each sub tree

Iterations - Epsilon



The number of iterations decreases as epsilon increases, moreover, also the variance reduces.

Increase epsilon means that the size of possible partition rise and as a consequence the total number of instances in the dynamic programming table increases. In addition, also the height of the trees rises

Test different Graphs: Results

We tested the graph with different type of graphs and an increasing amount of nodes. The time proposed is the average of the different tries of that configuration.

$k = 3, \varepsilon = 0.9$

Type of Graph	N = 15	N = 42	N = 123	N = 366
Mendes graph	T = 4.75 ms	T = 29.8 ms	T = 612 ms	12.1 s

Type of Graph	N = 50	N = 100	N = 150	N = 200
Gaussian random graph	T = 47.4 ms	T = 334 ms	T = 988 ms	1.81 s

Type of Graph	N = 50	N = 100	N = 150	N = 200
Random graph	T = 52,1 ms	T = 341 ms	T = 1.17 s	3.25 s

N = 300	N = 450	N = 600
T = 11.4 s	T = 38.7 s	T = 125.3 s

Test different Graphs: Analysis

As expected the overall time of execution is not exponential but polynomial to the number of initial nodes.

The algorithm perform particularly well when the ratio numberOfNodes/edges is high, especially because of the division section of the algorithm.

As expected the algorithm performs better with gaussian random graph than with simple random graph because the presence of natural partition helps the algorithm.

Conclusions

In the overall the complexity is the maximum between the two separated pieces of the algorithm; the division part that mainly depends on the number of nodes present in the graph and the dynamic programming part that mainly depends on how many different pieces are generated.

The algorithm has been implemented with non-directional graph but with edges of possibly different weight. Its solution has proven to be very effective in terms of computational time and it also suitable for a parallel implementation during the phase of partitions evaluation

References

http://www.math.cmu.edu/~kandreev/kpart.pdf

Balanced Graph Partitioning (Published paper)

Konstantin Andreev and Harald Racke (Authors)

https://en.wikipedia.org/wiki/Push%E2%80%93relabel_maximum_flow_algorithm

Push–relabel maximum flow algorithm