Multi-Source Second-Order Random Walks Generation on Graphs

Wiktor Jurasz
Technical University of Munich
wiktor.jurasz@tum.de

Thomas Neumann
Technical University of Munich
thomas.neumann@in.tum.de

ABSTRACT

Generating a large number of random walks is a necessary step for many machine learning tasks. However, such procedure requires long computations when preformed on large graphs.

Here we propose *msnode2vec* an improvement over *node2vec* [8]. We developed a new algorithm to faster generate seconorder random walks as defined in *node2vec*. Our algorithm can also be easily adjusted to any other task which requires large number of random walks. The solution is designed to run on single machine and be easily parallelizable.

We demonstrate that our algorithm outperforms *node2vec* walk generation procedure. We also perform detailed analysis of random walks characteristics and show that our solution does not decrease the quality of generated walks.

1. INTRODUCTION

Many machine learning tasks concerning graphs require to embed them into a vectorized representation. One example is labeling, where we try to assign a node to one or more groups depending on some metric [10]. Another popular challenge is link prediction, namely determining if two nodes should be connected by an edge [13].

One could try to manually come up with an appropriate feature set to embed a graph. However, capturing the whole complexity of a large graph structure might be a very perplexing challenge. Additionally, such approach does not generalize well for different graphs.

There exist several approaches that automatize this task. For example learning the representation by solving optimization problem [3] or by trying to preserve local neighbourhood of the graph nodes [15] [18].

However, random walks on graphs are useful not only for representation learning. One example might be extracting structural information from graph to generate new structures with similar properties [4].

The focus of this paper is optimizing the speed of the random walks generation. We take one of the most popular graph embedding methods, namely the *node2vec* algorithm [8], and we provide a new way of generating large quantity of the second-order random walks as defined in *node2vec* (detailed description in Section 3).

We propose a biased second-order random walks generation algorithm. Considering the notion of parametrized random walk defined in *node2vec* we develop a new algorithm with improved performance. Similarly to recent work on multi-source BFS [19] we exploit the fact that nodes are visited multiple times in the generation process and we decrease the amount of repeated steps.

The results of this work can also be consider independently from *node2vec* algorithm and applied to all cases where large number of random walks is necessary.

The rest of the paper is structured as follows. In Section 2, we discuss briefly the related work regarding improvements to node2vec and random walks generation. In Section 3, we give more detailed description regarding node2vec walk notion and generation approach. We also give some intuition on regarding our algorithm. In Section 4, we proceed to the actual algorithm description. In Section 5, there is more detailed analysis of the algorithm behaviour and experiments on various graphs. Finally in Section 6, we conclude and propose some areas to explore in the future.

2. RELATED WORK

Since *node2vec* paper publication there have been numerous extensions and improvements to the algorithm. Here we present and briefly describe some of those. We also mention some other recent approaches for representation learning.

J D Wilson et al. [20] found a way to extend second-order random walks embeddings for multi-layer networks. Furthermore, notable research was also conducted in order to efficiently use node2vec for dynamic graphs [14].

D Zhou et al. [21] significantly improved performance of node2vec in distributed setting, where message passing between within the computing cluster is a bottle neck. A D Sarma et al.,[7] proved that random walks can be generated in sub-linear time with high probability. Specifically, they achieved complexity of $O(\sqrt{lD})$ instead of $O(l)$ (where l is walk length and D is average node degree) by combining many short walks together.

Generalization of random walk based algorithms is also an active area. Frameworks that combine node2vec, DeepWalk and other random walks based approaches were recently proposed [1] [2]. [16] also showed that node2vec and other similar algorithms can be state as matrix factorization problem and have a closed form solution. Other researches diverge from the random walk notion, proposing different ways of examining the graph structures like utilizing the idea of *coreness* of the graphs [17] or approximating them by simpler structures[6].

3. BACKGROUND

As mentioned in Section 1, we focus on generating second-order random walks as defined in *node2vec*. Before we present

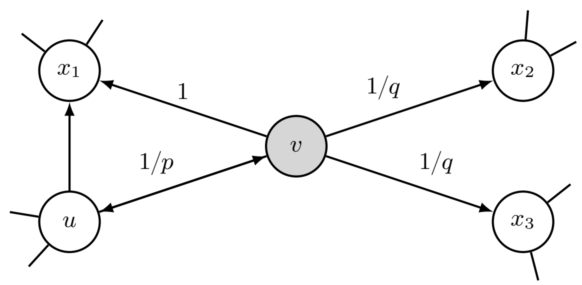

Figure 1: In this diagram random walk has just transitioned from u to v. Now, the next step from v to one of it's neighbours is considered. On the edges we see bias α.

our algorithm it is important to understand how *node2vec* works. It consists of two main steps:

1. Random walks generation
2. Features learning with *word2vec* algorithm using skip-gram approach

First, a number of walks of a given length are generated for every node in the graph. Next, the walks treated as sentences, are fed into the *word2vec* algorithm, which produces vector embeddings for each node.

The second part, i.e. features learning is irrelevant for our research, and is only mentioned here completeness. In following subsection we describe random walks generation on *node2vec*.

3.1 node2vec random walks generation

Let $G = (V, E)$ be a given graph. The second order random walk with first two steps $s_o = u$ and $s_1 = v$ is generated by following the probability distribution.

$$P(s_i = x | s_{i-1} = v, s_{i-2} = u) = \begin{cases} \frac{\pi_{uvx}}{Z} & : if\ (v,x) \in E \\ 0 & : else \end{cases}$$

Where π_{uvx} is the unnormalized transition probability between nodes v and x, given that previously the walk visited node u. Z is normalization constant.

