Computing the Probability Vectors for Random Walks on Graphs with Bounded Arboricity

Gaurav Goel

Computing the Probability Vectors for Random Walks on Graphs with Bounded Arboricity*

Gaurav Goel
IIT Delhi, India
Internship at INRIA Lorraine, Nancy, France

Supervised by: Dr. Jens Gustedt

The problem of detecting dense subgraphs (*communities*) in large sparse graphs is inherent to many real world domains like social networking. A popular approach of detecting these communities involves first computing the *probability vectors* for *random walks* on the graph for a fixed number of steps, and then using these probability vectors to detect the communities. Such an approach has been discussed by Latapy and Pons in [5]. They compute the probability vectors using simple matrix multiplication and define a measure of the structural similarity between vertices which they call *distance*. Based on the probability vectors, they compute the distances between vertices and then based on these distances group the vertices into communities. Their algorithm takes $O(n^2 \log n)$ time where n is the number of vertices in the graph. We focus on the first part of the approach i.e. computation of the probability vectors for the random walks, and propose a more efficient algorithm (than matrix multiplication) for computing these vectors in time complexity that is linear in the size of the output.

*This internship has been supported by the joint IIT-INRIA internship programme and a grant by the *Région Lorraine*.

1 Introduction

Consider a few real world large sparse graphs — the World Wide Web (WWW) graph where the vertices are HTML pages connected by links (edges) pointing from one page to another, the social acquaintance network where the vertices represent people and the edges represent the association between them, the graph representing the citation pattern of scientific publications with the vertices being the publications and the edges being the links to the articles cited in a publication, or for that matter the collaboration graph of movie actors with the vertices representing actors and edges joining actors which have worked together in at least one movie. All of these graphs and most other real world sparse graphs have a unique property — they have a bounded *arboricity*. Arboricity can be defined as follows:

Definition 1 *A graph G is said to have arboricity k, if k is the smallest integer for which there exists forests $T_1 \dots T_k$ which are subgraphs of G, such that the union of all of them is the entire G.*

A graph G with arboricity k has two important properties which are stated in the following two lemmas:

Lemma 1 *It has average degree $z \leq 2 * k$.*

Lemma 2 *All subgraphs of G have arboricity k.*

Proof: Take any subgraph G' of G. Consider forests $T'_1 \dots T'_k$ in G' s.t. T'_i is the largest possible subgraph of T_i in G'. Such forests will always exist since any subgraph of a forest is also a forest. The union of these new forests will give the entire G'. This implies that the arboricity of G' is also k (by Definition 1). □

In fact not just real world sparse graphs, but all minor closed graph families and all random graphs that are generated by the model discussed by Barabasi and Albert in [1], also have a bounded arboricity. All graphs mentioned above and most real world complex networks, have 2 major characteristics:

Char 1: They are continuously expanding with new vertices being introduced every day.

Char 2: The new vertices tend to share more edges with those old vertices which are already well connected as compared to the ones which are less connected.

The model discussed in [1] takes into account these characteristics and achieves a scale invariant distribution of vertex degrees which is an inherent property of most real world networks.

The properties of graphs with bounded arboricity along with the approach discussed by Gustedt in [4] for computing minimum spanning trees for minor closed graph classes, form the basis of the algorithm discussed in this document. The method discussed in [4]

divides the graph into portions based on the degrees of vertices, computes the required quantities only in one of the portions, passes the results to the remaining graph and finally recurses the procedure in this remaining graph.

2 Problem Definition

Before going into the problem specifications, it is necessary to define random walks in our context:

Definition 2 *Consider a graph G. A random walk starting from a vertex v, is a sequence of neighboring vertices (beginning with v) in G, each one being chosen with a predefined probability for the edge between itself and the vertex before it in the sequence.*

Input: An undirected graph $G = (V, E)$ with average degree of each vertex z and a bounded arboricity k. It is imperative for every vertex in G to have a self loop. If a vertex doesnot have a self loop, then introduce one. Once at a particular vertex v_i during the random walk, the probability of choosing any of the outgoing edges in the next step is the same p for all edges and the probability of staying at the same vertex in the next step is $(1 - (p * d_i))$ where d_i is the degree of vertex v_i. Such a *symmetric* probability distribution deviates a bit from what is found in literature but it ensures the following useful property for the graph:

Lemma 3 *If in G the probability of going from a vertex v_j to v_k through a m step random walk is pr, then the probability of going from v_k to v_j through a m step random walk is also pr.*

Proof: Consider a m step path from v_j to v_k in G. The total probability of this path being chosen during a random walk is the product of the probabilities of choosing the edges along this path. Note that no matter which side of an edge you are looking from, the probability of choosing a particular edge is the same. Therefore it can be said that the probability of the reverse of this path (i.e. from v_k to v_j) being chosen during the random walk is also the same. This is true for all paths connecting v_j and v_k. Hence it follows that the probability of going from v_k to v_j through a m step random walk is also pr. □

Size of Input: $O(n + m)$ where n is the number of vertices and m is the number of edges in the given graph. Let this be denoted by N.

