InfoFlow Maths

Felix Fung

February 7, 2019

Abstract

This document contains the relevant discrete maths and algorithmic considerations involved in the InfoMap and its parallelized algorithm, InfoFlow. We first provide a very brief overview of the InfoMap algorithm and relevant maths, then we derive discrete maths for the InfoMap algorithm specifically useful in a Spark implementation. We then expand the maths for a greedy algorithm that allows for improved performance and discuss the algorithm, which we call InfoFlow.

Contents

1 Fundamentals **2**
- 1.1 Nodes . . . 2
- 1.2 Modules . . . 2

2 Simplifying calculations **2**
- 2.1 Calculating merging quantities . . . 3
- 2.2 Calculating code length reduction . . . 5

3 InfoMap Algorithm **5**
- 3.1 Initial Condition . . . 5
- 3.2 Loop . . . 6
- 3.3 Algorithm . . . 6

4 Merging multiple modules at once **7**
- 4.1 Mathematics . . . 8
- 4.2 Algorithm . . . 9
- 4.3 Performance Improvement . . . 11

1 Fundamentals

These are the fundamental maths found in the original paper.

1.1 Nodes

Each node is indexed, with the index denoted by greek alphabets α, β or γ. Each node α is associated with an ergodic frequency p_α.

In between nodes there may be a directed edge $\omega_{\alpha\beta}$ from node α to node β. The directed edge weights are normalized with respect to the outgoing node, so that

$$\sum_\alpha \omega_{\alpha\beta} = 1$$

The nodes are unchanged for all partitioning schemes.

1.2 Modules

The nodes are partitioned into modules. Each module is indexed with Latin alphabets i, j, or k.

Each module has ergodic frequency:

$$p_i = \sum_{\alpha \in i} p_\alpha \tag{1}$$

and probability of exiting the module is:

$$q_i = \tau \frac{n - n_i}{n - 1} p_i + (1 - \tau) \sum_{\alpha \in i} \sum_{\beta \notin i} p_\alpha \omega_{\alpha\beta} \tag{2}$$

with these, we try to minimize the code length:

$$L = \text{plogp}\left(\sum_i q_i\right) - 2\sum_i \text{plogp}\left(q_i\right) - \sum_\alpha \text{plogp}(p_\alpha) + \sum_i \text{plogp}\left(p_i + q_i\right) \tag{3}$$

where

$$\text{plogp}(x) = x \log_2 x \tag{4}$$

2 Simplifying calculations

We develop maths to reduce the computational complexity for merging calculations. Specifically, we find recursive relations, so that when we merge modules j and k into i, we calculate the properties of i from those of j and k.

2.1 Calculating merging quantities

We can rewrite Eq. (2) as

$$q_i = \tau \frac{n - n_i}{n - 1} p_i + (1 - \tau) w_i \tag{5}$$

with

$$w_i = \sum_{\alpha \in i} \sum_{\beta \notin i} p_\alpha \omega_{\alpha\beta} \tag{6}$$

being the exit probability without teleportation.

We can define a similar quantity, the transition probability without teleportation from module j to module k:

$$w_{jk} = \sum_{\alpha \in j} \sum_{\beta \in k} p_\alpha \omega_{\alpha\beta} \tag{7}$$

Now, if we merge modules j and k into into a new module with index i, the exit probability would be follow Eq. (5) with

$$n_i = n_j + n_k \tag{8}$$

$$p_i = p_j + p_k \tag{9}$$

and the exit probability without teleportation can be calculated via:

$$\begin{aligned} w_i &= \sum_{\alpha \in i} \sum_{\beta \notin i} p_\alpha \omega_{\alpha\beta} && (10) \\ &= \sum_{\substack{\alpha \in j \\ \text{or } \alpha \in k}} \sum_{\substack{\beta \notin j \\ \text{and} \beta \notin k}} p_\alpha \omega_{\alpha\beta} && (11) \\ &= \sum_{\alpha \in j} \sum_{\substack{\beta \notin j \\ \text{and} \beta \notin k}} p_\alpha \omega_{\alpha\beta} + \sum_{\alpha \in k} \sum_{\substack{\beta \notin j \\ \text{and} \beta \notin k}} p_\alpha \omega_{\alpha\beta} && (12) \end{aligned}$$

