An Efficient Algorithm for State Propagation on Graph with Lockable Vertices

Yuxiang Lin（林宇翔）

Paper for the project submitted to
2021 Virginia Tech D.C. Global STEM Expo

Keywords: Computer Science, Algorithm, Graph

An Efficient Algorithm for State Propagation on Graph with Lockable Vertices

Yuxiang Lin（林宇翔）

(ABSTRACT)

There is an interesting class of problems which requires maintaining a binary state carried by each vertex of a graph, where there also exists an operation where all *true* state vertices propagate simultaneously to their *false* state neighbors. And moreover, it is required to be able to lock a vertex from both inbound and outbound state propagation.

This paper will discuss an $O\left(|V| + q\sqrt{|E|}\right)$ algorithm for such problem, and explore its application.

Contents

Chapter 1

Introduction and Previous Work

This algorithm was inspired by a competitive programming problem [1] that has a background of modelling the transmission of virus, which is stated as such, given a **tree** with each vertex holding two boolean variables, s (e.g. **infected** or **not infected**) and *locked* (e.g. **under lockdown** or **not under lockdown**), and the following operations must be implemented:

set(**vertex** v) Let $v.s \leftarrow \mathit{true}$.

flip(**vertex** v) Flip $v.\mathit{locked}$.

propagate() For each pair of neighboring vertices v_0 and v_1, where $v_0.s = \mathit{true}$ and $v_0.\mathit{locked} = \mathit{false}$, and $v_1.s = \mathit{false}$ and $v_1.\mathit{locked} = \mathit{false}$, let $v_1.s \leftarrow \mathit{true}$.

This problem can be solved in $\Theta(|V| + q)$ time, where q is the total number of calls to all three functions. Since the process of communicating states is between two vertices, but the propagation operation is applied by one to another, it would be sensible to have the vertex initiating such operation iterate through as few neighbors as possible. Then consider the fact that for each vertex on a tree, a fixed parent can be appointed for it.

Let's refer to a vertex v where $v.s = \mathit{true}$ and $v.\mathit{locked} = \mathit{false}$ as a **source**, and a vertex where $v.s = \mathit{false}$ and $v.\mathit{locked} = \mathit{false}$ as a **target**.
So the outline of the algorithm is:

- Fix the root of the tree.
- Have each vertex becoming a **source** or a **target** add itself to a global queue.

[1] The contest that provided this problem was an internal contest between a few schools, and I was unable to recover the name of its author.

- Have each vertex becoming or no longer being a **source** increase or decrease a counter variable owned by its parent respectively.
- Have each vertex becoming a **target** add itself to a queue owned by its parent.
- During the propagation process, go through the global queue; for each **source** vertex, propagate to its parent and each vertex in its queue that is a **target**; and for each **target** vertex, check whether its parent is a **source** or its counter variable is a positive number, and update accordingly.
- Both the global and vertex-owned queue will be cleared afterwards.

Note that although the vertex that became a **source** or **target** can revert its state after its inclusion into the queues, the number of pop operation is bounded by the number of push operations in a queue.

Now it is important to know that the major difference between the problem stated above and the problem I aim to solve is that whereas a **tree** was given above, my algorithm will be applicable to any **graph**.

Chapter 2

The Algorithm

2.1 The Algorithm

Now let's extend the algorithm to any **undirected graph**, with set(**vertex** v, **bool** s') and setLock(**vertex** v, **bool** *locked'*) to be implemented. The parameters are a little different from the ones in the introduction, since the caller actually knows the assumed state of the program for a competitive programming algorithm, whereas this algorithm does not assume such.

To design an algorithm for any undirected graph, its most prominent issue would be the varying degrees of the vertices, since the naive algorithm will suffer a huge time penalty during a state update to a vertex with a large degree (**large** vertex for short). Recall the idea of having each vertex communicate with its parent when the graph is a tree, so it would be intuitive to appoint some kind of "parent equivalent" for a vertex on any graph. Also to note is that the number of large vertices is rather small due to its number being bounded by the degree of the graph. So let's define a critical constant for the graph *crit*, then for any vertex v with $v.degree > crit$, it would be considered a **large** vertex, in another word, a "parent equivalent" for all its neighbors (including other large neighbors). Thus for all set() and setLock() operations, the vertex acted upon shall notify all its large neighbors (even if the vertex itself is a large vertex). 2.1 gives a summary of how different vertices interact with their neighbors when being acted on.

	set() and setLock()	propagate()
small vertex	notify all large neighbors	iterate through all neighbors
large vertex	notify all large neighbors	iterate through queues of potential sources and targets

Table 2.1: Vertex interaction with neighbors when being acted on.

