CORE
Metadata, citation and similar papers at core.ac.uk
Provided by HAL-ENS-LYON

Discovering the structure of complex networks: Implementation and Complexity of the heuristic MACH

Ronan Hamon, Pierre Borgnat, Patrick Flandrin, Céline Robardet

To cite this version:

Ronan Hamon, Pierre Borgnat, Patrick Flandrin, Céline Robardet. Discovering the structure of complex networks: Implementation and Complexity of the heuristic MACH. [Technical Report] Laboratoire de Physique, CNRS 5672; LIRIS UMR CNRS 5205. 2015. <ensl-01160737>

HAL Id: ensl-01160737

https://hal-ens-lyon.archives-ouvertes.fr/ensl-01160737

Submitted on 7 Jun 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Discovering the structure of complex networks: Implementation and Complexity of the heuristic MACH

Ronan Hamon
Physics Laboratory, ENS Lyon 69007 Lyon, France
Pierre Borgnat
Physics Laboratory, ENS Lyon, CNRS, 69007 Lyon, France
Patrick Flandrin
Physics Laboratory, ENS Lyon, CNRS, 69007 Lyon, France
Céline Robardet
LIRIS, INSA Lyon, 69621 Villeurbanne, France

Getting a labeling of vertices close to the structure of the graph has been proven to be of interest in many applications e.g., to follow signals indexed by the vertices of the network. This question can be related to a graph labeling problem known as the cyclic bandwidth sum problem. It consists in finding a labeling of the vertices of an undirected and unweighted graph with distinct integers such that the sum of (cyclic) difference of labels of adjacent vertices is minimized. Although theoretical results exist that give optimal value of cyclic bandwidth sum for standard graphs, there are neither results in the general case, nor explicit methods to reach this optimal result. In addition to this lack of theoretical knowledge, only a few methods have been proposed to approximately solve this problem. This report describes the implementation and the complexity of the heuristic MACH, developed to find an approximate solution for the cyclic bandwidth sum problem, by following the structure of the graph. The heuristic is a two-step algorithm: the first step consists of traversing the graph to find a set of paths which follow the structure of the graph, using a similarity criterion based on the Jaccard index to jump from one vertex to the next one. The second step is the merging of all obtained paths, based on a greedy approach that extends a partial solution by inserting a new path at the position that minimizes the cyclic bandwidth sum.

1 Description of the heuristic MACH

The aim of the heuristic, introduced in [1], is to build a labeling by traversing the graph to discover its structure.

The heuristic we propose consists of a two-step algorithm. The first step performs local searches in order to find a collection of independent paths with respect to the local structure of the graph, while the second step determines the best way to arrange the paths such that the objective function of the Cyclic Bandwidth Sum is minimized.

1.1 Step 1: Guiding the search towards locally similar vertices

The first step consists in finding a collection of paths in the graph, that is to say some sequences of vertices consecutively connected. The algorithm performs a depth-first search in which the next vertex is chosen among the neighbors of the current vertex, in order to preserve the connectivity of the path, based on its similarity with the current vertex. This similarity depends on the intersection of the two vertex neighborhoods: the more the neighborhoods of the two vertices intersect, the closer their labels are. Concretely the search is executed as follows: Starting from a vertex, the algorithm jumps to the most similar neighbor not yet labeled, and so on until there are no more accessible vertices. Then, the algorithm starts a new path from a vertex which has not been yet inserted in a path, and then continues to build paths until all the vertices are in a path. At the end of this step, a collection of paths is obtained that partitions the graph vertex set.

Initialization Any vertex not yet inserted in a path can be used as starting node. However, to favor the computation of longer paths, vertices that are at the periphery of the graph are preferred. The incentive behind this choice lies in the fact that the path should start at one of the extremity of the graph if there is one. For example, let us consider a simple path graph: Starting from a vertex in the middle of the path will generate two paths, although it is obvious that the graph can be traversed using a single path. There are several measures to determine the centrality of a vertex, that can also be used to find vertices that are at the periphery of the graph. We chose the simplest one, to minimize the computational cost, by namely using the degree of the vertices: the vertex with the smallest degree is selected to start the path.