Node2vec modifies this notion by introducing the search bias α that make the transition probabilities parameterized. The bias α depends on two previous nodes and is used in combination with edges weights to give unnormalized transition probabilities. Specifically $\pi_{uvx} = \alpha_{pq}(u, v) * w_{vx}$, where:

$$\alpha_{pq}(u, v) = \begin{cases} \frac{1}{p} & : if\ d(u,v) = 0 \\ 1 & : if\ d(u,v) = 1 \\ \frac{1}{q} & : if\ d(u,v) = 2 \end{cases}$$

Here $d(a, b)$ denotes distance between nodes a and b. Parameters p and q are algorithm meta-parameters defined by the user. In the Figure 1 we can see an example of a walk traversing part of a graph with bias α marked on edges.

The Algorithm 1 shows how the walks are generated in *node2vec*. This simple approach generates walks one by one r times for each node, where r is a meta-parameter defined by the user. For node2vec embeddings (and many other use cases) we need to generate significant amount of walks for every node. The consequence is that the algorithm will repeat exactly the same step many times.

Algorithm 1: Generation of random walks according to node2vec algorithm

```
Data: G = (V, E, π), length l, walks per node r
walks = []
for u in G.nodes() do
    for i = 0 to r do
        walk = [u];
        walk.append(G.getRandomNeighbour(u))
        for iter = 1 to l do
            curr = walk[-1]
            prev = walk[-2]
            s = aliaSample(prev, curr, π)
            walk.append(s)
        walks.append(walk)
return walks
```

This situation reassembles to some extend executing BFS algorithm from every node the graph. In 2014, an algorithm for multi-source BFS execution was developed [19]. By grouping multiple BFS procedures, when they reach the same node, the authors managed to significantly decrease the run time of multiple executions.

Steaming from this idea, we develop a new algorithm for generating large number of random walks. By grouping and reusing previous steps we decrease the run time over 2 times for C++ implementation and over 6 times for Python implementation.

3.2 Alias Method

Alias Sampling is a method used in *node2vec* to sample a next node in the random walk. The version introduced by Kronmal and Peterson in 1979 [22] allows to decrease the complexity of such operation from $O(d)$ to $O(1)$ where d is the degree of the node.

The drawback of this approach is its memory consumption. The method works by transforming multinomial distribution of transition probabilities in to the uniform distribution from which constant time sampling is possible. This method has $O(|V|d^2)$ memory complexity, where $|V|$ is the number of nodes and d is the average node degree.

4. ALGORITHM

Our solution is designed for single machine execution. This is reasonable even for huge graphs [9] with billions of nodes, since for random walks generation we only have to store nodes and edges weights which fit into 32 or 64 bit integers. There are three further core points on which we based our solution:

- Execute multiple walks concurrently reduces memory access overhead, as we are able to exploit memory locality when accessing the graph.
- By grouping multiple walks on a single edge we can reduce number of costly alias sampling operation.
- Whole procedure should avoid any type of synchronization and be easily parallelizable.

In this section we introduce the *msnode2vec* algorithm. In subsection 4.1 we describe the basic algorithm and give a clear overview of its advantages and disadvantages. We also point out which parts should be a subject of further research. In subsection 4.2 we present the improved version that leverages bit operations and better exploits memory locality. Finally in subsection 4.3 we present a version which further reduces run time at the cost of slightly biased walks.

4.1 msnode2vec

Algorithm 2: msnode2vec

```
   Data: G = (V, E, $\pi$), length l, walks per node r,
         concurrent walks c
1  walks = []
2  map<Pair<node>, walks> visit, visitNext = $\varnothing$
3  for startNodes in startNodesIterator(r,c) do
4    init(G, startNodes, r, visit)
5    while visit $\neq \varnothing$ do
6      while visit $\neq \varnothing$ do
7        prev, curr, walks = visit.pop()
8        for walk in walks do
9          step = aliasSample(prev, curr, $\pi$)
10         if walk.length < l then
11           visitNext[curr, step].append(walk + [step])
12         else
13           walks.append(walk + [step]))
14     swap(visit, visitNext)
15 return walks
```

The Algorithm 2 represents the basic version of Multi-Source Random Walk generation procedure. At the beginning (line 2) we initialize empty maps *visit* and *visitNext*. Those are the structures that group walks which are traversing the same edge at the same step. In line 3 we iterate thought *startNodes*, in this version of algorithm *startNodesIterator* will return appropriate number of nodes, such that total number of concurrent walks will not exceed *concurent walks* $\boldsymbol{c}$[1]. Next in line 4, we initialize the walks (detailed description below in Algorithm 3) and start the generation in lines 5-6. We get one pair of nodes, together with all the walks that are currently passing through them in line 7 and perform the update in lines 8-13. If we have not achieved required length yet, we move the walks to new node pairs in line 11, otherwise we append generated walk to result *walks* in line 13. In line 14 we swap *visit* with *visitNext* and start the next round.

Algorithm 3: init function for msnode2vec

```
1  Function init(G, walks, r, visit):
2    for node in = startNodes do
3      for i in range r do
4        nextNode = G.getRandomNeighbour(node)
5        visit[node, nextNode].append([node, nextNode])
```

The *init* function, described in Algorithm 3, is just a simple nested loop which initialize r walks for every node in *startNodes*. In line 4 we uniformly randomize the first step, as we cannot yet perform second order walk. After this function we have 2 nodes in every walk and we can perform the actual algorithm.

The algorithm in its simple form has two disadvantages.