With these ideas, the algorithm is given below with the minimum total time complexity of $O\left(|V| + q\sqrt{|E|}\right)$ for $crit = \Theta(\sqrt{|E|})$. This technique of separating small items of large

quantity and large items of small quantity is similar to the meet-in-the-middle technique popularized by its application in the Baby-step Giant-step algorithm.[1]

At the beginning of the program, initialize() is called once with $v.adj$ being an array of vertices reached by edges of v, then any of the three functions can be called each time for a total of q times

Algorithm 1 Initialize

```
for each v do
    v.s ← false
    v.locked ← false

    for each v′ in v.adj do
        if v′.degree > crit then
            v.large.pushBack(v′)
            v′.targets.pushBack(v)
        end if
    end for
end for
```

Algorithm 2 Set

```
Require: vertex v, bool s′
if v.s = s′ ∨ v.locked then
    return
end if
v.s ← s′

if v.s then
    for each v′ in v.large do
        v′.sources ← v′.sources + 1
    end for
else
    for each v′ in v.large do
        v′.sources ← v′.sources − 1
        v′.targets.pushBack(v)
    end for
end if
queue.pushBack(v)
```

Lemma 1 *The time complexity of* set() *is* $O(\frac{|E|}{crit})$.

PROOF For each vertex v' in $v.large$, it must satisfy that $v'.degree > crit$.
Since the total degree of an undirected graph is $2|E|$, the number of large vertices is at most $\frac{2|E|}{crit}$.

Algorithm 3 setLock

```
Require: vertex v, bool locked'
  if v.locked = locked' then
      return
  end if
  v.locked ← v.locked'

  if v.locked then
      if v.s then
          for each v' in v.large do
              v'.sources ← v'.sources − 1
          end for
      end if
  else
      if v.s then
          for each v' in v.large do
              v'.sources ← v'.sources + 1
          end for
      else
          for each v' in v.large do
              v'.targets.pushBack(v)
          end for
      end if
      queue.pushBack(v)
  end if
```

Lemma 2 *The time complexity of* setLock() *is* $O(\frac{|E|}{crit})$.

PROOF This is proved the same as Lemma 1.

Algorithm 4 Propagate

```
Declare empty temporary queues temp and pending
swap(queue, temp)
for each v in temp do
    if v.locked then
        continue
    end if

    if v.s then
        if v.degree > crit then
            for each v′ in v.targets do
                pending.pushBack(v′)
            end for
            v.targets.clear()
        else
            for each v′ in v.adj do
                pending.pushBack(v′)
            end for
        end if
    else
        if v.degree > crit then
            if v.sources > 0 then
                pending.pushBack(v)
            end if
        else
            for each v′ in v.adj do
                if ¬v′.locked ∧ v′.s then
                    pending.pushBack(v)
                    break
                end if
            end for
        end if
    end if
end for

for each v in pending do
    set(v, true)
end for
```

Lemma 3 *The total amortized time complexity of* propagate() *is* $O\left(|V| + q \cdot \left(crit + \frac{|E|}{crit}\right)\right)$.

Proof First of all, $|V|$ targets are pushed into vertex-owned queue by init().

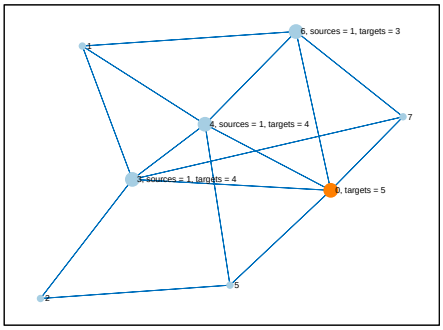
For each call to set() and setLock(), at most one vertex is pushed into the global queue.
For each call to setLock(), the current vertex might be pushed into $O(\frac{|E|}{crit})$ vertex-owned queues.
For each vertex v with $v.locked = false$ in the global queue, if it is a large vertex with $v.s = true$, then the run time of propagate() will bounded by the size of $v.targets$; and if it is a small vertex, it will go through all its neighbors, which there are, $O(crit)$ of them.

