The Kernighan-Lin algorithm

Paolo Dolce

May 9, 2020

Contents

1 Introduction

The goal of these notes is to give a general description and an implementation of the well known Kernighan-Lin algorithm for partitioning a graph. The problem of graph partitioning can be summarized as follows:

Problem 1.1 (Graph partitioning)**.** An undirected graph $G = (V, E)$ is given together with two positive numbers n_1, n_2 such that $n = |V| = n_1 + n_2$. We have to find a partition of the set of nodes $V_1 \sqcup V_2 = V$ with the following properties:

- $|V_1| = n_1$ and $|V_2| = n_2$.
- The number of edges of G running between V_1 and V_2 is the least as possible.

Any solution V_1, V_2 (which is not unique) is called *an optimal partition of* G.

Remark 1.2. By a *partition of a graph* $G = (V, E)$ we always mean a partition of the set V.

Definition 1.3. Let $V_1 \sqcup V_2 = G$ be a partition of a graph G, then the *cutsize* of the partition is the number of edges of G running between V_1 and V_2. In symbols it is denoted with $\mathrm{CUT}[V_1, V_2]$.

Remark 1.4. In other words, the problem 1.1 consists in finding a partition of G (in two sets) with prescribed cardinalities and minimal cutsize.

A slight generalization of this problem has applications in parallel computing. Suppose indeed that our task is the elaboration of a big amount of data with a cluster composed of r linked computers. We need to to split up the data in r pieces, and then give each piece to a member of the cluster. Every computer performs the operation on his input, but in general the elements of the dataset are not independent from each other; it means that a certain computer of the cluster needs the results of another computer to carry out the computation. We can modelize the situation as a graph where the nodes are the elements of the dataset and where there is an edge between two nodes x and y if and only if y is necessary for processing x or viceversa. Now, how should we choose the assignation of data to the r computers? Due to hardware limitations, the most inefficient part of the total computation consists in communications between the members of the cluster (if r is large enough); therefore we have to choose the assignments in order to minimize the communications. But this is exactly the above problem of graph partitioning with the only difference in the number of sets which subdivide the graph. Obviously once we have solved the case $r = 2$, namely the problem 1.1, then we are able to handle the generic case: we first find an optimal partition V_1,

G_1 of G such that $|V_1| = \frac{n}{r}$ and $|G_1| = (r-1)\frac{n}{r}$, then we repeat the operation on G_1 to obtain optimal partition (for G_1) V_2, G_2 with $|V_1| = \frac{n}{r}$ and $|G_2| = (r-2)\frac{n}{r}$ and so on. By continuing in this fashion we arrive at a partition $V_1, \ldots, V_n$ of G with the required properties.
Unfortunately the problem 1.1 is **NP**-hard (cf. [GJ79]) so the only way to find "a solution" in a reasonable amount of time is by using heuristic algorithms. These kind of algorithms run in polynomial time on every input, but as the word "heuristic" suggests, they only return an acceptable approximation of an exact solution. In particular, a heuristic algorithm that solves the partitioning problem, won't find an optimal partition, but a partition with "a small enough" cutsize (nevertheless with a polynomial running time). The Kernighan-Lin algorithm, which was presented for the first time in [KL70], is maybe the simplest and most known heuristic algorithm for graph partitioning.

2 Informal presentation

The most straightforward approach to a heuristic algorithm for solving the graph partitioning could consist in generating a certain number random partitions of G and then taking the best one. This is a very simple and fast probabilistic algorithm, but the severe drawback is that the probability to find a good approximation of an optimal partition is very small. For example [KL70] says that for a graph with only 32 nodes there are typically from 3 to 5 optimal partitions of size 16; on the other hand the total number of partitions is $\frac{1}{2}\binom{32}{16}$, so the probability to solve exactly the problem in this simple case with a single random choice is less than 10^{-7}!
Nevertheless random partitions are a good starting point for a reasonable algorithm: consider a random partition V_1, V_2 of G, can we obtain an optimal partition from it? The answer is clearly yes, in fact there are two sets of the same size $X \subseteq V_1$ and $Y \subseteq V_2$ such that their exchanging (Y goes in V_1 and X goes in V_2) gives an optimal partition. The Kernighan-Lin algorithm consists in approximating such X and Y. We describe in a schematic way how the algortihm performs this task:

Input. A graph G with the sizes n_1 and n_2.