Output: A vector P_i for every vertex v_i which contains the probabilities of reaching all other vertices in d steps if a random walk is started from v_i. For computational efficiency, this probability vector is maintained as a vector of tuples, each containing a vertex number and the non zero probability of reaching that vertex in d steps from v_i through a random walk. Vertices which cannot be reached in d steps from v_i are not present in this vector.

Size of Output: The total number of probability vectors obtained are n and the length of each one of them is between d and n. So the total size of output is between nd and n^2. Let this be denoted by M.

3 Our Approach and Result

Our algorithm divides the given graph (of bounded arboricity) into a *core* and a *periphery* based on some parameter which is a function of the average degree z of the vertices. The vertices in the periphery have a lower and bounded degrees whereas those in the core have high degrees. Computations are first done on the periphery, then the information is passed on to the core. The core in itself is a graph of bounded arboricity and thus the procedure is recursed on the core. Once the innermost core is reached then the direction of flow of information changes and the information starts flowing from the core to the periphery till it reaches the outermost periphery. This entire process is repeated a number of times which is a function of the length of the random walks d, to get the final probability vectors. This approach makes it possible to compute the probability vectors in time complexity that is linear in the size of these vectors i.e. in $O(M)$ time which is a major improvement over the existing algorithm.

4 The Algorithm

For the sake of simplicity for the moment assume that there is only one core and one periphery, although the approach can be easily extended for multiple levels as discussed in Section 7 of the document. The algorithm can be broadly divided into 5 steps each one of which is discussed in the next five subsections.

Step 1 Dividing the graph in to a core and a periphery

Look at the degree of each vertex. Call the set of vertices with degree less than or equal to the ceiling of z, the **periphery**; and the set of vertices with degree more than that, the **core**. Note that z is bounded by the arboricity k of the graph (by Lemma 1). The core alone (on removing the edges which go into the periphery) in itself is a graph with an average degree which is again bounded by the same arboricity k (by Lemma 2). According to the assumption, there is only one core and one periphery. This means that the degree of each vertex in the core alone is also less than or equal to the ceiling of its average degree.

Step 2 Computations inside the core

Consider the core alone. Initially each vertex v_i in the core would have a probability vector consisiting of its neighbours and the probability of reaching them in 1 step of a random walk. Call this probability vector P_i^1. Now do computations in $(d-1)$ steps (d is the length of the random walk) such that in the j^{th} step, each vertex v_i collects the latest

probability vectors $P_k^j s$ (containing the probabilities of reaching different vertices in j steps if a random walk is started from that particular vertex) from all its neighbours and uses them to compute its own P_i^{j+1}. This can easily be done by simlpy going through all the vectors once. Thus at the end of these $(d-1)$ steps, each vertex v_i in the core would have d probability vectors $P_i^1 \dots P_i^d$ that would contain the probabilities of reaching different vertices inside the core alone in $1 \dots d$ steps of random walk.

Step 3 Computations in the periphery

For each vertex v_i in the periphery compute its probability vectors $P_i^1 \dots P_i^d$ in the same way as for the vertices in the core in Step 2, but with the difference that some of the neighbours of these vertices are in the core, and have their vectors precomputed in Step 2. These vectors can be used as it is by the vertices in the periphery. The $P_i^1 \dots P_i^d$ computed in this step would contain the probabilities of reaching different vertices in $1 \dots d$ steps of random walk by going from the periphery to the core atmost once. Note that the degree of each vertex in the periphery is bounded (even on including the neighbours which are in the core).

