Faster unfolding of communities: speeding up the Louvain algorithm.

V.A. Traag[1, 2, *]

[1] *Royal Netherlands Institute of Southeast Asian and Caribbean Studies*
[2] *e-Humanities Group, Royal Netherlands Academy of Arts and Sciences*

Many complex networks exhibit a modular structure of densely connected groups of nodes. Usually, such a modular structure is uncovered by the optimisation of some quality function. Although flawed, Modularity remains one of the most popular quality functions. The Louvain algorithm was originally developed for optimising Modularity, but has been applied to a variety of methods. As such, speeding up the Louvain algorithm, enables the analysis of larger graphs in a shorter time for various methods. We here suggest to consider moving nodes to the community of a random neighbour, instead of the best neighbouring community. Although incredibly simple, it reduces the theoretical runtime complexity from $\mathcal{O}(m)$ to $\mathcal{O}(n \log\langle k\rangle)$ in networks with a clear community structure. In benchmark networks, resembling real networks more closely, it speeds up the algorithm roughly 2–3 times. This is due to two factors: (1) a random neighbour is likely to be in a "good" community; and (2) random neighbours are likely to be hubs, helping the convergence. Finally, the performance gain only slightly diminishes the quality, thus providing an excellent quality-performance ratio. However, these gains do not seem to hold up when detecting small communities in large graphs.

I. INTRODUCTION

Complex networks have gained attention the past decade [1]. Especially with the rise of social media, social networks of unprecedented size became available, which contributed to the establishment of the computational social sciences [2, 3]. But networks are also common in disciplines such as biology [4] and neurology [5]. Many of these networks share various common characteristics. They often have skewed degree distributions [6], show a high clustering and a low average path length [7]. Nodes often cluster together in dense groups, usually called communities. Nodes in a community often share other characteristics: metabolites show related functions [8] and people have a similar background [9]. Revealing the community structure can thus help to understand the network [10].

Modularity [11] remains one of the most popular measures in community detection, even though it is flawed. There have been many algorithms suggested for optimising Modularity. The original algorithm [11] created a full dendrogram and used Modularity to decide on a cutting point. It was quite slow, running in $\mathcal{O}(n^2 m)$, where n is the number of nodes and m the number of links. Many algorithms were quickly introduced to optimize Modularity, such as extremal optimization [12], simulated annealing [13, 14], spectral methods [15], greedy methods [16], and many other methods [10]. One of the fastest and most effective algorithms is the Louvain algorithm [17], believed to be running in $\mathcal{O}(m)$. It has been shown to perform very well in comparative benchmark tests [18]. The algorithm is largely independent of the objective function to optimize, and as such has been used for different methods [19–24]

We first briefly describe the algorithm, and introduce the terminology. We then describe our improvement, which we call the random neighbour Louvain, and argue why we expect it to function well. We derive estimates of the runtime complexity, and obtain $\mathcal{O}(m)$ for the original Louvain algorithm, in line with earlier results, and $\mathcal{O}(n \log\langle k\rangle)$ for our improvement, where $\langle k\rangle$ is the average degree. This makes it one of the fastest known algorithms for community detection to optimize an objective function. Whereas the original algorithm runs in linear time with respect to the number of edges, our improvement is essentially linear with respect to the number of nodes. Finally, we show on benchmark tests that our improvement indeed leads to significant reductions in running time, without losing much quality. However, these gains do not seem to hold up when detecting small communities in large graphs.

II. LOUVAIN ALGORITHM

Community detection tries to find a "good" partition for a certain graph. In other words, the input is some graph $G = (V, E)$ with $n = |V|$ nodes and $m = |E|$ edges. Each node has k_i neighbours, which is called the degree, which on average is $\langle k\rangle = \frac{2m}{n}$. The output is some partition $\mathcal{V} = \{V_1, V_2, \ldots, V_r\}$, where each $V_c \subseteq V$ is a set of nodes we call a community. We work with non-overlapping nodes, such that $V_c \cap V_d = \emptyset$ for all $c \neq d$ and all nodes will have to be in a community, so that $\bigcup V_c = V$. Alternatively, we denote by σ_i the community of node i, such that $\sigma_i = c$ if (and only if) $i \in V_c$. Both σ and $\mathcal{V}$ may be used interchangeably to refer to the partition. If the distinction is essential, we will explicitly state this.

The Louvain algorithm is suited for optimising a single objective function that specifies some quality of a partition. We denote such an objective function with $\mathcal{H}$,

* traag@kitlv.nl

which should be maximized. We use $\mathcal{H}(\sigma)$ and $\mathcal{H}(\mathcal{V})$ to mean the same thing. There are various choices for such an objective function, such as Modularity [11], Potts models [13, 19, 22], Significance [25], Surprise [26], Infomap [21] and many more. We won't specify any of the objective functions here, nor shall we discuss their (dis)advantages. We focus on the Louvain algorithm as a general optimisation scheme, rather than an optimisation procedure for any one particular method.