Random-step. Generate a random partition $V_1 \sqcup V_2$ of G such that $|V_1| = n_1$ and $|V_2| = n_2$.

switch-step. Can be divided in 3 substeps:

a. Find *one* couple $(i, j) \in V_1 \times V_2$ such that

$$\mathrm{CUT}\left[(V_1 \setminus \{i\}) \cup \{j\}, (V_2 \setminus \{j\}) \cup \{i\}\right]$$

is minimal and then define $V_1^{(1)} := (V_1 \setminus \{i\}) \cup \{j\}$, $V_2^{(1)} := (V_2 \setminus \{j\}) \cup \{i\}$. Practically here we exchange two vertexes lying in different sets of the partition in order to increase the less as possible the cutsize.

b. Repeat the point *a.* to obtain $V_1^{(2)}$ and $V_2^{(2)}$ but this time choose the couples in $V_1^{(1)} \times V_2^{(1)} \setminus \{(j, i)\}$. Continue the switchings in this way until there are switchable nodes, with the rule that *during all the switch-step* a node can be moved only *once* from a set to another. At the end of this substep we have done exactly $\min(n_1, n_2)$ swappings.

c. Among all partitions $V_1^{(1)} \sqcup V_2^{(1)}, V_1^{(2)} \sqcup V_2^{(2)}, \ldots, V_1^{(\min(n_1,n_2))} \sqcup V_2^{(\min(n_1,n_2))}$ choose one $V_1^{(k)} \sqcup V_2^{(k)}$ with minimal cutsize.

Recursive-step. If $\mathrm{CUT}\left[V_1^{(k)}, V_2^{(k)}\right] < \mathrm{CUT}\left[V_1, V_2\right]$ then re-define $V_1 := V_1^{(k)}$, $V_2 := V_2^{(k)}$ and repeat the switch-step on the new partition $V_1 \sqcup V_2$. Else return the old partition given at the beginning of the switch-step and terminate the algorithm.
Note that every *new* partition $V_1 \sqcup V_2$, is obtained by exchanging two subsets $\widetilde{X}$ and $\widetilde{Y}$ respectively of the *old* V_1 and V_2. These $\widetilde{X}$ and $\widetilde{Y}$ are precisely the approximations of X and Y we were searching for. The process continues until we have improvements of the cutsize.

The description of the Kernighan-Lin algortihm is complete, but how many times does it find an optimal partition? We can answer with considerations based on experiments and assuming we know apriori the exact solution: let, for example, G be a random graph with n nodes and of density $\rho \approx \frac{1}{2}$; moreover denote with $p(n)$ the probability that a Kernighan-Lin algorithm finds an optimal partition for G with only one recursive step. Under these hypotheses [KL70] asserts that $p(n) \approx 2^{-\frac{n}{30}}$.

3 Asymptotic analysis

Suppose that the input graph G with n nodes and m edges is stored through its adjacency list, however let's denote with A the adjacency matrix of G. Moreover let $V_1 \sqcup V_2$ be a partition of G, then for $i \in V_1$ we define the two relative degrees:

$$k_i^{\text{other}} = \text{number of nodes in } V_2 \text{ which are adjacent to } i$$

$$k_i^{\text{same}} = \text{number of nodes in } V_1 \text{ which are adjacent to } i$$

The same definitions hold for a vertex $j \in V_2$. At this point the cutsize of the partition is given by

$$\text{CUT}[V_1, V_2] = \sum_{i \in V_1} k_i^{\text{other}} = \sum_{j \in V_2} k_j^{\text{other}} \tag{1}$$