Construction of a path A path is obtained by performing a depth-first search where the next adjacent vertex is the one that (1) is not labeled and (2) has a neighborhood that is the most similar to the one of the current vertex. The neighborhood similarity of two vertices is evaluated based on the Jaccard index, a quantity used to compare the similarity between two sets by looking at the total number of common elements over the total number of elements. Let $\texttt{Adj}(u)$ be the adjacent vertices of the vertex u, i.e. the neighborhood of u, including the vertex u. The similarity index between the vertex u and v, denoted $J(u, v)$, is defined by:

$$J(u, v) = \frac{|\texttt{Adj}(u) \cap \texttt{Adj}(v)|}{|\texttt{Adj}(u) \cup \texttt{Adj}(v)|} \tag{1}$$

This measure is equal to 1 if the two vertices are connected and have the same adjacent vertices, otherwise it is strictly lower than 1. A value close to 0 means that the total number of vertices in the two neighborhoods is much higher than the number of common neighbors.

It is preferable that the adjacent vertices of degree 1 that are only adjacent, to the current vertex, are not chosen as the following vertices, despite their high similarity, because it would end up the path. These vertices are immediately inserted after their unique neighbor to guarantee that the vertices are as close in the labeling as they are in the graph. However, we let the traversal continue.

End of the search The search for a path ends when all the neighbors of the current vertex have been inserted in a path. The algorithm starts a new path using the remaining vertices, until all the vertices belong to a path.

1.2 Step 2: Greedy merger of paths

The second step of the algorithm aims at aggregating the paths obtained in Step 1 in a unique labeling. The results of this step is a list of vertices, where the position of the vertex in the list gives its label. The merging is performed using the following process: From an empty list, the paths are sequentially added in such a way that at each iteration, the value of CBS is locally minimized. More precisely, for a given partial labeling and a given path, the position in which the path is inserted in the current labeling, as well as the direction in which the path is inserted (i.e. if the path is reversed or not), are chosen such that the value of CBS, computed over the vertices in the partial labeling and in the path, is minimized. The paths are selected in turns according to their length, the largest one being selected first. The rational behind this choice is so that to broadly explore the space of solutions.

Incremental computing of the CBS Evaluating the CBS is demanding computationally, as it requires considering every edge of the graph. For each insertion of path, the current CBS is computed twice (ordered and reverse ordered) for each possible insertion index of the current labeling. If the partial labeling has n vertices, there are $n + 1$ possible indices in which the path can be inserted, and then $2(n + 1)$ computations of the value of CBS required to select the best index. It is thus very costly, but can be largely alleviated by observing that, from an index to the next one, many edges have the same contribution in the total CBS value. From this perspective, we propose an incremental update of the CBS to take into account the state before the shift: At each update, only the edges whose adjacent vertex labels have been modified are considered.

We adopt the following notations: at a given iteration, the partial ordering is denoted O and contains n_o vertices while the path is denoted P and contains n_p vertices. Let i be the index of insertion, with $i \in \{0, \dots, n_o\}$. The partial ordering is decomposed into three parts: the first part is denoted O_1 and is made of the vertices located before i. The vertex right after the position i is denoted k, while the remaining vertices compose the third part called O_2. With these notations, the insertion of the path P in the current labeling O at the index i means that the resulting labeling is composed of the list O_1 followed by the path P, the vertex k and terminated by the list O_2. Likewise, the insertion of the path P in the current labeling O at the index $i + 1$ means that the resulting labeling is composed of the list O_1 followed by the vertex k, the path P, and terminated by the list O_2.