since we are looking at the exit probability of a module, there are no self connections within modules, so that the specification of $p_\alpha w_{\alpha\beta}$ given $\alpha \in i$, $\beta \notin i$ is redundant. Then we have

$$w_i = \sum_{\alpha \in j} \sum_{\beta \notin k} p_\alpha \omega_{\alpha\beta} + \sum_{\alpha \in k} \sum_{\beta \notin j} p_\alpha \omega_{\alpha\beta} \tag{13}$$

which conforms with intuition, that the exit probability without teleportation of the new module is equal to the exit probability of all nodes without counting for the connections from j to k, or from k to j.

We can further simplify the maths by expanding the non-inclusive set specification:

$$w_i = \sum_{\alpha\in j}\left[\sum_{\beta} p_\alpha \omega_{\alpha\beta} - \sum_{\beta\in k} p_\alpha \omega_{\alpha\beta}\right] + \sum_{\alpha\in k}\left[\sum_{\beta} p_\alpha \omega_{\alpha\beta} - \sum_{\beta\in j} p_\alpha \omega_{\alpha\beta}\right] \tag{14}$$

Expanding gives:

$$w_i = \sum_{\alpha\in j}\sum_{\beta} p_\alpha \omega_{\alpha\beta} - \sum_{\alpha\in j}\sum_{\beta\in k} p_\alpha \omega_{\alpha\beta} + \sum_{\alpha\in k}\sum_{\beta} p_\alpha \omega_{\alpha\beta} - \sum_{\alpha\in k}\sum_{\beta\in j} p_\alpha \omega_{\alpha\beta} \tag{15}$$

which by definition is

$$w_i = w_j - w_{jk} + w_k - w_{kj} \tag{16}$$

This allows economical calculations.

We can do similar for w_{il}, if we merged modules j and k into i, and l is some other module:

$$\begin{aligned}
w_{il} &= \sum_{\alpha\in i}\sum_{\beta\in l} p_\alpha \omega_{\alpha\beta} && (17)\\
&= \sum_{\substack{\alpha\in j\\ \text{or } \alpha\in k}}\sum_{\beta\in l} p_\alpha \omega_{\alpha\beta} && (18)\\
&= \sum_{\alpha\in j}\sum_{\beta\in l} p_\alpha \omega_{\alpha\beta} + \sum_{\alpha\in k}\sum_{\beta\in l} p_\alpha \omega_{\alpha\beta} && (19)\\
&= w_{jl} + w_{kl} && (20)
\end{aligned}$$

and similarly for w_{li}:

$$w_{li} = w_{lj} + w_{lk} \tag{21}$$

We can simplify further. The directionality of the connections are not needed, since w_{ij} and w_{ji} always appear together in Eq. (16). Then, we can define

$$w_{i\leftrightharpoons l} = w_{il} + w_{li} \tag{22}$$

and we can verify that Eq. (20) and Eq. (21) combine to give

$$\begin{aligned}
w_{i\leftrightharpoons l} &= w_{il} + w_{li} && (23)\\
&= w_{jl} + w_{kl} + w_{lj} + w_{lk} && (24)\\
&= w_{jl} + w_{lj} + w_{kl} + w_{lk} && (25)\\
&= w_{j\leftrightharpoons l} + w_{k\leftrightharpoons l} && (26)
\end{aligned}$$

and this quantity is applied via

$$q_i = \tau \frac{n - n_i}{n - 1} p_i + (1 - \tau) w_i \tag{27}$$

$$w_i = w_j + w_k - w_{j \leftrightharpoons l} \tag{28}$$

The calculations above has a key, central message: that **for the purpose of community detection, we can forget about the actual nodal properties; after each merge, we only need to keep track of a module/community**.