- The alias sampling (line 9) is still done one by one for every node, and the only performance gain comes from memory locality. Additional performance gain can be achieved if more efficient sampling methods are used [5] (This task is beyond the scope of this research and is mentioned in section 6 as possible future extension). Additionally, as shown in Section 5, the number of nodes which traverse the same edge at given time is on average very small.
- Accessing the map and appending to *walks* list in lines 7,11 and 13, despite of having $O(1)$ complexity, are operation with high constant and turns out very costly if executed many times.

Despite some problems, the above algorithm is a solid base for further improvements. Below we show further development of this algorithm using more advanced techniques.

4.2 Bit Operations

To address the second issue of base algorithm, we leverage bit operations. This however, limits number of concurrent walks to the size of used bit field. Typically we would use 64bit long integer, but newer Intel CPU models support 256bit instructions and registers (or even 512bit with AVX-512 technology), which allows to execute even more walks simultaneously.

Algorithm 4: msnode2vec with bit operations

```
   Data: G = (V, E, $\pi$), length l, walks per node r,
         concurrent walks c
1  vector<int> walks[$G.nodes.length * r * l$] = []
2  int walkOffset = 0
3  vector<int> previousNodes, currentNodes = {0}
4  queue<int> visit, visitNext = $\varnothing$
5  for startNodes in startNodesIterator(r,c) do
6    init(G, walks, startNodes, r, visit, previousNodes,
       currentNodes, walkOffset)
7    vector<int> walksLengths
8    while visit $\neq \varnothing$ do
9      while visit $\neq \varnothing$ do
10       prev, curr = visit.pop()
11       indexes = previousNodes[prev] &
           currentNodes[curr]
12       previousNodes[prev] & = ~indexes
13       currentNodes[prev] & = ~indexes
14       while indexes > 0 do
15         index = findFirstSet(indexes)
16         nextNode = aliasSample(prev, curr, $\pi$)
17         if walksLengths[index] < l then
18           visitNext.push(node, nodeNext)
19           walksLengths[index]++
20           setBit(previousNodes[curr], index)
21           setBit(currentNodes[nextNode], index)
22         walks.putXY(walkOffset + index,
             walksLengths[index], nextNode)
23         clearBit(indexes, index)
24     swap(visit, visitNext)
25 return walks
```

[1]This type of iteration of course imposes some limitation on *concurent walks* **c** and *walks per node* **r**, i.e. $c \geqslant r$ and (for best performance) $c\%r = 0$. However, any type of iteration can be used, specifically one that is independent from relative size of **r** and **c** and always causes algorithm to perform **c** walks regardless of **r**. Here, this simplified iteration was used to make the pseudo code more concise and readable.

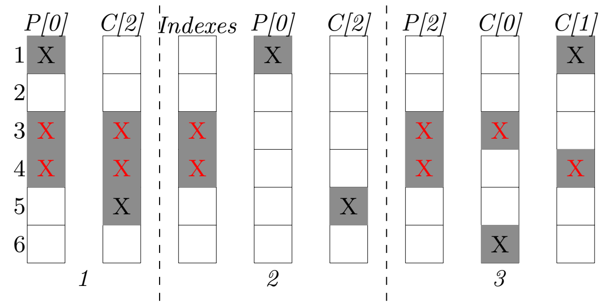

Figure 2: Example sequence of bit operations in *P* - *previousNodes*, *C* - *currentNodes* and *Indexes*. The symbol X indicates bit set to 1. 1) walks 3 and 4 are on the edge between nodes 0 and 2 (line 10). 2) Walks indexes are extracted and removed from *previousNodes* and *currentNodes* (lines 11-13). 3) New steps are assigned, walk 3 is now on the 2-0 edge and walk 4 on 2-1 (lines 25-26).

The algorithm starts with the initialization of *previousNodes* and *currentNodes* (line 3) to the vector of zeros. Those vectors will keep track of which walks are present in which nodes. To keep track of which nodes are currently visited, *visit* and *visitNext* queues are used (line 4). In line 4 and 5 we proceed in the same way as in base version (*init* function implementation is analogous and is ommited here for clarity and brevity). To get *indexes* of walks that are currently on the edge *prev* → *curr* we perform bit wise *and* operation (line 11), next we remove those walks using *and* with *negation* (line 12 and 13). Then, we iterate until there are no more walks to consider (line 14-23). In the loop we find index of first bit set to 1, which is also the index of the walk (line 15) and sample *nextNode* the same way as in base algorithm. Next, if we have not yet achieved required length, we push the new pair to be considered later and set bits for this walk in new nodes (lines 17 - 21). Finally we update the walk (line 22) and clear (line 23) previously found (line 15). Iteration continues until all walks reach the required length.

Because we want to keep the indexes of the walks in a fixed range (given by the size of the bit field) in line 2 we introduce *walkOffset*. This variable is then updated accordingly in *init* to point to the first walk in the *walks* array.

Figure 2 shows an example pass of the algorithm for bit field size 6. At some point in the line 10 we might get $prev = 0$ and $curr = 2$. It means that currently we are looking at the walks that traverse edge between node 0 and node 2. Next, we check index 0 in the *previousNodes* and index 2 in the *currentNodes*. Binary structure of values stored there reveals us that only the bits in positions 3 and 4 have value one set in both bit fields (part 1 in Figure 2). In the lines 11-14 we extract those *indexes* and remove them from *previousNodes* and *currentNodes* (part 2 in Figure 2). Finally, we iterate over indexes finding set bits (line 15) and removing them (line 23) after the walk proceeds. During every iteration we set new bits in *previousNodes* and *currentNodes* arrays. In our example, walk 3 proceeded to node 0 and walk 4 to node 1 (part 3 Figure 2). After this operation walk 3 has "0-2-0" suffix and walk 4 "0-2-1".