Corollary 1 *The total time complexity of the algorithm is* $O\left(|V| + q \cdot \left(crit + \frac{|E|}{crit}\right)\right)$.

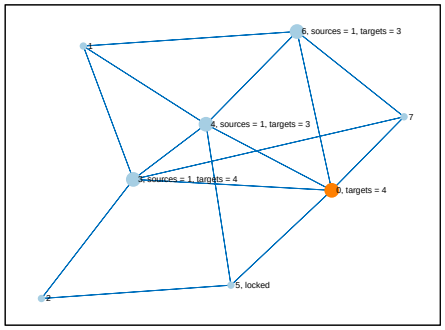
Theorem 1 *The minimum total time complexity of the algorithm is* $O\left(|V| + q\sqrt{|E|}\right)$ *for* $crit = \Theta(\sqrt{|E|})$.

Proof Easily proved by basic inequality $crit + \frac{|E|}{crit} \geq 2\sqrt{|E|}$, with $crit + \frac{|E|}{crit} = 2\sqrt{|E|}$ only when $crit = \sqrt{|E|}$.

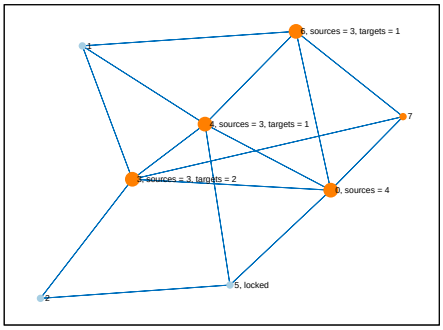
2.2 Example



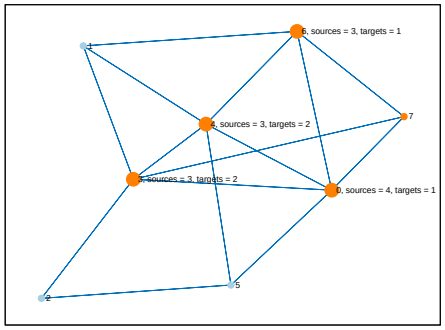
(a) set(0, *true*)



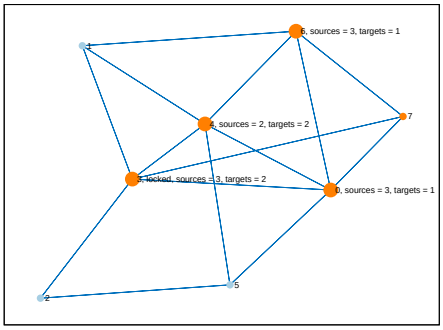
(b) setLock(5, *true*)



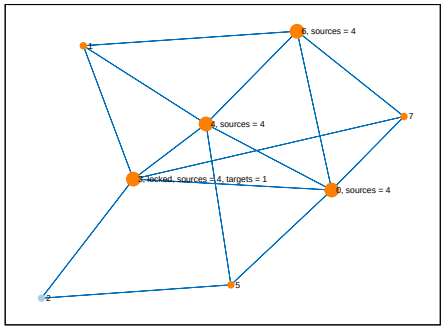
(c) propagate()



(d) setLock(5, *false*)



(e) setLock(3, *true*)



(f) propagate()

Figure 2.1: An example for $|V| = 8, |E| = 15, q = 6$, with the size of the vertex mark denoting whether it is a large vertex, and the color of light blue and orange for $v.s = 0$ and $v.s = 1$ respectively.

Chapter 3

Experiments

The following experiments are all done with C++ implementations compiled with Clang 11.0.1 (no optimization flags), and run on a PC with Intel(R) Core(TM) i7-8750H CPU @ 2.20 GHz (max turbo frequency 4.10 GHz) and 32026.3 MiB RAM, the operating system is Debian GNU/Linux bullseye/sid.