Briefly, the Louvain algorithm works as follows. The algorithm initially starts out with a partition where each node is in its own community (i.e. $\sigma_i = i$). So, initially, there are as many communities as there are nodes. It moves around nodes from one community to another, to try to improve $\mathcal{H}(\sigma)$. We denote by $\Delta\mathcal{H}(\sigma_i \mapsto c)$ the difference in moving node i to another community c. In particular, $\Delta\mathcal{H}(\sigma_i \mapsto c) = \mathcal{H}(\sigma') - \mathcal{H}(\sigma)$ where $\sigma'_j = \sigma_j$ for all $j \neq i$ and $\sigma'_i = c$, implying that if $\Delta\mathcal{H}(\sigma_i \mapsto c) > 0$, the objective function $\mathcal{H}$ is improved. At some point, the algorithm can no longer improve $\mathcal{H}$ by moving around individual nodes, at which point it aggregates the graph, and reiterates on the aggregated graph. We repeat this procedure as long as we can improve $\mathcal{H}(\sigma)$. The outline of the algorithm is displayed in Algorithm 1.

There are two key procedures: MoveNodes and Aggregate. The MoveNodes procedure displayed in Algorithm 1 loops over all nodes (in random order), and considers moving them to an alternative community. This procedure relies on SelectCommunity to select a (possibly) better community c. Only if the improvement $\Delta\mathcal{H}(\sigma_v \mapsto c) > 0$, we will actually move the node to community c. The Aggregate procedure may depend on the exact quality function $\mathcal{H}$ used. In particular, the aggregate graph G' should be constructed according to σ, such that $\mathcal{H}(G', \sigma') = \mathcal{H}(G, \sigma)$, where $\sigma'_i = i$ is the default initial partition. That is, the quality of the default partition of each node in its own community should be equal to the quality of the original partition σ on the original graph G. In Algorithm 1 a version is displayed which is suited for Modularity. Other methods may require additional variables to be used when aggregating the graph (e.g. [19]).

The only procedure that remains to be specified is SelectCommunity. In the original Louvain algorithm, this procedure commonly considers all possible neighbouring communities, and then greedily selects the best community. It is summarized in Algorithm 2.

We created a new flexible and fast implementation of the Louvain algorithm in `C++` for use in `python` using `igraph`. The implementation of the algorithm itself is quite detached from the objective function to optimize. In particular, all that is required to implement a new objective function is the difference when moving a node $\Delta\mathcal{H}$ and the quality function $\mathcal{H}$ itself (although the latter is not strictly necessary). This implementation is available open source from `GitHub`[1] and `PyPi`[2].

```
function Louvain(Graph G)
    σ_i ← i.                                   ▷ Initial partition
    σ' ← MoveNodes(G)                          ▷ Initial move nodes
    while H(σ') > H(σ) do
        σ ← σ'
        G ← Aggregate(G, σ)
        Σ ← MoveNodes(G)                       ▷ Move nodes
        σ'_i ← Σ_{σ'_i} for all i              ▷ Correct σ' according to Σ
    end while
    return σ'
end function

function MoveNodes(Graph G)
    σ_i ← i for i = 1, ..., |V(G)|.            ▷ Initial partition
    q ← −∞
    while H(σ) > q do
        q = H(σ)
        for random v ∈ V(G) do
            c ← SelectCommunity(v)
            if ΔH(σ_v ↦ c) > 0 then
                σ_v ← c.
            end if
        end for
    end while
end function

function Aggregate(Graph G, Partition σ)
    A ← Adjacency(G)
    A'_cd ← Σ_ij A_ij δ(σ_i, c)δ(σ_j, d)
    return A'
end function
```

ALGORITHM 1. Louvain method. The algorithm loops over all nodes and moves nodes to alternative communities. When no more improvement can be made, it aggregates the graph and reiterates the procedure until no more improvement can be made.

```
function SelectCommunity(Node v)
    δ ← −∞.
    c ← σ_v.
    C ← {σ_u | (uv) ∈ E(G)}                    ▷ Neighbour communities.
    for Community c' ∈ C do
        if ΔH(σ_v ↦ c') > δ then
            δ ← ΔH(σ_v ↦ c')
            c ← c'
        end if
    end for
    return c
end function
```

ALGORITHM 2. Selecting the best community among its neighbours.

[1] https://github.com/vtraag/louvain-igraph
[2] https://pypi.python.org/pypi/louvain

function SELECTCOMMUNITY(Node v)
  return random $\sigma \in \{\sigma_u \mid (uv) \in E(G)\}$.
end function

ALGORITHM 3. Selecting a random community of its neighbours.

III. IMPROVEMENT

Not surprisingly, the Louvain algorithm generally spends most of its time contemplating alternative communities. While profiling our implementation, we found that it spends roughly 95% of the time calculating the difference $\Delta\mathcal{H}(\sigma_v \mapsto c)$ in Algorithm 2. Much of this time is spent moving around nodes for the first time. With an initial partition where each node is in its own community, almost any neighbouring community would be an improvement. Moreover, when the algorithm has progressed a bit, many neighbours are likely to belong to the same community. We therefore suggest that instead of considering all neighbouring communities, we simply select a random neighbour, and consider that community (as displayed in Algorithm 3), which we call the random neighbour Louvain.

There are several advantages to the selection of a random neighbour. First of all, it is likely to choose a relatively "good" community. In general, a node should be in a community to which relatively many of its neighbours belong as well (although this of course depends on the exact quality function). By selecting a community from among its neighbours, there is a good chance that a relatively good community is picked. In particular, if node i has $k_i(c)$ neighbours in community c, the probability that community c will be considered for moving is $k_i(c)/k_i$. The probability for selecting a community is thus proportional to the number of neighbours in that community. Bad communities (with relatively few neighbours) are less frequently sampled, so that the algorithm focuses more on the promising communities (those with relatively many neighbours).