In contrast to the base algorithm, here we put updates directly into 2 dimensional grid instead of updating single vectors. This further reduces memory allocation overhead.

To reduce the memory consumption we used two vectors ($previousNodes$ and $currentNodes$) with the length $\#V$ instead of using single vector with length $\#E$. In majority of real-world graphs the number of edges is many times bigger then number of nodes, thus we in most cases: $2|V| \ll |E|$. Thus, this version requires $2 * |V| * |int|$ memory per execution thread, where $|V|$ is the number of nodes and $|int|$ is the size of the bit field.

4.3 Reusing Steps

Algorithm 5: msnode2vec with bit operations and reusing steps technique

```
Data: G = (V, E, π), length l, walks per node r,
      concurrent walks c, reuse prob rp
1  vector<int> walks[$G.nodes.length * r * l$] = []
2  int walkOffset = 0
3  vector<int> previousNodes, currentNodes = {0}
4  queue<int> visit, visitNext = ∅
5  vector<int> savedSteps = {−1}
6  for startNodes in startNodesIterator(r,c) do
7      init(G, walks, startNodes, r, visit, previousNodes,
        currentNodes, walkOffset)
8      vector<int> walksLengths
9      while visit ≠ ∅ do
10         while visit ≠ ∅ do
11             prev, curr = visit.pop()
12             indexes = previousNodes[prev] &
                currentNodes[curr]
13             previousNodes[prev] & = ∼indexes
14             currentNodes[prev] & = ∼indexes
15             while indexes > 0 do
16                 index = findFirstSet(indexes)
17                 if savedSteps[curr] ≠ -1 & getRand() < rp
                    then
18                     nextNode = savedSteps[curr]
19                 else
20                     nextNode = aliasSample(prev, curr, π)
21                     savedSteps[curr] = nextNode
22                 if walksLengths[index] < l then
23                     visitNext.push(node, nodeNext)
24                     walksLengths[index]++
25                     setBit(previousNodes[curr], index)
26                     setBit(currentNodes[nextNode], index)
27                 walks.putXY(walkOffset + index,
                    walksLengths[index], nextNode)
28                 clearBit(indexes, index)
29         swap(visit, visitNext)
30 return walks
```

As we show in Section 5, in many cases there are relatively few walks that end up in the same edge at the same step. This does not allow us to optimize calls to expensive *aliasSample* method. We also show that repetition of longer sequences of nodes between walks is very unlikely. However, repeating sequences of length 3 or 4 are much more common.

Steaming from this we introduce the *reuse probability* ***rp*** parameter. We allow the algorithm to reuse some of the previous steps, thus reducing number of calls to *aliasSample* method. This technique allows us to partially solve this issue.

We define one additional vector *savedSteps* which we initialize to -1. Each entry in this vector corresponds to one node in the graph. As the algorithm runs it saves some steps for future reuse (line 21). If there exist a saved step in current node, it will be used with some probability **rp** (lines 17 and 18). The rest of the algorithm stays the same as in the previous version. The memory requirements increases in comparison to previous version by one vector of size $|V|$.

It is also worth to mention that we can apply analogous technique in the Algorithm 2.

5. EXPERIMENTS

In this section we show how the walks are generated on different graphs. We compare performance, characteristics and show embeddings use-cases of our version vs the original node2vec.

All reported times correspond only to the amount of time that is needed in order to generate the random walks (i.e. we do not report transition probabilities and embedding learning times as they are not part of this research) and are generated by running the algorithms in single thread. Both algorithms do not use any synchronization methods and are easily scalable, thus for clarity only single thread is considered.

All the experiments were repeated several times and results were averaged. The machine used to run the experiments has the following specifications: *Intel(R) Core(TM) i9-7900X CPU @ 3.30GHz, 10 physical cores* and *128GB ram.*

5.1 Performance comparison

To compare performance of the algorithms we picked 4 different graphs.

- Facebook (*FB*) [12]: 4,039 Nodes represent users and 88,234 edges represent friendship relation between them. (Diameter of the graph: 8, average degree: 43.7)
- Twitter (*TW*) [12]: 81,306 Nodes represent profiles and 1,768,149 edges represent followers. (Diameter of the graph: 7, average degree: 43.5)
- Pennsylvania road network (*PRN*) [12]: 1,088,092 Nodes represent intersections or endpoints and 1,541,898 edges represent roads connecting them. (Diameter of the graph: 786, average degree: 2.83)
- Youtube social network (*YSN*) [12]: 1,134,890 Nodes represent users and 2,987,624 edges represent friendship relation between them. (Diameter of the graph: 20, average degree 5.3)

For all the graphs we executed 32 walks per node each of length 100. The *node2vec* parameters were set to $p = 0.5$ and $q = 2$. The C++ version was using bit operations (Algorithm 4 and 5), with bit field 64. The Python version was also executing 64 concurrent walks but without bit operations (Algorithm 2).