When we exchange two nodes $i \in V_1$ and $j \in V_2$ the increment (or decrement) of the original cutsize can be obtained by the equation

$$\Delta = k_i^{\text{same}} - k_i^{\text{other}} + k_j^{\text{same}} - k_j^{\text{other}} + 2A_{ij} \tag{2}$$

Note that for the calculation of Δ we don't really need the adjacency matrix of G, but we have only to know if i and j are adjacent and this can be checked in the adjacency list. So the term $2A_{ij}$ in the equation (2) only simplifies the notation.
Now let's analyze the running time of the Kernighan-Lin algorithm when n_1 is of the same order of magnitude respect to n. Generating a random partition $V_1 \sqcup V_2$ has cost $O(n)$ and the calculation of the cutsize of this partition has cost $O(|V_1|\frac{m}{n})$ in fact we have to check all the neighborhoods of every vertex in V_1 to get the sum of the equation (1). Obviously with our assumptions $O(|V_1|\frac{m}{n}) = O(\frac{mn}{n}) = O(m)$. In the part *a.* of the switch-step we have to calculate $n_1 n_2$ times a cutsize, but given the initial cutsize of the random partition it is enough to get the cutsize increment with the equation (2) which has cost $O(\frac{m}{n})$ since we have to check only the neighborhoods of i and j. Therefore the total cost of the point *a.* is $O(n^2)O(\frac{m}{n}) = O(mn)$. The cost of *b.* and *c.* basically consists in repeating *a.* exactly $\min(|V_1|, |V_2|)$ times, so it is $O(n)$. Reassuming, at the end of the switch-step we have an asymptotic running time of $O(n) + O(m) + O(mn)O(n) = O(mn^2)$. It remains to analyze the recursive-step which possibly calls t times all the previous procedure, but it turns out experimentally that t is a small constant (for example $t < 10$ for $n \approx 1000$). The final cost of the Kernighan-Lin algorithm is $O(mn^2)$ and note that this running time is $O(n^3)$ if G is sparse and $O(n^4)$ if G is dense.

Remark 3.1. A cutsize is calculated explicitly only for the first random partition. All the others cutsizes involved in the algorithm are obtained through Δ.

The conclusion is that the Kernighan-Lin algorithm is not very efficient, but on the other hand it is of simple implementation as we will see in the next section.

4 Implementation

In this section we give an implementation of the Kernighan-Lin algorithm in Python 2.7 (Anaconda distribution). In particular we use the Python package `NetworX` to manage graphs.

Why Python (Anaconda distribution)? Python is an high-level dynamic object oriented programming language ideated by Guido Van Rossum (Benevolent Dictator for Life) at the beginning of '90s. It was designed to be highly extensible and portable, moreover its license is free. Python's development is carried out through the Python Enhancement Proposal (PEP) process. The PEP process is the primary mechanism for proposing new features, for collecting issues, and for documenting the design decisions. The PEPs are reviewed and commented upon by the very large Python community and by Van Rossum. One of the commonly cited greatest strenghts of Python is the very extensive standard library made of built-in modules compatible with all platforms. In addition to the standard library, there is a growing collection of several thousand components from individual programs and modules to packages and entire application development frameworks. For example the Anaconda distribution contains more than 200 of the most popular Python packages for science, math, engineering and data analysis. Maybe the package `NetworkX` is not the best choice, in terms of performances, to handle graphs (`Igraph` is faster); but on the other hand `NetworkX` is already supported in Anaconda, it is more intuitive and moreover contains many built in functions. Anyway it seems that the Kernighan-Lin algorithm is not implemented in `NetworkX`.