Step 4 Passing of information — periphery to core

In the given setting, if the probability of going from a vertex v_j to v_k through a m step ramdom walk is pr, then the probabilty of going from v_k to v_j through a m step random walk is also pr (by Lemma 3). Using this fact, for every vertex in the periphery look at its d probability vectors, search for probabilities of reaching vertices in the core which has a neighbour in the periphery, and reverse this information i.e. if a vertex v_j in the periphery has the knowledge that it can reach another vertex v_k in the core (which has a neighbour in the periphery) in m steps of random walk with probability p, then give this information to v_k, telling it that it can reach v_j in m steps of random walk with probability p. This step will generate additional probability vectors $P_i'^1 \dots P_i'^d$ for some vertices v_i in the core. A second type of information which needs to be passed on to the core at this step is the following: If a vertex v_i in the periphery knows that it can reach vertices v_j and v_k in the core in m_j and m_k steps of random walk with probabilities p_j and p_k respectively, then it needs to *stich* these two informations together (only if $(m_j + m_k)$ is $\leq d$) and pass this combined information to both v_j and v_k, telling each one of them that they can reach the other in $(m_j + m_k)$ steps of random walk with probability $(p_j * p_k)$. To avoid duplication, this *stiching* of information needs to be done only by those vertices in the periphery which have a neighbour in the core and needs to be passed to v_j and v_k only if they are their neighbours. Note that the size of the information passed in this step is bounded by the size of the output, M.

Step 5 Processing of information in the core

For each vertex in the core which has a neighbour in the periphery, process the information received in the last part of Step 4 and merge it with the probability vectors

$P_i'^1 \dots P_i'^d$ to get new vectors $P_i''^1 \dots P_i''^d$. Further, for each vertex v_i in the core compute its probability vectors $P_i^1 \dots P_i^d$ in the same way as in Step 2, but also taking into account the already available information in the form of vectors $P_i''^1 \dots P_i''^d$ for some of the vertices.

Repeat Step 3 to Step 5, $(d/2)$ number of times to be able to take into account all paths of length d that exist in the entire graph and which cross the boundary between the core and the periphery any number of times. The final probability vectors $P_i s$ would be the last computed $P_i^d s$.

5 Correctness

If we consider a path as a sequence of vertices then all paths of length d in G can be broadly classified into the following seven categories:

Cat 1: Paths which are totally inside the core i.e. paths in which all the vertices lie inside the core.

Cat 2: Paths which lie totally in the periphery.

Cat 3: Paths which start from a vertex in the core and end in a vertex in the periphery and which cross the boundary between the core and the periphery only once.

Cat 4: Paths which start from a vertex in the perphery and end in a vertex in the core and which cross the boundary only once.

Cat 5: Paths which both start and end in a vertex in the core and which cross the boundary twice.

Cat 6: Paths which both start and end in a vertex in the periphery and which cross the boundary twice.

Cat 7: Paths which cross the boundary more than twice.

If all paths lying in any of the above categories starting from a particular vertex are taken into account exactly once in the calculation of the final probability vector for that vertex, then the algorithm can be said to be correct.

Paths of different categories are taken into account in different steps of the algorithm:

Step 2 In this step all paths of Cat 1 are taken into account.

Step 3 All paths of Cat 2 and 3 are accounted for in the computations in this step.

Step 4 In this step all vertices in the core which have a neighbour in the periphery get to know about the probabilities of the paths of Cat 4 and 5.

Step 5 All vertices in the core get to know about the paths of Cat 4 and 5 by the end of computations in this step. Probabilities for paths of Cat 1 are also recomputed in this step.

When recursion takes place for the first time and computations are done in the periphery, paths of Cat 6 are also accounted for in addition to the recomputation of probabilities of Cat 2 and 3 paths.

Paths of Cat 7 are accounted for in the subsequent recursions as proved in the following lemma:

Lemma 4 *All paths of length d which cross the boundary between the core and the periphery any number of times are taken into account by the algorithm.*

Proof: By Induction.

Base case: All paths of length d which cross the boundary between the core and the periphery once or twice are taken into account by the algorithm.

This has already been shown above while discussing paths of Cat 1 to Cat 6.

Induction Hypothesis: All paths of length d which cross the boundary between the core and the periphery less than or equal to k times are taken into account by the algorithm.

Induction Step: Consider a path of length d which crosses the boundary between the core and the periphery $(k+1)$ times.

Keep on removing vertices from the tail of the path till the remaining path is such that it crosses the boundary k times. By induction hypothesis, this remaining path is being taken into account by the algorithm. Without loss of generality we can assume that this happens at the end of the j^{th} recursion. Now there are two possibilities:

1) The last removed vertex lies in the periphery and the last vertex of the remaining path lies in the core.

   In this case the given path would be taken into account in Step 3 of the $(j+1)^{th}$ recursion.

2) The last removed vertex lies in the core and the last vertex of the remaining path lies in the periphery.

   In this case the given path would be taken into account in Step 5 of the $(j+1)^{th}$ recursion.

□

Hence we can claim that all paths of length d are taken into account by the algorithm.