In Table 1 and Table 2 we report execution time for C++ and Python version respectively. Columns description:

- *n2v* - classical *node2vec*
- *mn2v* - multi-source node2vec - *msnode2vec*
- $rp = *$ - multi-source node2vec with *reuse probability* equal to $*$

Not surprisingly, for almost all the cases algorithm with the highest *rp* was the fastest as the number of expensive alias

Table 1: C++ generation times (seconds)

Graph	*n2v*	*mn2v*	*rp=0.2*	*rp=0.4*	*rp=0.6*	*rp=0.8*
FB	6.6	5.0	5.8	5.1	4.3	**3.6**
TW	245	223	223	182	146	**112**
PRN	1,473	**955**	1,264	1,182	1,105	1,001
YSN	3,426	3,006	2,999	2,588	2,317	**1,696**

Table 2: Python generation times (seconds)

Graph	*n2v*	*mn2v*	*rp=0.2*	*rp=0.4*	*rp=0.6*	*rp=0.8*
FB	57.7	22.8	36.0	33.4	27.4	**22**
TW	4,597	751	1,038	946	845	**698**
PRN	9,249	5,722	7,402	6,547	5,653	**4,338**

sampling operations was minimized. Worth to notice is that if the *rp* is small (i.e. 0.2) then the algorithm without reuse is better. This is because the additional randomization introduced in this version of algorithm (Algorithm 5, line 17) is executed for every iteration and for small *rp* the performance gain from *savedStep* is too low to account for that.

For Pennsylvania road network (PRN) the fastest algorithm in C++ version was *msnode2vec* without reusing steps. This is because the average dergee of a node in this graph is very low, thus in most of the cases alias sampling operation was not as costly as for the other graphs.

On the Figure 3 we can see that increasing the *rp* parameter improves the run time of the algorithm. For *TW* which is a *Small World Graph* (graph with low diameter) this technique speeds up computation 2.5 times between $rp = 0.1$ and $rp = 0.9$. For *PRN* the computations are only 1.25 times faster because of the reasons mentioned above.

Our version is on average **1.9** times faster for C++ and **3.8** for Python. Worth noticing is that regardless of configuration and language, our version is always faster than the original algorithm.

5.2 Walks Characteristics

We report walks characteristics on Facebook graph. The behaviour of algorithm is very similar on other *Small World Graphs.* All the statistics are reported the same settings as in previous subsection, namely: 32 walks per node each of length 100. The *node2vec* parameters were set to $p = 0.5$ and $q = 2$.

5.2.1 Reuse Probability Effect

All the statistics are reported for 64 concurrent walks.

In Figure 4 we can see that over 50% of the edges were visited at least 60 times and over 93% at least 30 times. We also calculated that, on average we have around 71 visits per edge, which means that we preform the same operation (traversing the edge and choosing next step) 71 times. The *rp* parameter does not influence this behaviour (i.e. The chart in Figure 4 looks the same for all values of $rp \in [0, 0.8]$). It means that this improvement does not introduce any bias towards some edges, which is very important feature for correct graph structure interpretation.

Charts in the Figure 7 show fraction of visits that aggregated given number of nodes. As we can see without *reuse probability* almost no aggregation happened. *Reuse probability* slightly increases the aggregation, however still over

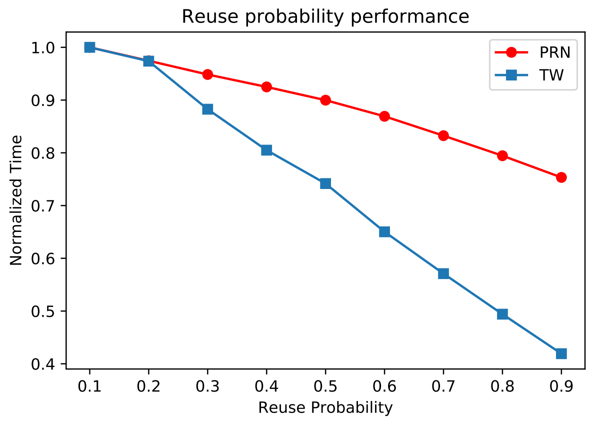

Figure 3: Algorithm performance gain with increasing Reuse Probability

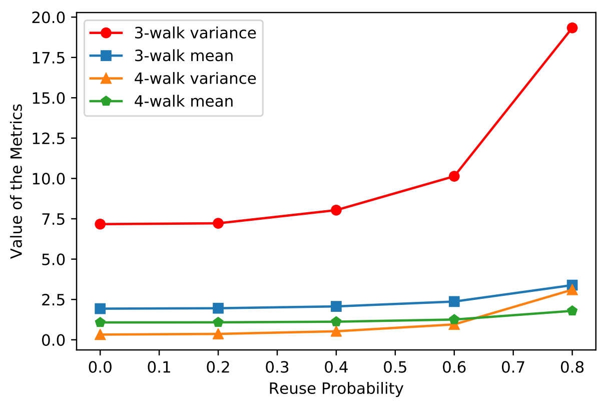

Figure 5: Mean and variance of sub-walks with length 3 and 4.

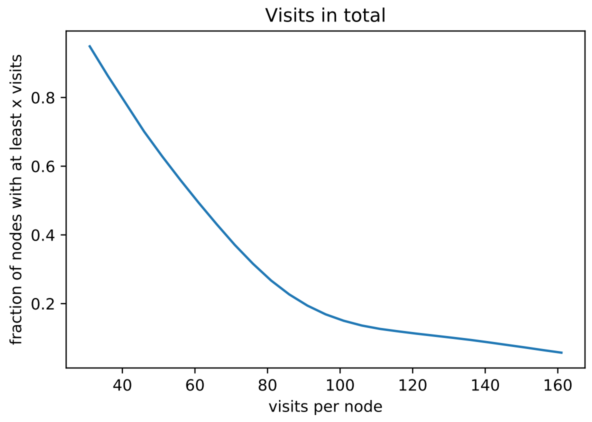

Figure 4: How many times a node is visited

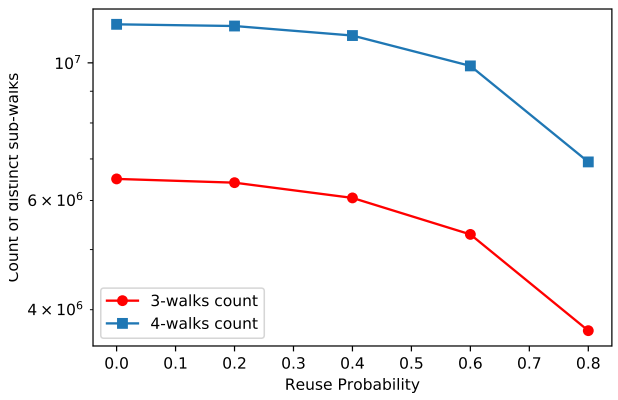

Figure 6: Count of distinct 3 and 4 sub-walks.