2.2 Calculating code length reduction

Given an initial code length according to Eq. (3), further iterations in code length calculation can calculated via dynamic programming. Whenever we merge two modules j and k into i, with new module frequency p_i and q_i, the change in code length is:

$$\begin{aligned}\Delta L_i = &\ \mathrm{plogp}\left[q_i - q_j - q_k + \sum_i q_i\right] - \mathrm{plogp}\left[\sum_i q_i\right] \\ &- 2\mathrm{plogp}(q_i) + 2\mathrm{plogp}(q_j) + 2\mathrm{plogp}(q_k) \\ &+ \mathrm{plogp}(p_i + q_i) - \mathrm{plogp}(p_j + q_j) - \mathrm{plogp}(p_k + q_k) \end{aligned} \tag{29}$$

so that if we keep track of $\sum_i q_i$, we can calculate ΔL quickly by plugging in p_i, p_j, p_k, q_i, q_j, q_k.

3 InfoMap Algorithm

The algorithm consists of two stages, the initial condition and the loop:

3.1 Initial Condition

Each node is its own module, so that we have:

$$\begin{aligned} n_i &= 1 \\ p_i &= p_\alpha \\ w_i &= p_\alpha \sum_{\beta \neq \alpha} \omega_{\alpha\beta} \\ q_i &= \tau \frac{n - n_i}{n - 1} p_i + (1 - \tau) w_i \\ w_{i \leftrightharpoons j} &= \omega_{ij} + \omega_{ji}, \quad \forall \omega_{ij} \text{ and } \omega_{ji} \end{aligned}$$

and ΔL is calculated for all possible merging pairs according to Eq. (29).

3.2 Loop

Find the merge pairs that would minimize the code length; if the code length cannot be reduced then terminate the loop. Otherwise, merge the pair to form a module with the following quantities, so that if we merge modules j and k into i, then: (these equations are presented in previous sections, but now repeated for ease of reference)

$$\begin{aligned}
n_i &= n_j + n_k \\
p_i &= p_j + p_k \\
w_i &= w_j + w_k - w_{j\leftrightharpoons k} \\
q_i &= \tau \frac{n - n_i}{n - 1} p_i + (1 - \tau) w_i \\
w_{i\leftrightharpoons l} &= w_{j\leftrightharpoons l} + w_{k\leftrightharpoons l}, \quad \forall l \neq i
\end{aligned}$$

and

$$\begin{aligned}
\Delta L_{i\leftrightharpoons l} = \text{plogp} &\left[q_{i\leftrightharpoons l} - q_i - q_l + \sum_i q_i \right] - \text{plogp} \left[\sum_k q_k \right] \\
&- 2\text{plogp}(q_{i\leftrightharpoons l}) + 2\text{plogp}(q_i) + 2\text{plogp}(q_l) \\
&+ \text{plogp}(p_i + q_{i\leftrightharpoons l}) - \text{plogp}(p_i + q_i) - \text{plogp}(p_l + q_l) \qquad (30)
\end{aligned}$$

is recalculated for all merging pairs that involve module i, i.e., for each $w_{i\leftrightharpoons l}$. The sum $\sum_i q_i$ is iterated in each loop by adding $q_i - q_j - q_k$.

3.3 Algorithm

Given the above math, the pseudocode is:

Initiation:

- Calculate the RDD of edges in the format ((vertex1,vertex2),weight). Vertex1 and vertex2 are arranged such that vertex1 is always smaller than vertex2. If the directed edge vertex1 to vertex2 exists and the directed edge vertex2 to vertex1 exists, then the weight of the RDD entry is the sum of the two directed edge weights. Otherwise, the weight of the RDD entry is the weight of the existing edge.
- Construct a table (i.e. row and column), where each row is an undirected edge between modules in the graph. Each row is of format ((vertex1,vertex2), (n1, n2, p1, p2, w1, w2, w12, q1, q2, ΔL)). The quantities n, p, w, q, are properties of the two modules, n is the nodal size of the module, p is the ergodic frequency of the module, w is the

exit probability of a module without teleportation, q is the exit probability of a module with teleportation, ΔL is the change in code length if the two modules are merged. Vertex1 and vertex2 are arranged such that vertex1 is always smaller than vertex2.