Moreover, when considering the initial partition of each node in its own community, almost any move would improve the quality function $\mathcal{H}$. The difference between alternative communities in this early stage is likely to be marginal. Any move that puts two nodes in the same community is likely to be better than a node in its own community. Such moves quickly reduce the number of communities from roughly n to $n/2$. But instead of considering every neighbouring community as in the original Louvain algorithm, which takes roughly $\mathcal{O}(\langle k\rangle)$, our random neighbour Louvain algorithm only considers a single random neighbour, which takes constant time $\mathcal{O}(1)$. So, for the first few iterations, Louvain runs in $\mathcal{O}(n\langle k\rangle) = \mathcal{O}(m)$, whereas selecting a random neighbour runs in $\mathcal{O}(n)$.

Notice there is a big difference between (1) selecting a random neighbour and then its community and (2) selecting a random community from among the neighbouring communities. The first method selects a community proportional to the number of neighbours that are in that community, while the second method selects a community uniformly from the set of neighbouring communities. Consider for example a node that is connected to two communities, and has $k_i - 1$ neighbours in the first community and only 1 in the other community. When selecting a community of a random neighbour, the probability the good community is considered is $1 - \frac{1}{k_i}$, while the probability is only $\frac{1}{2}$ when selecting a random community.

Secondly, random selection of a neighbour increases the likelihood of quick convergence. The probability that node i is selected as a random neighbour is roughly $k_i/2m$, resembling preferential attachment [27] in a certain sense. Hubs are thus more likely to be chosen as a candidate community. Since, hubs connect many vertices, there is a considerable probability that two nodes consider the same hub. If these two (or more) nodes (and the hub) should in fact belong to the same community, chances are high both nodes and the hub quickly end up in the same community.

As an illustration of this advantage, consider a hubs-and-spokes structure, with one central hub and only neighbouring spokes that are connected to each other (and always to the hub). So, any spoke node i is connected to nodes $i-1$ and $i+1$ and to the central hub, node n. Consider for simplicity that the nodes are considered in order and that every move will be advantageous. The probability that the first node will move to community n is $p_1 = \frac{1}{3}$. For the second node, he will move to community n if he chooses node n immediately (which happens with probability $\frac{1}{3}$), or if he chooses node 1, and node 1 moved to community n, so that $p_2 = \frac{1}{3} + p_1\frac{1}{3}$. Similarly, for the other nodes $p_i = \frac{1}{3} + p_{i-1}\frac{1}{3} = \sum_{j=1}^{i}\left(\frac{1}{3}\right)^j$ which goes to $\frac{1}{2}$ for $n \to \infty$. This is higher than when just considering a random neighbour community. In that case, the probability the first node will move to community n is still $\frac{1}{3}$. But for the second node, if node 1 moved to community n, only two communities are left: n and 3. In that case, community n is chosen with probability $\frac{1}{2}$. If node 1 didn't move to community n, then node 2 will move to community n with probability $\frac{1}{3}$. In general, node i moves to community n with probability $p_i = p_{i-1}\frac{1}{2} + (1-p_{i-1})\frac{1}{3} = p_{i-1}\frac{1}{6} + \frac{1}{3}$. Working out the recurrence, we obtain that $p_i = \frac{1}{3}\sum_{j=0}^{i-1}\left(\frac{1}{6}\right)^j$, which tends to $\frac{2}{5}$. Selecting a community of a random neighbour thus works better than selecting a random community from among the neighbours. Selecting a community of a random node is even worse. In that case, the probability is $p_i = \frac{1}{n}(1 + \frac{1}{n})^{i-1}$ which tends to 0 for $n \to \infty$. In short, selecting the community of a random neighbour is likely to choose a new community that will also be chosen by other nodes.

In summary, selecting a random neighbour should work well because of two reasons. First, it tends to focus on communities that are "good". Secondly, it