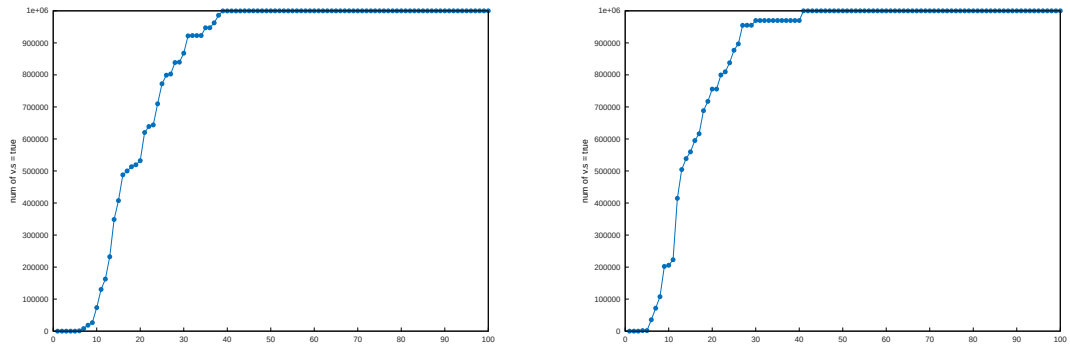
Two kinds of graphs are used for the experiment:

- The $G(n, m)$ **model** of the **Erdős–Rényi model**, which is a random graph out of all possible graphs with n vertices and m edges, it will be referred to as $G_{\text{rand}}(n, m)$ below.[2]
- The **Barabási–Albert model**, which generates a graph of n vertices by adding vertices incrementally, with each new vertex creating edges to m known vertices, and the old vertices' probability to connect to the new vertex is proportional to its degree, it will be referred to as $G_{\text{BA}}(n, m)$ below.[3]

The graphs generated are assumed to have no self-loops and duplicated edges. In implementation, a G_{rand} is generated by adding m edges with two random end vertices. And a G_{BA} is generated by initially choose m known vertices, and for each of the rest of the vertices it will create m random edges to known vertices, where the probabilty to connect to a known vertex v out of any known vertex i is: $p_v = \frac{w_v}{\sum_i w_i}$, then it will be added the known vertices; w_v is set to 1 for each initial vertex v, otherwise $w_v \leftarrow 0$, then w_v is incremented for the endpoints of each edge added.

The q operations generated are of random type, and for set() and setLock(), s' and *locked*$'$ are also random. What to note is that v will be random, but is of some discrete distribution specified below.

The two kinds of graphs I will use below will now be specifically $G_{\text{rand}}(10^6, 10^7)$ and $G_{\text{BA}}(10^6, 10)$, they were chosen for their moderately large size and sparseness, and they



(a) $G_{\text{rand}}(10^6, 10^7)$ (b) $G_{\text{BA}}(10^6, 10)$

Figure 3.1: Number of $v.s = true$ after executing q operations when all vertices are initially unlocked, tests done by repeating on 5 graphs and 5 sets of operations for each graph and averaging the results..

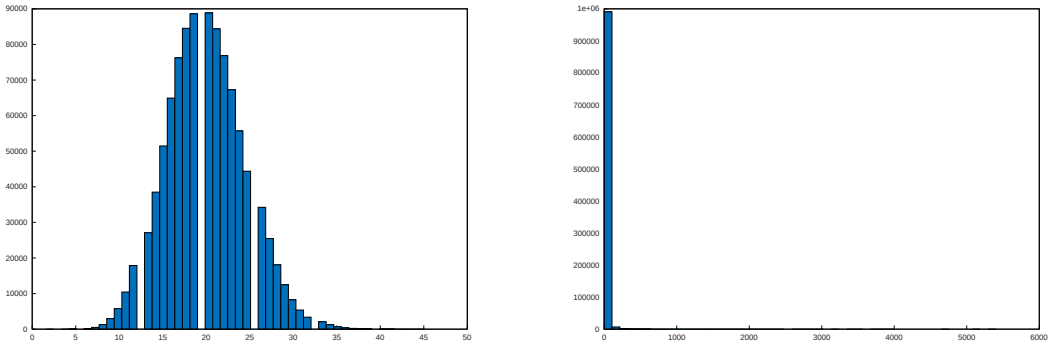
also have approximately same number of edges.