Loop:

1. Pick the row that has the smallest ΔL. If it is non-negative, terminate the algorithm.
2. Calculate the newly merged module size, ergodic frequency, exit probabilities.
3. Calculate the new RDD of edges by deleting the edges between the merging modules, and then aggregate all edges associated with module 2 to those in module 1.
4. Update the table by aggregating all rows associated with module 2 to those in module 1. Join the table with the RDD of edges. Since the RDD of edges contain $w_{1\rightleftharpoons k}$, we can now calculate ΔL and put it in the table for all rows associated with module 1.
5. Repeat from Step 1.

There are $O(e)$ merges, e being the number of edges in the graph.

4 Merging multiple modules at once

In the previous sections, we have developed the discrete mathematics and algorithm that performs community detection with $O(e)$ loops, based on the key mathematical finding that we only need to remember modular properties, not nodal ones.

The algorithm above does not take advantage of the parallel processing capabilities of Spark. One obvious improvement possibility is to perform multiple merges per loop. However, algorithms so far focus on merging two modules on each iteration, which is not compatible with the idea of performing multiple merges, unless we can make sure no module is involved with more than one merge at once.

Here, rather than focusing on making sure that no module is involved with more than one merge at once, we can explore the idea of merging multiple modules at once. Thus, we can perform parallel merges in the same loop iteration, where possibly *all* modules are involved in some merge.

4.1 Mathematics

Here we develop the mathematics to keep track of merging multiple modules at once.

We consider multiple modules $\mathcal{M}_i$ merging into a module $\mathcal{M}$. Another way to express this equivalently is to say that a module $\mathcal{M}$ is partitioned into i non-overlapping subsets:

$$\mathcal{M} = \sum_i \mathcal{M}_i \tag{31}$$

Then we can expand the nodal sum over module $\mathcal{M}$ into the sum over all nodes in all submodules $\mathcal{M}_i$, the exit probability of the merged module $\mathcal{M}$ becomes:

$$\begin{aligned}
w_{\mathcal{M}} &= \sum_{\mathcal{M}_i} \sum_{\alpha \in \mathcal{M}_i} \sum_{\beta \notin \mathcal{M}} p_\alpha w_{\alpha\beta} && (32)\\
&= \sum_{\mathcal{M}_i} \sum_{\alpha \in \mathcal{M}_i} \left[\sum_{\beta} p_\alpha w_{\alpha\beta} - \sum_{\beta \in \mathcal{M}} p_\alpha w_{\alpha\beta} \right] && (33)\\
&= \sum_{\mathcal{M}_i} \sum_{\alpha \in \mathcal{M}_i} \sum_{\beta} p_\alpha w_{\alpha\beta} - \sum_{\mathcal{M}_i} \sum_{\alpha \in \mathcal{M}_i} \sum_{\beta \in \mathcal{M}} p_\alpha w_{\alpha\beta} && (34)\\
&= \sum_{\mathcal{M}_i} \sum_{\alpha \in \mathcal{M}_i} \sum_{\beta} p_\alpha w_{\alpha\beta} - \sum_{\mathcal{M}_i} \sum_{\alpha \in \mathcal{M}_i} \sum_{\mathcal{M}_j} \sum_{\beta \in \mathcal{M}_j} p_\alpha w_{\alpha\beta} && (35)
\end{aligned}$$

where we expand the second term with respect to the $\mathcal{M}_j$'s to give

$$\begin{aligned}
w_{\mathcal{M}} = & \sum_{\mathcal{M}_i} \sum_{\alpha \in \mathcal{M}_i} \sum_{\beta} p_\alpha w_{\alpha\beta} && (36)\\
& - \sum_{\mathcal{M}_i} \sum_{\alpha \in \mathcal{M}_i} \sum_{\mathcal{M}_j \neq \mathcal{M}_i} \sum_{\beta \in \mathcal{M}_j} p_\alpha w_{\alpha\beta} - \sum_{\mathcal{M}_i} \sum_{\alpha \in \mathcal{M}_i} \sum_{\beta \in \mathcal{M}_i} p_\alpha w_{\alpha\beta} && (37)
\end{aligned}$$