90% of visits aggregated less than 3 nodes which is not a lot considering 71 visits on average during whole run.

Nevertheless, it means r that we introduced slight bias and walks generated with higher rp may have more common sub-walks in the same positions.

We also analyzed how *reuse probability* influences short sub-walks (length 3 and 4) diversity. In the Figure 6 we see that the number of different 3 and 4 sub-walks remain very similar for $rp \in (0; 0.6)$ (within 20% range), the same goes for mean and variance (presented in Figure 5) of visits per unique 3 or 4 sub-walks. Only for $rp = 0.8$ those statistic diverges significantly, which means that for this value of rp quality of the walks suffers.

The sub-walks longer than 4 are very unlikely to repeat (for sub-walks of length 5 below 0.5% repetitions) and were not included in the graph.

General conclusion from above analysis, is that reusing previous steps up to the probability of 0.6 does not introduce high bias and still allows to maintain second-order characteristic of the walks. However, as shown in previous paragraph, it provides significant speed up in computations.

5.2.2 *Walks Aggregation*

All the statistics are reported for $rp = 0$.

On the Figure 8 we see reports on how many walks were aggregated on one edge at single step. For 16 or 512 concurrent walks we see only about in 5% of the cases there was more than one walk aggregated. For 131,072 (all walks were executed concurrently) the results are better, but still only 5% aggregated more than 4 walks. This statistic shows that aggregating per step is not enough and to leverage the fact that we visit one edge multiple times (71 times on average in this case) other methods have to be used, like proposed *reuse probability* (We mention different ideas in the Section 6).

5.3 Scalability

To test scalability we used Erdos-Renyi graphs with constant average node degree of 40. All the walks with *msnode2vec* were performed with $rp = 0.6$ and bit field of size 64. Both algorithms were generating walks with length 100.

We compared times on (Figure 9) different graph sizes with 32 number of walks per node and (Figure 10) graph with 16384 nodes and 327680 edges and variable number of walks per node.

We see that for small graphs and small number of walks

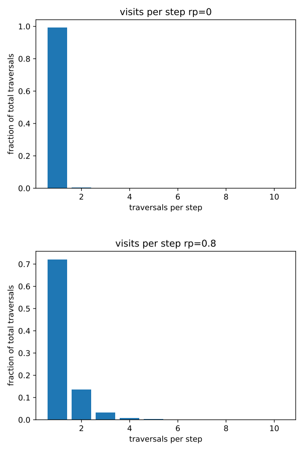

Figure 7: How many walks visit the same edge at the same time for $rp = [0, 0.4, 0.8]$

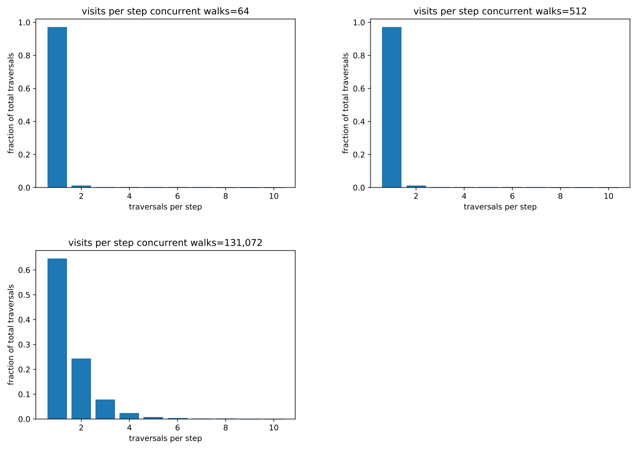

Figure 8: How many walks visit the same edge at the same time for 64, 512, 137,072 concurrent walks

both algorithms have similar run-time. However *msnode2vec* scales much better with graph size and walk number.

5.4 Clustering

Here we compare how *reuse probability (rp)* influences embeddings in the context of graph clustering. We consider the following two graphs:

- *Les Misérables* [11] - 77 Nodes represent the novel characters and 254 edges represent their interaction. We embed the graph in 16 dimensions, using 16 walks per node, 80 nodes per walk, $p = 1$ and $q = 0.5$ node2vec parameters.
- A small graph representing the interaction between the members of a *karate* club. We embed the graph in 32 dimensions using 16 walks per node, 80 nodes per walk, $p = 0.5$ and $q = 2$ node2vec parameters. It has 34 nodes and 78 edges.

We learn their embeddings using respectively *node2vec* and *msnode2vec*. For clustering we use *kmeans* algorithm.

We also report *Adjusted Rand Score* which is *Rand Score* [24] adjusted for chance [23]. In short, it is a way to compute similarity between two clusterings by considering all data points (in our case embedded nodes) pairs and counting pairs that are assigned in the same or different clusters in both clusterings. This metrics yields value close to 0 for random assignments and is equal to 1 if clusterings are identical.