Above arguement shows that all the paths are taken into account in some or the other step of the algorithm. But it is also necessary to argue that each path is taken

into account only once. This is being ensured in the algorithm by discarding the old probability vectors once the new ones are computed both for the core as well as for the periphery. Also the computations are being done in such a way that double counting of any path in a single step of the algorithm is not possible.

6 Time Complexity

The break up of time consumed in each step is as follows :

Step 1 Linear in the number of vertices, i.e. $O(n)$.

Step 2 $O(n' * d * z * v')$ where n' is the number of vertices in the core and v' is the bound on the size of the probability vector. This is because the computation is done $(d-1)$ times for every vertex in the core and each computation involves looking at the probablity vector (bounded by v') of the at most $(z+1)$ neighbours. This is again $O(n)$.

Step 3 Note that the sum of the vectors of all vertices obtained at the end of this step is bounded by the size of the output M. And again the computation is done $(d-1)$ times for every vertex and each computation involves looking at the probablity vectors (each one of which cannot in itself be greater than the final vector obtained for the vertex under consideration) of at most $(z+1)$ neighbours. Therefore, the time consumed in this step is $O(d * z * M)$ or $O(M)$.

Step 4 Time consumed in this step is clearly $O(M)$.

Step 5 The total amount of information received from the previous step is bounded by M, and it can be merged together in linear time. Also the complexity of the second part (computation of probability vectors) of this step if the same as that of Step 3. Therefore the total time consumed is this step also is $O(M)$.

Hence the total time complexity of the algorithm becomes $O(M) * (d/2)$ which is again $O(M)$. To be more specific, the complexity can be written as $c * f(d,z) * M$, where c is a constant, $f(d,z)$ is $(d^2 * z)/2$ and M is the size of the output.

7 Extension of the Algorithm for Multiple Levels

The algorithm discussed above can easily be extended for multiple levels i.e. for multiple peripheries. Step 1 of the algorithm remains the same except that the core now has to be recursively divided on the basis of degree of vertices till a core is reached in which the degree of vertices is bounded by the ceiling of z. So, by the end of this step there are multiple peripheries $L_1 \ldots L_k$ (L_1 being the innermost one and L_k being the outtermost), and one core (innermost of all layers). Step 2 remains totally the same and is carried out on the core. Step 3 and Step 4 have to be carried out together starting from the L_1

and ending at L_k i.e. computations are first done in L_1 and information is passed from here to the core; then computations are done in L_2 and information is passed from here to L_1 and the core; and so on till L_k is reached. Step 5 is carried out in the opposite direction i.e. from L_{k-1} to the core. In this step all the information received from all the outter layers is combined to get a single set of probability vectors $P_i^{''1} \dots P''d_i$ and then computations are done within the layer as are done in the core in the dual level algorithm. Finally, Step 3 to Step 5 are repeated $(d/2)$ number of times and the final probability vectors are computed in the same way as in the previous algorithm. The number of peripheries in a graph are logarithmic in the number of vertices n and the base b of the log is a function of the parameter used for dividing the graph on the basis of the degree of vertices, in our case the ceiling of the average degree z.

8 Correctness

Consider the core and the multiple peripheries as layers numbered in increasing order, with the core being the lowest numbered layer and the outtermost periphery being the highest numbered. Also consider a path as a sequence of vertices. Then all paths of length d in G can be broadly classified into the following seven categories:

Cat 1: Paths which are totally inside the core i.e. paths in which all the vertices lie inside the core.

Cat 2: Paths which lie totally in a particular periphery.

Cat 3: Paths which start from a vertex in a lower numbered layer and end in a vertex in a higher numbered layer and donot change direction i.e. as you move along the paths, the layer number never decreases.

Cat 4: Paths which start from a vertex in a higher numbered layer and end in a vertex in a lower numbered layer and donot change direction i.e. as you move along the paths, the layer number never increases.

Cat 5: Paths which change direction exactly once and that to in the highest numbered layer amongst the layers through which they pass i.e. as you move along the paths, the layer number first increases and then decreases.

Cat 6: Paths which change direction exactly once and that to in the lowest numbered layer amongst the layers through which they pass i.e. as you move along the paths, the layer number first decreases and then increases.

Cat 7: Paths which change direction more than once.

If all paths lying in any of the above categories starting from a particular vertex are taken into account exactly once in the calculation of the final probability vector for that vertex, then the algorithm can be said to be correct.

Paths of different categories are taken into account in different steps of the algorithm:

Step 2 In this step all paths of Cat 1 are taken into account.