First of all we need two functions: one for calculating the cutsize and one for calculating the variation of the cutsize after a switching of two nodes (equations (1) and (2)). We have collected both the functions in a new Python module `gr_tools.py` :

```
import networkx as nx


# The function 'cutsize' receives as input:
# - A graph G.
# - Two lists V1, V2 which represent the two sets of the partition.
#
# It returns the cutsize of the partition.

def cutsize(G,V1,V2):
    cut=0  # the cutsize is initialized
    for i in V1:
        for j in V2:
            if G.has_edge(i,j): # the method 'G.has_edge(i,j)' returns
                                # 'True' if there is an edge between
                                # i and j, otherwise it returns 'False'.

                cut=cut+1
    return cut



# The function 'cutchange' receives as input:
# - A graph G.
# - Two lists V1, V2 which represent the two sets of the partition.
# - The two nodes i and j to be exchanged.
#
# It returns the cutsize increment if we effectively exchange the two nodes.


def cutchange(G,V1,V2,i,j):
    ki_o=0  # We initialize the relative degrees of i and j.
    ki_s=0
    kj_o=0
    kj_s=0
    for x in G[i]: # Here 'G[i]' is the set of all neighborhoods of i.
        if x in V1:
            ki_s=ki_s+1
        else:
            ki_o=ki_o+1

    for y in G[j]:
        if y in V2:
            kj_s=kj_s+1
        else:
            kj_o=kj_o+1
    if G.has_edge(i,j): # We apply the equation (2) without using the
                        # adjacency matrix.

       return ki_s-ki_o+kj_s-kj_o+2
    else:
       return ki_s-ki_o+kj_s-kj_o
```

At this point we construct other two auxiliary functions in the python module `Ker_Lin_tools.py`:

```
import networkx as nx
import gr_tools as gr
import sys


# Suppose that the switch-step has already exchanged k-1 times two nodes.
# The moved nodes are placed at the beginning of the two lists V1 and V2,
# and at this point only the nodes of V1[k:] and V2[k:] can be exchanged.
# Now the function 'Switch' finds the best way to perform the swappings.
#
# INPUT: the graph G, the partition V1,V2 (two lists) with the
#         cutsize and the integer k.
#
```

```
# OUTPUT: The two lists V1 and V2 where we two nodes are exchanged
#         (these nodes now are in the head of both lists) and
#         the new cutsize.


def Switch(G,V1,V2,cut,k):
    change=sys.maxint  # 'change' is the smallest
                       #  variation of cutsize.
    for i in V1[k:]:
        for j in V2[k:]:
            temp_change=gr.cutchange(G,V1,V2,i,j) # Here we calculate
                                                  # the variation of
                                                  # cutsize of every
                                                  # possible couple
                                                  # of nodes.

            if temp_change<change: # We check if the swapping
                                   # "is convenient"

               I=i    # The nodes which give the smallest variation
                      # of cutsize are memorized as I and J.
               J=j
               change=temp_change

    del V1[V1.index(I)]  # I and J are insercted respectively
                         # at the beginning of V1 and V2.
    V1.insert(0,J)
    del V2[V2.index(J)]
    V2.insert(0,I)
    cut=cut+change
    return V1, V2, cut


# The function 'oneiter' perform  an iteration of the recursive-step,
# namely it executes all the possible swappings and then returns the
# best configuration.

# INPUT: The graph G, the current partition V1, V2 (two lists) with
#         its cutsize

# OUTPUT: The best partition after the swappings V1fin, V2fin with its
#         cutsize.


def oneiter(G,V1,V2,cut):
    tempcut=sys.maxint # This is the smallest cutsize.

    for k in range(min(len(V1),len(V2))): # This loop tries
                                          # all possible swappings
                                          # by recalling the
                                          # 'Switch' function.
        V1, V2, cut=Switch(G,V1,V2,cut,k)
        if cut<tempcut:    # Every time we meet a better
                           # configuration than the previous
                           # ones we memorize it. In this way,
                           # at the end we obtain the
                           # best configuration.
           tempcut=cut
           V1fin=V1[:]
           V2fin=V2[:]

    return V1fin, V2fin, tempcut,
```