Combining the first and third terms,

$$w_{\mathcal{M}} = \sum_{\mathcal{M}_i} \sum_{\alpha \in \mathcal{M}_i} \sum_{\beta \notin \mathcal{M}_i} p_\alpha w_{\alpha\beta} - \sum_{\mathcal{M}_i} \sum_{\alpha \in \mathcal{M}_i} \sum_{\mathcal{M}_j \neq \mathcal{M}_i} \sum_{\beta \in \mathcal{M}_j} p_\alpha w_{\alpha\beta} \tag{38}$$

which we can recognize as

$$\begin{aligned}
w_{\mathcal{M}} &= \sum_{\mathcal{M}_i} w_{\mathcal{M}_i} - \sum_{\mathcal{M}_i} \sum_{\mathcal{M}_j \neq \mathcal{M}_i} w_{\mathcal{M}_i \mathcal{M}_j} && (39)\\
w_{\mathcal{M}_i} &= \sum_{\alpha \in \mathcal{M}_i} \sum_{\beta \notin \mathcal{M}_i} p_\alpha w_{\alpha\beta} && (40)\\
w_{\mathcal{M}_i \mathcal{M}_j} &= \sum_{\alpha \in \mathcal{M}_i} \sum_{\beta \in \mathcal{M}_j} p_\alpha w_{\alpha\beta} && (41)
\end{aligned}$$

where we can immediately see that Eq. (39) is a linear generalization of Eq. (28), while Eq. (40) and Eq. (41) are identical to previous definitions, and may be calculated iteratively as the previous algorithm. We can calculate $w_{\mathcal{M}_i\mathcal{M}_j}$ by expanding on the partitioning:

$$w_{\mathcal{M}_i\mathcal{M}_j} = \sum_{\mathcal{M}_k\in\mathcal{M}_i}\sum_{\alpha\in\mathcal{M}_k}\sum_{\mathcal{M}_{k'}\in\mathcal{M}_j}\sum_{\beta\in\mathcal{M}_j} p_\alpha w_{\alpha\beta} \tag{42}$$

$$= \sum_{\mathcal{M}_k\in\mathcal{M}_i}\sum_{\mathcal{M}_{k'}\in\mathcal{M}_j} w_{\mathcal{M}_k\mathcal{M}_{k'}} \tag{43}$$

so that when we merge a number of modules together, we can calculate its connections with other modules by aggregating the existing modular connections. This is directly analogous to Eq. (20).

Thus, the mathematical properties of merging multiple modules into one are identical to that of merging two modules. This is key to developing multi-merge algorithm.

4.2 Algorithm

Initiation is similar to the infomap algorithm, so that we have an RDD of modular properties in the format (index,n,p,w,q), and edge properties ((vertex1,vertex2),summed connection weight), ((vertex1,vertex2),deltaL).

Loop:

1. For each module, seek to merge with another module that would offer the greatest reduction in code length. If no such merge exists, the module does not seek to merge. Then, we have $O(e)$ edges.

2. For each of these edges, label it to the module index to be merged to. This has $O(k)$ complexity, if k edges have linked vertices. The precise algorithm is described below.