Using this representation, it is clear that the positions in the partial ordering and then the corresponding labels change only for the vertices in P and the vertex k. Let $\pi_i[u]$ be the label of vertex u when P is inserted

at index i, i.e. after the vertex at the ith position in the partial labeling. The changes in the labels for each group of vertices are the following:

$$\pi_{i+1}[k] = \pi_i[k] - n_p \tag{2}$$
$$\forall u \in P,\ \pi_{i+1}[u] = \pi_i[u] + 1 \tag{3}$$
$$\forall u \in O_1,\ \pi_{i+1}[u] = \pi_i[u] \tag{4}$$
$$\forall u \in O_2,\ \pi_{i+1}[u] = \pi_i[u] \tag{5}$$

We denote by $CBS^{(i)}$ the value of the cyclic bandwidth sum when P is inserted at index i. The computation of $CBS^{(i)}$ can be decomposed according to the different groups of vertices defined above:

$$\begin{aligned} CBS^{(i)} &= CBS^{(i)}(O_1, O_1) + CBS^{(i)}(O_2, O_2) + CBS^{(i)}(O_1, O_2) + CBS^{(i)}(P, P) \\ &+ CBS^{(i)}(k, O_1) + CBS^{(i)}(k, O_2) + CBS^{(i)}(k, P) \\ &+ CBS^{(i)}(P, O_1) + CBS^{(i)}(P, O_2) \end{aligned} \tag{6}$$

where $CBS^{(i)}(X, Y) = \sum_{u \in X, v \in Y, \{u,v\} \in E} d_C(\pi_i[u], \pi_i[v])$ is the value of CBS when only the edges of the graph between the two sets X and Y are considered, with $d_C(\pi[u], \pi[v])$ defined by:

$$d_C(\pi[u], \pi[v]) = \min\{|\pi[u] - \pi[v]|, n - |\pi[u] - \pi[v]|\} \tag{7}$$

The definition of the distance d_C shows trivially that if the labels of the endpoint vertices of an edge are not shifted or are shifted equally, then the value of d_C remains the same:

$$CBS^{(i+1)}(O_1, O_1) = CBS^{(i)}(O_1, O_1) \tag{8}$$
$$CBS^{(i+1)}(O_2, O_2) = CBS^{(i)}(O_2, O_2) \tag{9}$$
$$CBS^{(i+1)}(O_1, O_2) = CBS^{(i)}(O_1, O_2) \tag{10}$$
$$CBS^{(i+1)}(P, P) = CBS^{(i)}(P, P) \tag{11}$$

When the labels of endpoint vertices are not shifted equally, it is necessary to consider not only the changes induced by the shift, but also which terms between $|\pi[u] - \pi[v]|$ and $n_o - |\pi[u] - \pi[v]|$ is the minimum, both at index i and $i + 1$, as it can vary. We prove in the following the results when the endpoint vertices are k and a vertex in O_1. The other results follows the same reasoning.

Theorem 1.1. *(Edges between k and the vertices of O_1)*
Let $u \in O_1$ and $\Delta = \pi_i[k] - \pi_i[u]$. We have:

1. *if $\Delta \leq \frac{n}{2}$ then $CBS^{(i+1)}(k, u) = CBS^{(i)}(k, u) - n_p$.*
2. *if $\Delta \geq \frac{n}{2} + n_p$ then $CBS^{(i+1)}(k, u) = CBS^{(i)}(k, u) + n_p$.*
3. *if $\frac{n}{2} < \Delta < \frac{n}{2} + n_p$ then $CBS^{(i+1)}(k, u) = CBS^{(i)}(k, u) + 2\Delta - (n_o + n_p)$*

1.3 Detailed algorithm

The whole algorithm MACH comprises the consecutive execution of two steps, introduced in the previous Section. For readability, the algorithm of each step is described separately, respectively in Algorithms 1 and 2.

From a connected, unweighted, and undirected graph $G = (V, E)$ with n vertices, the algorithm outputs a one-to-one mapping π from V to $\{0, \cdots, n-1\}$. We consider in the following a `List` as a list of elements with the associated functions `List-Insert`(A, a, idx) which inserts the element a in the list A at the index idx (if idx is not given, the element a is inserted at the end of the list), `List-Remove`(A, a) which removes the element a from the list A, `Length`(A) which returns the number of elements of the list A, and the function `Reverse`(A), which returns the list A in the reverse order. The function `Degree`(u) returns the degree of the vertex u in the graph G, i.e. the number of vertices adjacent to the vertex u. Finally `Adj`(u) returns the adjacent vertices of u.