Finally we get the complete implementation of the Kernighan-Lin algorithm:

```
import random as ran
import Ker_Lin_tools as kl
import gr_tools as gr
import networkx as nx
import sys
import timeit


#INPUT: The graph G and the size of one set of the partition.
```

```
#
#OUTPUT: The approximate solution V1, V2 with the relative cutsize,
#        the number of iterations of the recursive-steps,
#        and the running time in seconds.


def KL(G, n1):
    start_time = timeit.default_timer() # Cronometer starts.
    n=len(G)
    n2=n-n1
    V=nx.nodes(G)
    ran.shuffle(V) # Generation of the random partition: first we
                   # shuffle the set of nodes and the we pick the
                   # first n1 elements.
    V1=V[:n1]
    V2=V[-n2:]
    s=[V1,V2,gr.cutsize(G,V1,V2)]
    h=0    # This variable counts the iterations of the recursive-step.

    best=sys.maxint  # It is the cutsize of the partition returned
                     # by the algorithm.

    while s[2]<best:  # We check if at each iteration of the recursive-step
                      # the cutsize decreases.

          best=s[2]
          V1end=V1[:]
          V2end=V2[:]
          s=kl.oneiter(G,V1,V2,best)
          V1=s[0]
          V2=s[1]
          h=h+1
    elapsed_time = timeit.default_timer() - start_time # Cronometer stops.
    return(V1end, V2end, best, h, elapsed_time)
```

5 Tests

In section 3 we have shown that when the graph G is sparse the Kernighan-Lin algorithm has asymptotic running time of $O(n^3)$ and that when G is dense the asymptotic running time is $O(n^4)$. Now we report the results of some tests confirming this different behavior. We have fixed two densities $\rho_1 \approx 0.1$, $\rho_2 \approx 0.9$ and then we have generated (thanks to the `NetworkX` package) random graphs of $10k$ nodes for $k = 1, 2, \ldots, 10$ with such densities[1]:

nodes	edges
10	4
20	19
30	43
40	78
50	122
60	177
70	241
80	316
90	400
100	495

Table 1: Graphs with density $\rho_1 \approx 0.1$

nodes	edges
10	40
20	171
30	391
40	702
50	1102
60	1593
70	2173
80	2844
90	3604
100	4445

Table 2: Graphs with density $\rho_2 \approx 0.9$

At this point we have executed our Python implementation of the Kernighan-Lin algorithm on graphs of both tables 1 and 2; but since the algorithm has a random nature, we have run it five times on each graph. The following tables contain the resulting running time in seconds (approximated at the fourth decimal digit):

[1]Remember that the density of a graph is given by $\rho = \dfrac{2m}{n(n-1)}$ where n is the number of nodes and m the number of edges.

nodes	t_1	t_2	t_3	t_4	t_5
10	0.0015	0.0014	0.0014	0.0016	0.0007
20	0.0096	0.0090	0.0140	0.0055	0.0085
30	0.0436	0.0259	0.0304	0.0177	0.0270
40	0.1200	0.0506	0.0802	0.0502	0.0799
50	0.2968	0.3279	0.2555	0.1327	0.1982
60	0.3000	0.7639	0.4205	0.4089	0.5890
70	0.8690	0.8609	0.8592	1.4745	0.5812
80	1.6532	1.5593	2.6614	1.5615	1.0342
90	3.8223	3.8073	4.5100	3.6861	4.6930
100	6.0111	4.5487	7.4706	7.4331	5.9562

Table 3: Running times for graphs with density $\rho_1 \approx 0.1$

nodes	t_1	t_2	t_3	t_4	t_5
10	0.0020	0.0020	0.0019	0.0019	0.0019
20	0.0342	0.0202	0.0147	0.0138	0.0138
30	0.1517	0.0902	0.0912	0.0836	0.1245
40	0.3449	0.3177	0.5378	0.4803	0.3367
50	1.4182	1.3640	1.3590	1.4682	1.8576
60	3.3455	4.3034	3.2409	4.3101	3.1775
70	7.0367	6.9176	6.9531	9.2152	9.1629
80	21.9624	13.1523	17.6205	13.1435	13.1344
90	22.8763	31.2235	30.7370	30.6419	23.1074
100	39.8967	51.5302	38.7174	38.9893	38.5840

Table 4: Running times for graphs with density $\rho_2 \approx 0.9$

Now we plot the data:

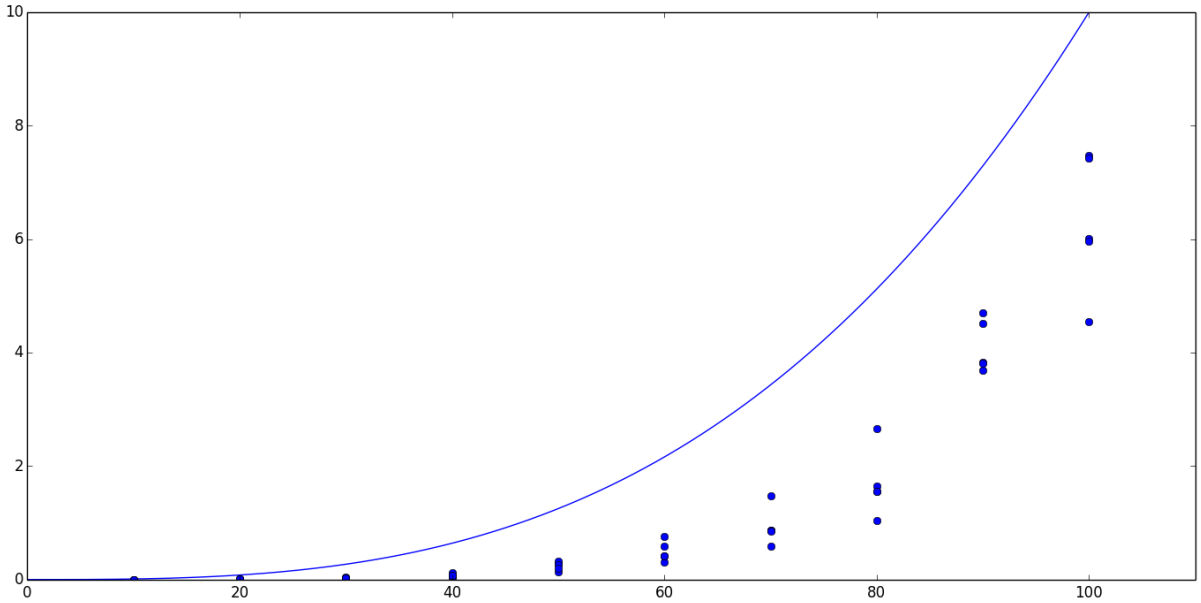

Figure 1: The x-axis contains the number of nodes, the y-axis contains the running times for graphs with density $\rho_1 \approx 0.1$. We have also plotted the function $f(x) = 10^{-5}x^3$ to give an idea of the growth rate.