Step 3 All paths of Cat 2 and 3 are accounted for in the computations in this step.

Step 4 In this step all vertices which have a neighbour in any of the outter (higher numbered than itself) layers get to know about the probabilities of paths of Cat 4 and 5.

Step 5 All vertices get to know about the paths of Cat 4 and 5 by the end of computations in this step. Probabilities for path of Cat 1 (in case of core) and Cat 2 (in case of peripheries) are also recomputed in this step.

When recursion takes place for the first time and computations are done in the peripheries, paths of Cat 6 are also accounted for in addition to the recomputation of probabilities of Cat 2 and 3 paths. Paths of Cat 7 are accounted for in the subsequent recursions. This can be proved by an induction arguement in exactly the same way as is done in Section 5 for the dual level algorithm.

Above arguement shows that all the paths are taken into account in some or the other step of the algorithm. But it is also necessary to argue that each path is taken into account only once. This is being ensured in the algorithm by discarding the old probability vectors once the new ones are computed both for the core as well as for the peripheries. Also the computations are being done in such a way that double counting of any path in a single step of the algorithm is not possible.

9 Time Complexity

The break up of time consumed in each step is as follows:

Step 1 Linear in the number of vertices i.e. $O(n)$. Although the division between the core and the periphery is done a number of times but at each division, the number of vertices under consideration is less than a fixed fraction $(1/b)$ of the vertices at the previous stage and thus the complexity remains $O(n)$.

Step 2 $O(n' * d * z * v')$ where n' is the number of vertices in the core and v' is the bound on the size of the probability vector. This is because the computation is done $(d-1)$ times for every vertex in the core and each computation involves looking at the probablity vector (bounded by v') of the at most $(z+1)$ neighbours. This is again $O(n)$.

Step 3 Note that the sum of the vectors of all vertices (of any periphery) obtained at the end of this step is bounded by the size of the output M. And again the computation is done $(d-1)$ times for every vertex and each computation involves looking at the probablity vectors (each one of which cannot in itself be greater than the final vector obtained for the vertex under consideration) of at most $(z+1)$ neighbours. Therefore, the time consumed in this step is $O(d * z * M)$ or $O(M)$.

Step 4 Time consumed in this step is clearly $O(M)$.

Step 5 The total amount of information received from the previous step is bounded by M, and it can be merged together in linear time. Also the complexity of the second part (computation of probability vectors) of this step if the same as that of Step 3. Therefore the total time consumed is this step also is $O(M)$.

Hence the total time complexity of the algorithm becomes $O(M) * (d/2)$ which is again $O(M)$. To be more specific, the complexity can be written as $c * f(d, z) * M$, where c is a constant, $f(d, z)$ is $(d^2 * z)/2$ and M is the size of the output.

10 Conclusion and Further Work

We proposed an *output sensitive* algorithm for computing probability vectors for random walks on graphs with bounded arboricity. We first gave an algorithm for the simple case of one periphery and one core and latter extended it for multiple levels. Our algorithm takes $O(M)$ time.

An important parameter in our algorithm is the one that divides the graph into a core and a periphery. We have taken it to be the ceiling of the average degree of the graph, but the optimum value of this parameter is yet to be estimated. We have been able to achieve theoritical improvement in complexity and feel that our algorithm is well suited for coarse grained parallel computing models. We hope to do some work in this direction using the *SSCRAP* [2] library.

The approach we discussed gives an alternative and more efficient way to compute the probability vectors for random walks, but it doesnot change the community structure that emerge from the approach discussed by Latapy and Pons in [5]. However, if after the computation of probability vectors, the method to detect communities is also altered, then it would lead to changes in the final community structure. This part of the problem is still open. A good approach could be to use union-find strategies which have linear complexity in some conditions (see [3]). Any such approach would require an oracle on the basis of which smaller communities would be joined to form bigger ones till a community structure emerges that is satisfactory according to some predefined criteria. We hope to do some work in this direction also.

References

[1] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

[2] Mohamed Essaïdi, Isabelle Guérin Lassous, and Jens Gustedt. SSCRAP: Soft synchronized computing in rounds for adequate parallelization. *RR INRIA*, May 2004.

[3] Christophe Fiorio and Jens Gustedt. Two linear time union-find strategies for image processing. *Theoretical Computer Science*, 154:165–181, 1996.

[4] Jens Gustedt. Minimum spanning trees for minor-closed graph classes in parallel. In *Symposium on Theoretical Aspects of Computer Science*, pages 421–431, 1998.

[5] Matthieu Latapy and Pascal Pons. Computing communities in large networks using random walks.