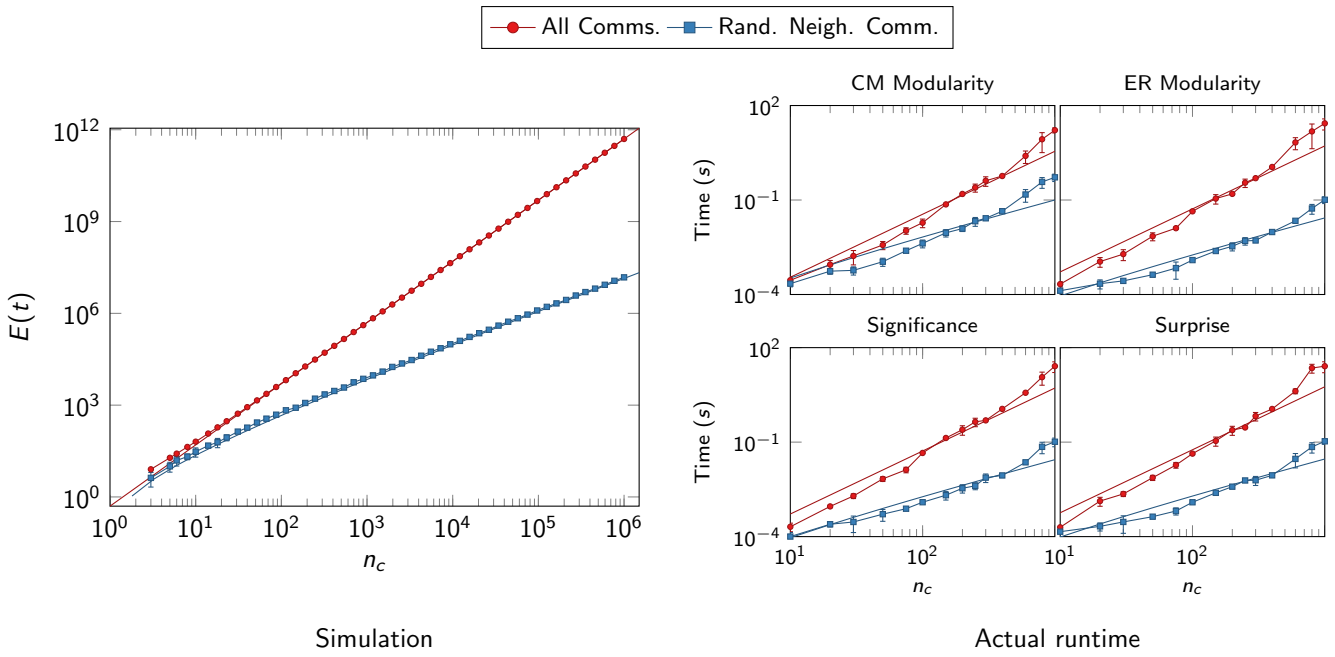

FIG. 1. **Clique.** The original Louvain algorithm considers all communities, which leads to $E(t) = \mathcal{O}(n_c^2)$ operations for putting all n_c nodes of a clique in a single community. The improvement considers only random neighbours, which takes only $E(t) = \mathcal{O}(n_c \log n_c)$ operations before the whole clique is identified as a single community. Left shows the number of operations in a simulation, with the markers indicating the simulated number of operations, and the solid lines the analytically derived estimates. Right shows the actual time used when optimizing the indicated quality functions for a clique. The solid lines denote best fits to n_c^2 and $n_c \log n_c$ in log-space.

should help in convergence because of higher likelihood of selecting hubs. In particular, the evaluation of SelectCommunity in the random neighbour Louvain takes a constant time $\mathcal{O}(1)$ whereas evaluating all communities takes about $O(\langle k \rangle)$. However, one essential question is whether SelectCommunity will not be too frequently evaluated in the random neighbour Louvain to counter this benefit.

To study this question, let us consider a ring of r cliques of n_c nodes each. The cliques (which are complete subgraphs containing $\binom{n_c}{2}$ links) are connected to another clique only by a single link in a circular fashion (i.e. clique i is connected only to clique $i-1$ and $i+1$). Most methods will tend to find the cliques (or sets of multiple cliques due to the resolution limit [19, 28]). Indeed, it is one of the best possible community structures: we cannot add any more internal edges, nor can we delete any external edges without disconnecting the graph. However, for the runtime complexity, the external edges will play only a marginal role. We may therefore simply assume we will work with r disconnected cliques of size n_c. Although the actual runtime will deviate from this, it should provide a reasonable runtime for relatively "clear" communities, and as such provide a lower bound for more difficult communities.

The core question is thus how quickly both variants run on cliques. We will assume the clique should become a single community, which is likely to be the case for most methods. Additionally, we assume $\Delta\mathcal{H} > 0$ only if a node is moved to a larger community, which is likely to be the case for most methods as nodes in a clique have more links to larger communities. The complexity of the original Louvain implementation is simple to evaluate in this case. The first node will be moved to one of its neighbours, an operation that costs n_c evaluations. The second node has only $n_c - 1$ evaluations to make, since the community of the first node disappeared. If we continue in this fashion, the total number of evaluations t is then $\sum_{i=1}^{n_c} n_c - i + 1 = \frac{n_c(n_c+1)}{2} = \mathcal{O}(n_c^2)$. The analysis of the expected runtime of the improvement is more difficult (see Appendix A for more details). However, we can provide a lower bound that serves as a rough estimate. Let us again denote by t the total number of operations before the whole clique is identified as a single community. We divide this in different phases of the algorithm, where each phase i runs from the time where there are $n_c - i + 1$ communities, until there are $n_c - i$ communities. In phase 1 we thus start out with n_c communities, and in the next phase there are only $n_c - 1$ communities. If we denote by t_i the number of operation in phase i, then by linearity $E(t) = E(\sum_t t_i) = \sum_t E(t_i)$. Notice that we will only leave phase i whenever a community of size 1 disappears. The probability that a community of 1 disappears is $\frac{n_c-1}{n_c}$, since it will join any other commu-