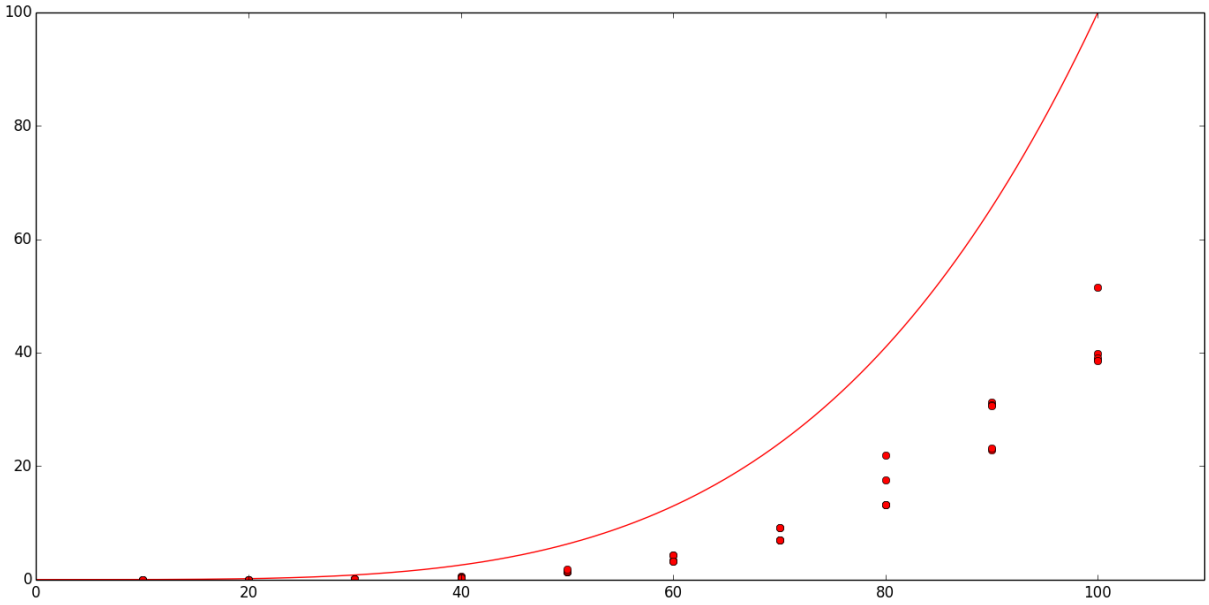

Figure 2: The x-axis contains the number of nodes, the y-axis contains the running times for graphs with density $\rho_1 \approx 0.9$. We have also plotted the function $f(x) = 10^{-6}x^4$ to give an idea of the growth rate.

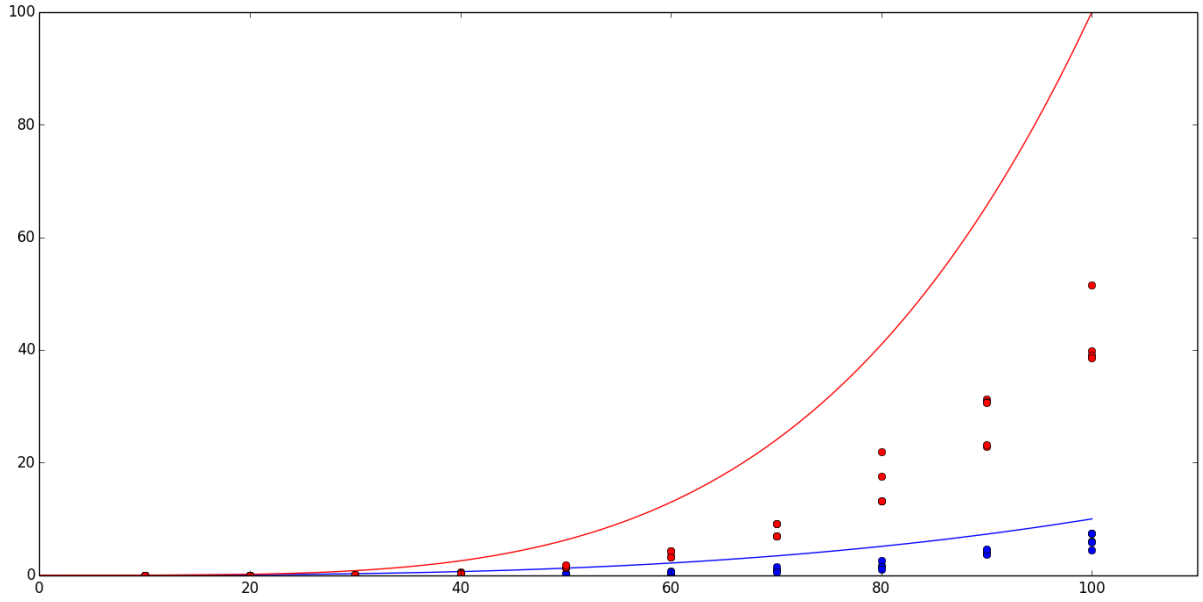

Figure 3: The figures 1 and 2 are merged. Note the two different growth rates as expected from the theory.

References

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, 1979.

[KL70] B.W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell Systems Technical Journal*, 49(2), 1970.

[New10] Mark Newman. *Networks: An Introduction*. Oxford University Press, 1 edition, 2010.