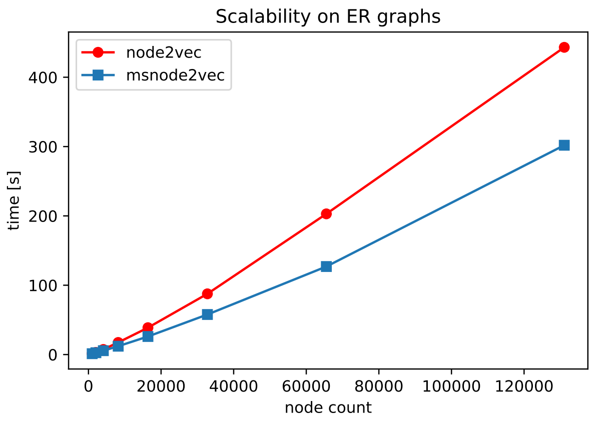

Figure 9: Scalability regarding graph size.

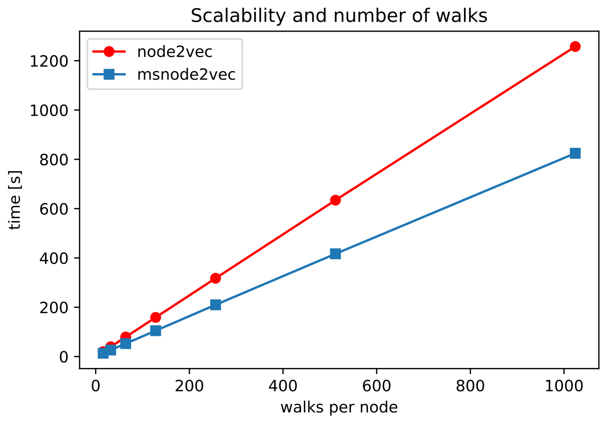

Figure 10: Scalability regarding number of concurrent walks.

5.4.1 *Les Misérables*

In the graphs (in Figure 11), we can clearly see 6 clusters. Both algorithms created walks that allowed to correctly embed the nodes. We can see a few differences in the assignments, however it is hard to judge which algorithm assigned the nodes better.

Running this experiment for $rp \in [0.1, ..., 0.8]$ yields similarly results for both algorithms, with average Adjusted Rand Index: 0.69.

5.4.2 *Karate*

In those graphs (in Figure 12) the clustering is not very obvious. With two relatively high degree hub nodes we tried 2 clusters. Above, the best obtained result is presented. Interestingly, despite not optimal clustering both algorithms gave the same results (making the same mistakes). Run-

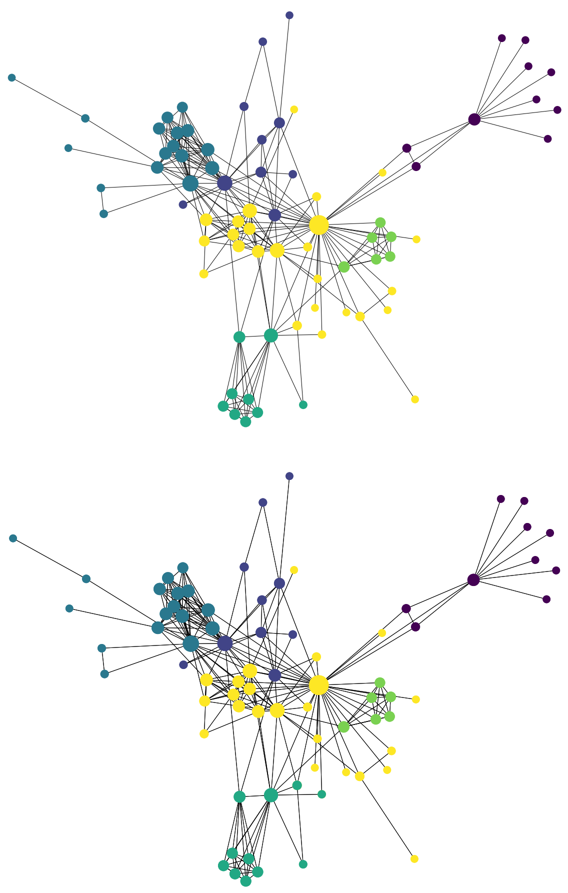

Figure 11: Top: clustering with base *node2vec*, Bottom: clustering with *msnode2vec* *rp=0.6*

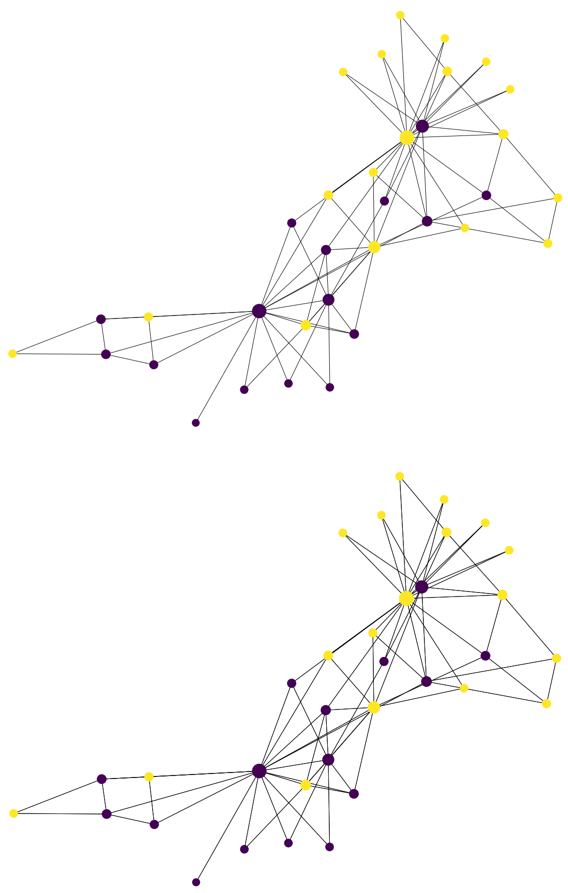

Figure 12: Top: clustering with base *node2vec*, Bottom: clustering with *msnode2vec* *rp=0.6*

ning this experiment for $rp \in [0.1, ..., 0.8]$ yields similarly results for both algorithms, with average Adjusted Rand Index: 0.96.