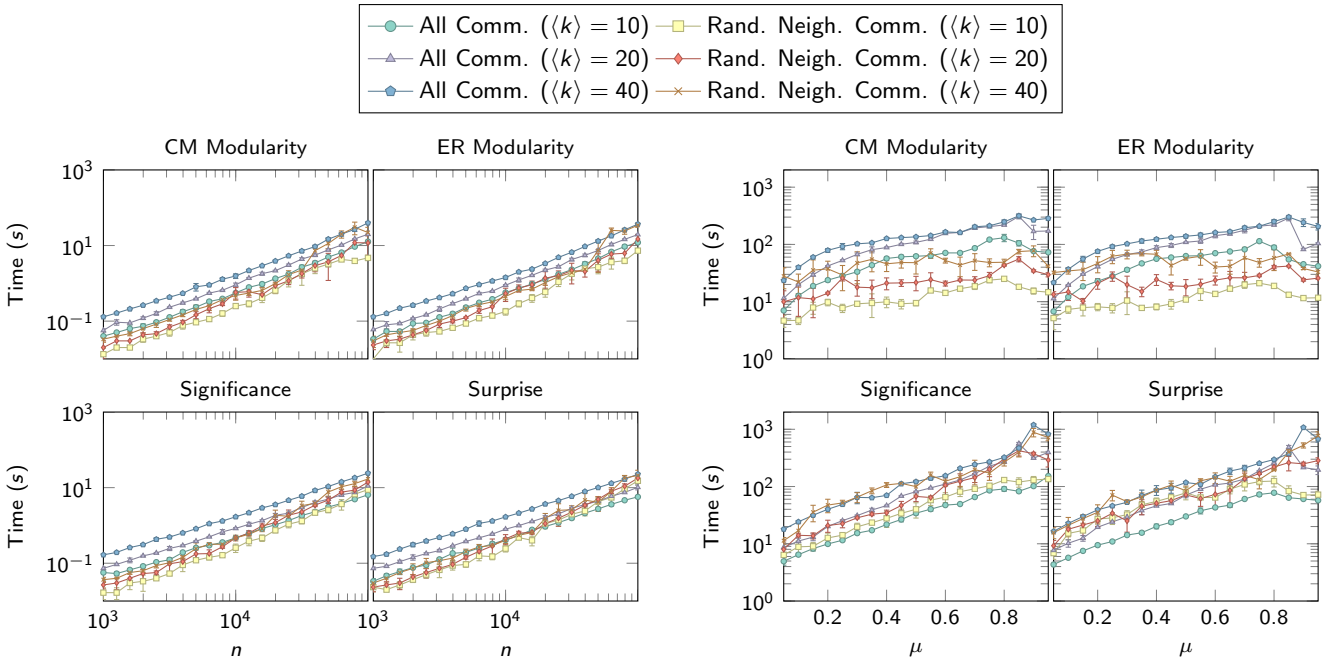

FIG. 2. **Speed.** The random neighbour Louvain algorithm is 2–3 times faster than the original version and at some points event faster. In addition, the running time scales less with increase degree, and is suggested to scale with $\log\langle k\rangle$, whereas the original Louvain algorithm scales with $\langle k\rangle$. The results for different n use $\mu = 0.1$, while the results for different μ use $n = 10^5$.

nity (except itself). There are at most i communities of size 1 in phase i, so that the probability a community of size 1 is selected is bounded above by $\frac{n_c-i+1}{n_c}$. In fact, such a state is also relatively likely, as the community size distribution tends to become more skewed than a more uniform distribution due to the preferential attachment on the basis of the community sizes. The number of expected operations in phase i is then bounded below by $\frac{n_c}{n_c-i+1}$, and the expected operations in total is bounded below by

$$E(t) \geq n_c \sum_{i=1}^{n_c} \frac{1}{n_c - i + 1} \tag{1}$$

$$= n_c \sum_{i=1}^{n_c} \frac{1}{i} = \mathcal{O}(n_c \log n_c) \tag{2}$$

However, this lower bound gives in fact a very accurate estimate of the expected running time, as can be seen in Fig. 1. Surprisingly then, whereas the original Louvain implementation runs in $\mathcal{O}(n_c^2)$, the improved version only uses $\mathcal{O}(n_c \log n_c)$ to put all nodes of a clique in a single community. We used an explicit simulation of this process to validate our theoretical analysis. Nonetheless, running the actual algorithms on cliques yields similar results (Fig. 1).

To get a rough idea of the overall running time, let us translate these results back to the ring of cliques. In that case, we have r cliques of n_c nodes. The runtime for the original Louvain method is $\mathcal{O}(n_c^2)$ for each clique, so that the total runtime is about $\mathcal{O}(rn_c^2)$. One factor of n_c^2 comes from running over n_c nodes, while the other factor comes from running over $\langle k\rangle \approx n_c$ neighbours. Since $rn_c = n$, and $n\langle k\rangle = m$, we thus obtain an overall running time of Louvain of about $\mathcal{O}(rn_c^2) = \mathcal{O}(n\langle k\rangle) = \mathcal{O}(m)$, similar to earlier estimates [10, 17]. Following the same idea, we obtain an estimate of roughly $\mathcal{O}(n \log\langle k\rangle)$ for the runtime of the random neighbour Louvain algorithm. So, whereas the original algorithm runs in roughly linear time with respect to the number of edges, the random neighbour algorithm runs in linear time with respect to the number of nodes. This is quite an improvement for such a small adjustment.

We use LFR benchmarks [29] to show that the random neighbour improvement also reduces the runtime drastically in more realistic networks. These benchmarks contain a planted partition, which we then try to uncover using both the original and the random neighbour Louvain algorithm. We tested networks of size ranging from $n = 10^3$ to $n = 10^5$, with average degrees $\langle k\rangle = 10$, 20 and 40. The community sizes and the degree both follow powerlaw distributions with exponents 1 and 2 respectively. The maximum degree was set at $2.5\langle k\rangle$, while the minimum community size was set at $\langle k\rangle$ and the maximum community size at $3\langle k\rangle$. An essential role is played by the probability that a link falls outside of the planted community μ. For low μ it is thus quite easy to identify communities, while for high μ it becomes increasingly more difficult. We used a 32-core machine with 64-bits