Algorithm 1 computes the first step of the heuristic as described in Section 1.1. It consists in building a collection of paths containing the vertices of the graph, each path traversing the graph following its structure. Line 1 initializes a list S containing all the vertices of the graph, while Line 2 initializes an empty list which will contain the paths . The search of paths (Lines 3 to 26) is then performed until all vertices are included in a path. A vertex of S minimum degree value is considered (Line 4). The selected vertex, noted u_0, is removed from S (Line 5) and is the starting vertex of the search from Line 8 to Line 24. The path is defined as a sequence of vertices added in a list P (Line 7) and is closed when there are no more successors available to extend the

Algorithm 1 Step 1: Guiding the search towards locally similar vertices

Require: $G = (V, E)$
Ensure: $Paths$
1: $S = \texttt{List}(V)$
2: $Paths \leftarrow \texttt{List}()$
3: **while** S is not empty **do**
4: $u_0 \leftarrow \arg\min_{u \in S} \texttt{Degree}(u)$
5: $\texttt{List-Remove}(S, u_0)$
6: $exist\_successors \leftarrow \texttt{True}$
7: $P \leftarrow \texttt{List}()$
8: **while** $exist\_successors$ **do**
9: $\texttt{List-Insert}(P, u_0)$
10: $H \leftarrow \texttt{List}()$
11: **for all** $v \in \texttt{Adj}(u_0) \cap S$ **do**
12: **if** $\texttt{Degree}(v) = 1$ **then**
13: $\texttt{List-Insert}(P, v)$
14: $\texttt{List-Remove}(S, v)$
15: **else**
16: $\texttt{List-Insert}(H, v)$
17: **end if**
18: **end for**
19: **if** H is not empty **then**
20: $u_0 \leftarrow \arg\max_{w \in H} \texttt{Similarity\_Index}(u, w)$
21: **else**
22: $exist\_successors \leftarrow \texttt{False}$
23: **end if**
24: **end while**
25: $\texttt{List-Insert}(Paths, P)$
26: **end while**

path or when the depth-first search ends. The first step of this loop consists of adding the vertex u_0 to the path P (Line 9). A new list H is then initialized (Line 10) and will contain the potential successors of u_0. These successors are selected among the adjacent vertices of u_0 which are still in the list S, i.e. which have not been included in a path beforehand (Line 11). For each of the successors, noted v, if the degree of v is equal to 1, i.e. the vertex v has only the vertex u_0 as adjacent vertex, then v is directly added in the path (Lines 12 to 14). Otherwise, it is added to the list H (Line 16). When all the potential successors have been either added to P or H, the next vertex to be considered is chosen among the elements of H, as the one with the highest similarity with the current u_0 according to Eq 1 and given by the function `Similarity_Index`. The heuristic then loops using the updated value of u_0. If H is empty, `exist_successors` is set to `False` (Line 22) and the path is inserted in the list of paths (Line 25). If S is not empty, then u_0 is updated using the procedure described in Line 4 and the search of a path from this vertex is repeated. When S is empty, the first step is completed.

Algorithm 2 computes the second step of the heuristic as described in Section 1.2. A list $Order$ is first initialized as the path in $Paths$ with the highest number of elements (Line 1). This path is then removed from the list $Paths$, and the algorithm inserts all the remaining path in the list $Order$ using a loop from Line 3 to Line 12. The path with the highest number of elements is selected (Line 4). The function `Incremental_CBS` returns the index and the direction of insertion of P that minimizes the CBS. Depending on the value of the boolean variable $reverse$, the path P_0 is inserted reversed (Line 7) or not (Line 9). The path is then removed from the list of paths $Paths$ and the heuristic loops until all the paths have been inserted in $Order$. As the result, the mapping π is built using the vertices as keys and the index of the vertices in the list $Order$ as values.

1.4 Worst-case complexity of the algorithm

We now examine the worst-case time complexity of the algorithm MACH described in the previous section, when applied on a graph $G(V, E)$ with $|V| = n$ and $|E| = m$.