6. SUMMARY

In this paper we studies possible performance improvements for second order random walks generation as defined in *node2vec* [8]. We managed to develop new algorithm which for a price of slight bias in the generation process, improves performance by (a) more efficient memory access pattern and (b) decreasing number of redundant operations.

There are many interesting areas to explore in the future. As mentioned in Section 5.2.2 the aggregation per step does not provide significant overlap of walks. One could try to change the generation method from "walkcentric" to "nodecentric", i.e. sampling at node n at which position in which walk it will appear. This would allow to use more efficient sampling method [5].

One can also try to leverage some random walk properties like reversibility or Green Function in order to generate walks in even more efficient way.

Currently the algorithms use *Alias Sampling* method, which provides computations speedup, at the cost of high memory usage (Section 3.1). This makes the algorithm not suitable for very large graphs. For further scalability improvements the memory footprint has to be decreased.

7. REFERENCES

[1] Nesreen K Ahmed, Ryan A Rossi, Rong Zhou, John Boaz Lee, Xiangnan Kong, Theodore L Willke, and Hoda Eldardiry. A framework for generalizing graph-based representation learning methods. *arXiv preprint arXiv:1709.04596*, 2017.

[2] Nesreen K Ahmed, Ryan Rossi, John Boaz Lee, Theodore L Willke, Rong Zhou, Xiangnan Kong, and Hoda Eldardiry. Learning role-based graph embeddings. *arXiv preprint arXiv:1802.02896*, 2018.

[3] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[4] Aleksandar Bojchevski, Oleksandr Shchur, Daniel Zügner, and Stephan Günnemann. Netgan: Generating graphs via random walks. *arXiv preprint arXiv:1803.00816*, 2018.

[5] Karl Bringmann and Konstantinos Panagiotou. Efficient sampling methods for discrete distributions. In *International Colloquium on Automata, Languages, and Programming*, pages 133–144. Springer, 2012.

[6] Haochen Chen, Bryan Perozzi, Yifan Hu, and Steven Skiena. Harp: Hierarchical representation learning for networks. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[7] Atish Das Sarma, Danupon Nanongkai, and Gopal Pandurangan. Fast distributed random walks. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 161–170. ACM, 2009.

[8] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM, 2016.

[9] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. Wtf: The who to follow service at twitter. In *Proceedings of the 22nd international conference on World Wide Web*, pages 505–514. ACM, 2013.

[10] Ioannis Katakis, Grigorios Tsoumakas, and Ioannis Vlahavas. Multilabel text classification for automated tag suggestion. In *Proceedings of the ECML/PKDD*, volume 18, page 5, 2008.

[11] D. E. Knuth. *The Stanford GraphBase: a platform for combinatorial computing, volume 37*. Addison-Wesley Reading, 1993.

[12] J. Leskovec and A. Krevl. Snap datasets: Stanford large network dataset collection. June 2014. URL `http://snap.stanford.edu/data`.

[13] David Liben-Nowell and Jon Kleinberg. The link-prediction problem for social networks. *Journal of the American society for information science and technology*, 58(7):1019–1031, 2007.

[14] Sedigheh Mahdavi, Shima Khoshraftar, and Aijun An. dynnode2vec: Scalable dynamic network embedding. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 3762–3765. IEEE, 2018.

[15] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.

[16] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Kuansan Wang, and Jie Tang. Network embedding as matrix factorization: Unifying deepwalk, line, pte, and node2vec. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, pages 459–467. ACM, 2018.

[17] Soumya Sarkar, Aditya Bhagwat, and Animesh Mukherjee. Core2vec: A core-preserving feature learning framework for networks. In *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 487–490. IEEE, 2018.

[18] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *Proceedings of the 24th international conference on world wide web*, pages 1067–1077. International World Wide Web Conferences Steering Committee, 2015.

[19] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T Vo. The more the merrier: Efficient multi-source graph traversal. *Proceedings of the VLDB Endowment*, 8(4):449–460, 2014.

[20] James D Wilson, Melanie Baybay, Rishi Sankar, and Paul Stillman. Fast embedding of multilayer networks: An algorithm and application to group fmri. *arXiv preprint arXiv:1809.06437*, 2018.

[21] Dongyan Zhou, Songjie Niu, and Shimin Chen. Efficient graph computation for node2vec. *arXiv preprint arXiv:1805.00280*, 2018.

[22] Kronmal, Richard A., and Arthur V. Peterson On the Alias Method for Generating Random Variables from a Discrete Distribution. *The American Statistician, vol. 33, no. 4, 1979, pp. 214–218. JSTOR, www.jstor.org/stable/2683739*

[23] Hubert, L. and Arabie, P. Comparing Paritions Journal of Classification (1985) 2: 193. *https://doi.org/10.1007/BF01908075*

[24] Rand, William M. Objective Criteria for the Evaluation of Clustering Methods. Journal of the American Statistical Association, vol. 66, no. 336, 1971, pp. 846–850. JSTOR, *www.jstor.org/stable/2284239.*