3. Recalculate modular and edge property values via aggregations:

$$\begin{aligned}
n_i &= \sum_{k\to i} n_k \\
p_i &= \sum_{k\to i} p_k \\
w_i &= \sum_{k\to i} w_k - \sum_{k\leftrightharpoons k'\to i} w_{k\leftrightharpoons k'} \\
q_i &= \tau \frac{n-n_i}{n-1} p_i + (1-\tau) w_i \\
w_{i\leftrightharpoons j} &= \sum_{k\to i} \sum_{k'\to j} w_{k\leftrightharpoons k'} \\
L &= \operatorname{plogp}\left(\sum_i q_i\right) - 2\sum_i \operatorname{plogp}(q_i) \\
&\quad - \sum_\alpha \operatorname{plogp}(p_\alpha) + \sum_i \operatorname{plogp}(p_i + q_i) \\
\Delta L_{i\leftrightharpoons j} &= \operatorname{plogp}\left[q_{i\leftrightharpoons j} - q_i - q_j + \sum_k q_k\right] - \operatorname{plogp}\left[\sum_k q_k\right] \\
&\quad - 2\operatorname{plogp}(q_{i\leftrightharpoons j}) + 2\operatorname{plogp}(q_i) + 2\operatorname{plogp}(q_j) \\
&\quad + \operatorname{plogp}(p_{i\leftrightharpoons j} + q_{i\leftrightharpoons j}) - \operatorname{plogp}(p_i + q_i) - \operatorname{plogp}(p_j + q_j)
\end{aligned}$$

Labeling the edges with module index

This algorithm labels all edges such that, all linked edges are labeled with the same module index, the module index being the one that occurs the most frequently within the linked edges.

Initiation:

1. Given the edges, count the occurrences of the vertices.
2. Label each edge to one of the two vertices with the higher occurrence.

Loop:

1. Count the edge label occurrences.
2. For each edge, if both vertices have nonzero label counts, produce a map from vertex of low count to vertex of high count.
3. If we have an empty map, terminate loop. Otherwise, apply the map to the edge labels.

4.3 Performance Improvement

Infomap performs greedy merges of two modules until no more merges can be performed. If there are n nodes in the graph and m modules in the end, then there are $n - m$ merges, since we assume the graph is sparse and the edges are proportional to the number of nodes. Since each loop merges two modules, there are $n-m-1$ loops. Thus, infomap has linear time complexity to the number of nodes/edges.

In the multiple merging scheme, within each loop each module merges with one other module. Let's assume in each loop, k modules merge into one on average. Then, let there be l loops, and as before, n nodes are merged into m modules. Since each loop reduces the amount of modules by k times, we have

$$nk^{-l} = m \tag{44}$$

$$k^l = \frac{n}{m} \tag{45}$$

$$l = \log_k n - \log_k m \tag{46}$$

Within each merge, there are $O(k)$ operations to aggregate the indices appropriately for the merges.

Thus, the overall time complexity is $O(k \log n)$. On one extreme, if $k = 2$, i.e., every pair of modules are merged every step, then we have $O(2 \log n)$ complexity, i.e., logarithmic complexity in the number of nodes/edges. On the other extreme, if $l = 1$ and $k = n/m$, then the overall complexity is $O(k) = O(n/m)$. This has worst case complexity equal to that of infomap, when $m = 1$, but otherwise the division by m in the parallel scheme is likely to present performance improvement compared to the subtraction by m complexity in infomap.

The central reason behind the logarithmic time complexity is the constant average merging of k modules into one. The constancy of this factor likely depends on the network structure. Given a sparse network, the number of edges is similar to the number of nodes. If we make the assumption that k is proportional to the number of edges, then after a loop, the number of modules is reduced by a factor of k, and so is the number of edges. Thus, the network sparsity remains unchanged, and k is unchanged also. Of course, ultimately, actual performance benchmarks are required.

The logarithmic complexity and the k constant suggests that perhaps enforced binary merging, i.e., either pair-wise merge or no merge at all for some modules, might achieve best runtime complexity. A possible catch-22 might be that, to enforce pair-wise merges, $O(k)$ explorations would be needed, so that the runtime complexity remains the same, and actual performance is even penalized. Further mathematical ideas, simulations and benchmarks would be required for further explorations.