We first examine the complexity of Algorithm 1. The set S initialized Line 1 can be implemented as a min-priority queue with a binary min-heap. The time to build the binary min-heap is $O(n)$. Lines 4 and 5 can be done using the EXTRACT-MIN function that takes time $O(\log n)$. Similarly, the set $Paths$ can be implemented as a max-priority queue with a binary max-heap and Line 13 takes in the worst case a time proportional to the logarithm of the number of paths, that is in the worst case $O(\log n)$. Using aggregate analysis, the `while` loop in Line 8 is executed at most once for each vertex of V, since the vertex u_0 is removed from S (Line 5). The

Algorithm 2 Step 2: Greedy merge of paths

Require: $Paths$
1: $Order \leftarrow \arg\max_{P \in Paths} \texttt{Length}(P)$
2: `List-Remove`($Paths$, $Order$)
3: **while** $Paths$ is not empty **do**
4: $P_0 \leftarrow \arg\max_{P \in Paths} \texttt{Length}(P)$
5: $idx, reverse \leftarrow$ `Incremental_CBS`($Order, P_0$)
6: **if** $reverse$ is `True` **then**
7: `Insert-List`($Order$, `Reverse`(P_0), idx)
8: **else**
9: `Insert-List`($Order, P_0, idx$)
10: **end if**
11: `List-Remove`($Paths, P_0$)
12: **end while**
13:
14: $i \leftarrow 0$
15: **for** $i = 0 : (n-1)$ **do**
16: $\pi[Order[i]] \leftarrow i$
17: **end for**
18: **return** π

function `List-Insert` is in constant time. The set H of vertices that are adjacent to u and in S is implemented as a max-priority queue using a binary heap data structure that makes possible to run Max_Heap_Insert, that inserts a new element into H (Line 16) while maintaining the heap property of H in $O(\log(|H|))$, that is in the worst case in $O(\log(|\texttt{Adj}[u]|))$. Thus, the loop on Lines 8-18 in is executed $|\texttt{Adj}[u]|$ times and at each iteration (1) the similarity index computation takes time $\Theta(\min(|\texttt{Adj}[u]|, |\texttt{Adj}[v]|))$, and (2) Max_Heap_Insert takes time $O(\log(|\texttt{Adj}[u]|))$. Therefore, the loop on Lines 8-18 is in $O(|\texttt{Adj}[u]|^2)$. Line 20 can be done using Extract_Max in time $O(\log(|\texttt{Adj}[u]|))$ and the total complexity of Lines 8-24 is in $O(|\texttt{Adj}[u]|^2)$. Consequently, the total time is $O(\sum_{u \in V}(|\texttt{Adj}[u]|^2))$. As K. Das established in [2] that

$$\sum_{u \in V} |\texttt{Adj}[u]|^2 \leq m\left(\frac{2m}{n-1} + n - 2\right) \tag{12}$$

we can conclude that the total cost of Algorithm 1 is in $O(n \log n + mn) = O(mn)$

We can also use an aggregate analysis to evaluate the time taken by the Algorithm 2. Lines 2 and 5 are in $O(n)$, when almost all the vertices have already been merged in `Order`. `Incremental_CBS` runs through (1) all the edges between the vertices of the current path P and the ones of `Order` and (2) between the vertex of `Order` at position `Position` and the other vertices of `Order` $\cup P$.

- Step (1) takes $O(mn)$ since all the edges of the graph are examined when aggregating the analysis over the all paths: the adjacency list of each vertex is examined once. Furthermore, for each of these m edges, the n positions of `Order` are evaluated.
- Step (2) is also in $O(mn)$ since aggregating the adjacency lists of the vertices of `Order` leads to the m edges of the graph that are evaluated for each of the at most n paths.

The other instructions of the loop are executed in constant time. Therefore, the total time spent in Algorithm 2 is $O(mn)$.

Finally, we have that the whole algorithm has a worst case complexity in $O(mn)$.

References

[1] R. Hamon, P. Borgnat, P. Flandrin, and C. Robardet, *Discovering the structure of complex networks by minimizing cyclic bandwidth sum*, Preprint arXiv:1410.6108, 2015.

[2] K. C. Das, *Sharp bounds for the sum of the squares of the degrees of a graph*, Kragujevac J. Math, vol. 25, pp. 31–49, 2003.