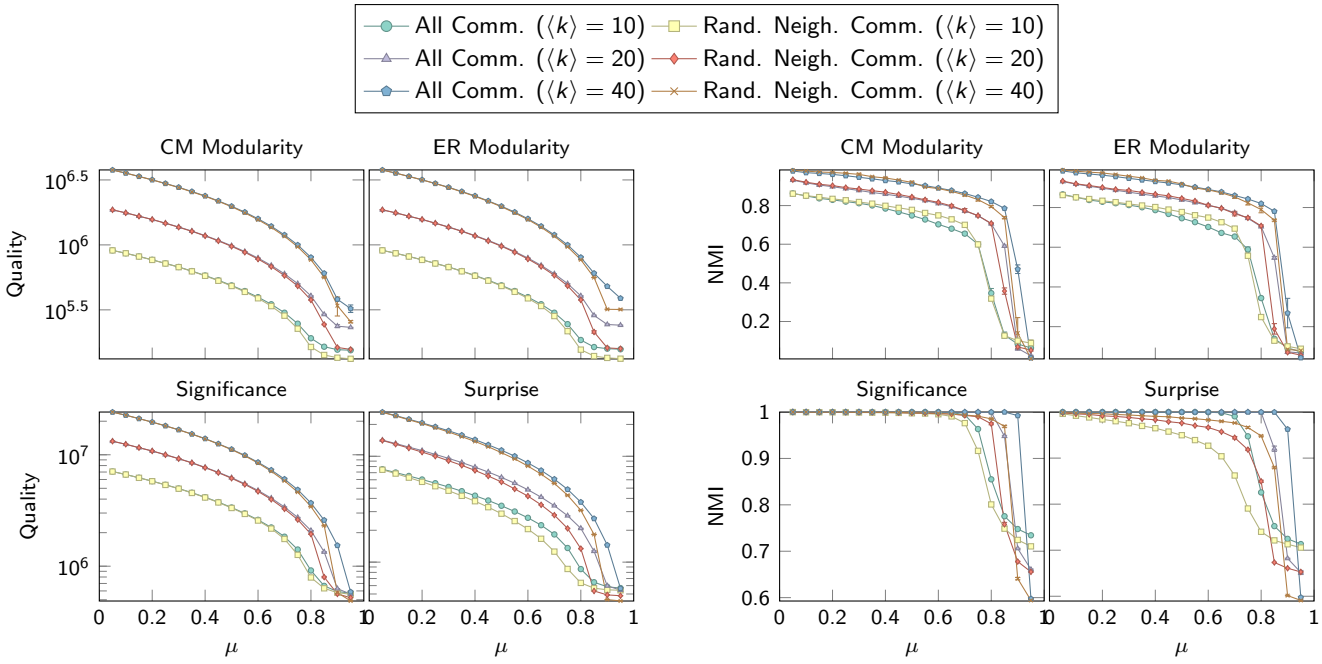

FIG. 3. **Quality.** The random neighbour Louvain does almost as well as the original version for a large part. Especially for Modularity and Significance the difference is largely negligible. Surprise is somewhat more affected by the random neighbour selection. We show results for $n = 10^5$.

Intel Xeon E5-2650 CPUs at a clock speed of 2 GHz with 64 GB of internal memory for testing. The time reported refers to used CPU time in seconds, not elapsed real time. The Louvain algorithm can be applied to many different methods, and we here show results for (1) Modularity using a configuration null model [11] (CM Modularity); (2) Modularity using an Erdös-Rényi null model [13] (ER Modularity); (3) Significance [25]; and (4) Surprise [30]. The speed of the original Louvain algorithm and the random neighbour Louvain algorithm for all four methods is reported in Fig. 2, while the quality of the detected partition is compared in Fig. 3.

For all these methods, the random neighbour Louvain speeds up the algorithm roughly 2–3 times. For larger networks, the improvement starts to deteriorate somewhat, especially for Significance and Surprise, and at some point the original Louvain algorithm is actually faster for those methods. In general, the improvement is larger for larger $\langle k\rangle$, consistent with our expectations. At the same time, the quality of the partitions found remain nearly the same for most methods. When using Surprise, the quality deteriorates more quickly however, especially for higher μ. So, for real networks it depends on the exact conditions which variant performs better. Surprise and Significance tend to find relatively smaller communities than Modularity, suggesting that the performance gain of using the random neighbour Louvain is especially pertinent when making a relatively coarse partition.

IV. CONCLUSION

Many networks seem to contain some community structure. Finding such communities is important across many different disciplines. One of the most used algorithms to optimize some quality function is the Louvain algorithm. We here showed how a simple adjustment leads to a substantial improvement in the runtime complexity. We argue that the approximate runtime of the original Louvain algorithm should be roughly $\mathcal{O}(m)$, while the improvement reduces the runtime to $\mathcal{O}(n \log\langle k\rangle)$ in a clear community structure. So, whereas the original algorithm is linear in the number of edges, the random neighbour algorithm is essentially linear in the number of nodes.

We have tested this improvement extensively. The improvement is quite consistent across various settings, sizes and methods. The runtime complexity was drastically reduced, speeding up the algorithm roughly 2–3 times, especially when concentrating on the coarser partitions found by Modularity. Nonetheless, it appeared that some methods, in this case Surprise, can be more sensitive to the selection of a random neighbour, with regard to the quality of the partition found. The current paper focused on the general optimization scheme, not on any particular objective function. Still, it would be interesting to understand better why random neighbour selection reduces the quality more for Surprise than for the other methods considered here. One possible solution

to address this issue is to first run the random neighbour Louvain algorithm, followed by a second run using the traditional Louvain algorithm, using the results from the first run as input for the second run.

The idea could also be applied in different settings. For example, the label propagation method is also a very fast algorithm [31], but it doesn't consider any objective function. It simply puts a node in the most frequent neighbouring community. But instead of considering every neighbour, it can simply choose a random neighbour, similar to the improvement here. We may thus expect a similar improvement in label propagation as for the Louvain algorithm. Similar improvements may be considered in other algorithms. The core of the idea is that a random neighbour is likely to be in a "good" community, which presumably also holds for other algorithms.

ACKNOWLEDGMENTS

This research is funded by the Royal Netherlands Academy of Arts and Sciences (KNAW) through its eHumanities project [3].

[1] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D.-U. Hwang, Phys. Rep. **424**, 175 (2006).
[2] D. J. Watts, Nature **445**, 489 (2007).
[3] D. Lazer, A. Pentland, L. Adamic, S. Aral, A.-L. Barabási, D. Brewer, N. Christakis, N. Contractor, J. Fowler, M. Gutmann, T. Jebara, G. King, M. Macy, D. Roy, and M. V. Alstyne, Science **323**, 721 (2009).
[4] R. Guimerà, D. B. Stouffer, M. Sales-Pardo, E. A. Leicht, M. E. J. Newman, and L. A. N. Amaral, Ecology **91**, 2941 (2010).
[5] R. F. Betzel, A. Griffa, A. Avena-Koenigsberger, J. Goñi, J.-P. Thiran, P. Hagmann, and O. Sporns, Netw. Sci. **1**, 353 (2013).
[6] A.-L. Barabási, Science **325**, 412 (2009).
[7] D. J. Watts and S. H. Strogatz, Nature **393**, 440 (1998).
[8] E. Ravasz, A. L. Somera, D. A. Mongru, Z. N. Oltvai, and A.-L. Barabási, Science **297**, 1551 (2002).
[9] A. L. Traud, P. J. Mucha, and M. A. Porter, Physica A **391**, 4165 (2012).
[10] S. Fortunato, Phys. Rep. **486**, 75 (2010).
[11] M. E. J. Newman and M. Girvan, Phys. Rev. E **69**, 026113 (2004).
[12] J. Duch and A. Arenas, Phys. Rev. E **72**, 027104 (2005).
[13] J. Reichardt and S. Bornholdt, Phys. Rev. E **74**, 016110+ (2006).
[14] R. Guimerà and L. A. Nunes Amaral, Nature **433**, 895 (2005).
[15] M. E. J. Newman, Phys. Rev. E **74**, 036104+ (2006).
[16] A. Clauset, M. E. J. Newman, and C. Moore, Phys. Rev. E **70**, 066111 (2004).
[17] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, J. Stat. Mech. **2008**, P10008 (2008).
[18] A. Lancichinetti and S. Fortunato, Phys. Rev. E **80**, 056117 (2009).
[19] V. A. Traag, P. Van Dooren, and Y. Nesterov, Phys. Rev. E **84**, 016114 (2011).
[20] M. Rosvall and C. T. Bergstrom, PLoS ONE **5**, e8694 (2010).
[21] M. Rosvall and C. T. Bergstrom, PLoS ONE **6**, e18209 (2011).
[22] P. Ronhovde and Z. Nussinov, Phys. Rev. E **81**, 046114 (2010).
[23] T. S. Evans and R. Lambiotte, Phys. Rev. E **80**, 016105 (2009).
[24] A. Lancichinetti, F. Radicchi, J. J. Ramasco, and S. Fortunato, PLoS ONE **6**, e18961 (2011).
[25] V. A. Traag, G. Krings, and P. Van Dooren, Sci. Rep. **3**, 2930 (2013).
[26] V. A. Traag, R. Aldecoa, and J.-C. Delvenne, arXiv:1503.00445 [physics, stat] (2015).
[27] A.-L. Barabási and R. Albert, Science **286**, 509 (1999).
[28] S. Fortunato and M. Barthélemy, Proc. Natl. Acad. Sci. USA **104**, 36 (2007).
[29] A. Lancichinetti, S. Fortunato, and F. Radicchi, Phys. Rev. E **78**, 046110 (2008).
[30] R. Aldecoa and I. Marín, Sci. Rep. **3**, 1060 (2013).
[31] U. N. Raghavan, R. Albert, and S. Kumara, Phys. Rev. E **76**, 036106 (2007).
[32] R. P. Stanley, *Enumerative Combinatorics: Volume 1*, 2nd ed. (Cambridge University Press, Cambridge, NY, 2011).

[3] http://www.ehumanities.nl/computational-humanities/elite-network-shifts/

Appendix A: Complexity in a clique

We here aim to determine the expected number of moves in the random neighbour algorithm. We assume it is always beneficial to move a node to a larger community. In other words, whenever we select a random node i, and a random neighbour j, and the community σ_j of the random neighbour is larger than the community σ_i of node i, i.e. if $|V_{\sigma_j}| \geq |V_{\sigma_i}|$, we will move the node.

Let us denote by f_k the number of communities that have size k

$$f_k = |\{c \mid |V_c| = k\}|. \tag{A1}$$

Then $g_k = kf_k$ denotes the number of nodes that belong to a community of size k. Additionally, define $F_k = \sum_{i=k}^{n} f_i$ the number of communities that have size k or larger. Similarly, define $G_k = \sum_{i=k}^{n} g_i$ the number of nodes in communities that have size k or larger. Clearly $\sum_k g_k = n$ so that $G_1 = n$. Also, $\sum_k f_k = r$ denotes the number of communities. The probability to select a node from a community of size k is then simply $\frac{g_k}{n}$. Let us denote by X_{cd} the event of moving a node from a community of size c to a community of size d. Then the probability of X_{cd} is

$$\Pr(X_{cd}) = \begin{cases} \frac{g_c}{n}\frac{g_d}{n} & \text{if } c < d \\ \frac{g_c}{n}\frac{g_c - c}{n} & \text{if } c = d \\ 0 & \text{if } c > d \end{cases} \tag{A2}$$

The probability to move it from any community to any other is then $\sum_{cd} \Pr(X_{cd})$. Alternatively, it is easy to see that the probability to move to any other community is $\frac{G_k - k}{n}$ (where we subtract k to make sure it moves to another community, and not the same community). So, overall, the probability we will move a node is then

$$\Pr(\text{move}) = \sum_{k=1}^{n} \frac{g_k}{n} \frac{G_k - k}{n} \tag{A3}$$

Similarly, the probability we will not move a node to another community (i.e. we remain stuck in the same partition) is then

$$\Pr(\text{not move}) = \sum_{k=1}^{n} \frac{g_k}{n} \frac{n - G_k + k}{n}. \tag{A4}$$

We only reduce the number of communities if we move a node from a community of size 1 of course. Hence, the probability to reduce the number of communities by 1 is then

$$\Pr(\sum_i f_k = r - 1) = \frac{g_1}{n} \frac{G_k - 1}{n} \tag{A5}$$

Now it would be possible to construct a complete transition network from any of the partitions to other partitions. However, this becomes quickly intractable, and rather difficult to solve.

Instead, we suggest to group partitions by the number of communities. Then, we divide the process into different phases. The algorithm would be in phase i whenever there are $n-i+1$ communities. In other words, in the first phase there are n communities, and the next phase starts whenever one of these nodes is put in another community. In the penultimate phase, there are only two communities. Let us then denote by t_i the number of moves during phase i, and by t the total number of moves during the whole process. We would like to examine $E(t_i)$, which we can write out as $\sum_i E(t_i)$ by linearity of expectation.

The number of ways to partition a set of n nodes in r sets can be denoted by $p_r(n)$, which is known as the partition function in number theory [32]. This function obeys the recursive identity

$$p_r(n) = p_r(n - r) + p_{r-1}(n - 1), \tag{A6}$$

since there are $p_{r-1}(n-1)$ partitions with at least one community of size 1 and $p_r(n-r)$ partitions that all have a community of size at least 2 (since we put r nodes in each one of the r communities). We would then like to know how many partitions there are that have s communities of size 1. Let us first define $q_k(n)$ to denote the number of ways to partition n into k sets with at least one community of size 1. Secondly, we need its counterpart $u_k(n)$ which denotes the number of ways to partition n into k sets without any community of size 1. Obviously then $p_k(n) = q_k(n) + u_k(n)$. We can then derive the recursion

$$q_k(n) = \sum_{r=1}^{k-1} u_{k-r}(n - r) \tag{A7}$$

$$= \sum_{r=1}^{k-1} p_{k-r}(n - r) - q_{k-r}(n - r) \tag{A8}$$

The reasoning is as follows. If there are r communities of size 1, we should know how many partition there are of $n - r$ nodes into $k - r$ communities without using communities of size 1. Here $q_1(1) = 1$. More specifically, let us denote by $q_k(n, r)$ the number of partitions that have r communities of size 1 and in total k communities, using n nodes. Then, obviously, $q_k(n) = \sum_{r=1}^{k-1} q_k(n, r)$. Moreover, $q_k(n, r) = u_{k-r}(n - r) = p_{k-r}(n - r) - q_{k-r}(n - r)$. The average probability to reduce the number of communities by 1 is then

$$\Pr\left(\sum f_s = k - 1 \mid \sum f_s = k\right) \tag{A9}$$

$$= \sum_{r=1}^{k-1} \frac{q_k(n, r)}{p_k(n)} \frac{g_1}{n} \frac{G_1 - 1}{n} \tag{A10}$$

$$= \sum_{r=1}^{k-1} \frac{q_k(n, r)}{p_k(n)} \frac{n - r}{n} \frac{n - 1}{n} \tag{A11}$$

This expression is unfortunately not easy to evaluate analytically. Moreover, it incorrectly assumes that each partition is equally likely a priori, whereas we know that

a more uneven distribution of community sizes is more likely due to the preferential attachment to the largest community. However, the following upper bound is immediate

$$\Pr\left(\sum f_s = k-1 \mid \sum f_s = k\right) \leq \frac{n-k+1}{n}\frac{n-1}{n} \tag{A12}$$

In general $E(t_i)$ can be calculated relatively straightforward as

$$E(t_i) = \frac{1}{\Pr(\text{reduce community by 1})} \tag{A13}$$

which using the bound in Eq. (A12) leads to

$$E(t_i) \geq \frac{n}{n-k+1}\frac{n}{n-1} \tag{A14}$$

so that

$$E(t) = \sum_i E(t_i) \geq \sum_{i=1}^{n} \frac{n}{n-i+1}\frac{n}{n-1} \tag{A15}$$

$$\approx n\sum_{i=1}^{n}\frac{1}{i} \tag{A16}$$

$$\approx \mathcal{O}(n\log n) \tag{A17}$$