It was found that for a graph of either type, when all vertices are initially unlocked, it will just take very few (< 50) instructions for almost all vertices have $v.s = true$. So before the random operations, n setLock() operations will be generated, iterating through all vertices, and 0.1 of the operations will have $locked' = true$. These operations will not be counted in the q operations, so $n + q$ operations will be executed *de facto*. Note that this still meant a large amount of vertices will have $v.s = true$ after the first few instructions, but then the speed of propagation will slow down.

I first pit the algorithm mentioned in this paper (the "novel" algorithm) against the "naive" implementation. The naive algorithm is implemented by having each set() and setLock() function add the current vertex to the global queue, and propagate() function will go through all neighbors of the current vertex.

It was found that in order for the novel algorithm to perform better than the naive implementation, it will be necessary to access vertices of higher degree much more often. So the v generated in an operation will be of a discrete distribution that the probability where vertex v is chosen out of any vertex i is: $p_v = \frac{w_v}{\sum_i w_i}$. I attempted to set the weight to be equal to the degree of the vertex, it was also attempted to have it to equal to some measures of graph centralities: betweenness, PageRank, and lobby index, which may be of some legitimate applicational implications, but the results are all unsatisfying.[4][5][6]

But I found that for $G_{\text{BA}}(10^6, 10)$, when setting $w_v = deg_v^a$, for some small $a \geq 3$, the novel algorithm will run faster. It was also tried to set $w_v = a^{deg_v}$, but then a unreasonably large weight would be given to too few vertices, and the entire process will just focus on them. It was as expected that the algorithm didn't run faster on $G_{\text{rand}}(10^6, 10^7)$ even when vertices



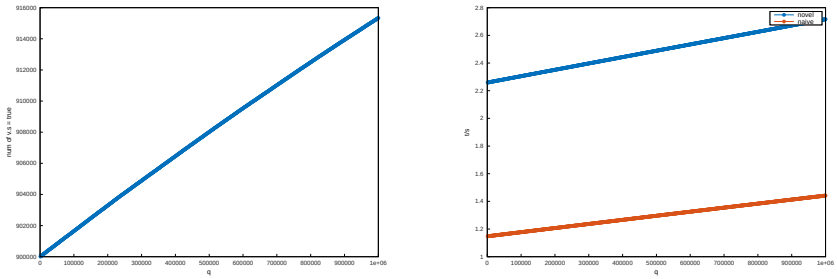
(a) $G_{\text{rand}}(10^6, 10^7)$ (b) $G_{\text{BA}}(10^6, 10)$

Figure 3.2: The degree distribution of a randomly generated graph from each of two kinds of graphs.

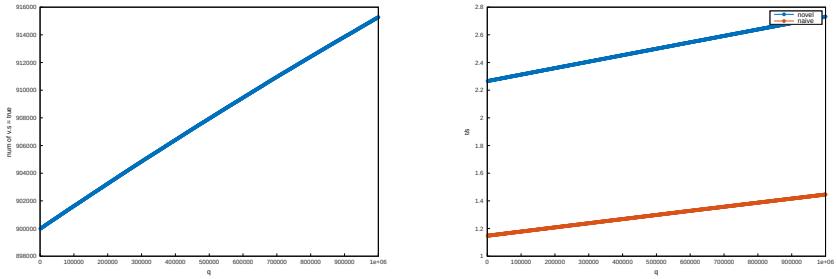
are weighted, since its degree distribution is approximately a normal distribution with almost no large vertices; but for a Barabási–Albert graph, its degree distribution in its tail follows a power law so that $P(x) \sim x^{-3}$ for some degree x, so there will be, even though few, large vertices.

The results for both kind of graphs are shown here for no weight, degree as weight, and power of degree as weight, all setting $crit = \sqrt{m}$. Note that running time begins from non-zero, since the execution of the initial n instructions were taken into account. It can also be observed that when the input is unfavorable to the novel algorithm, if initialization time is ignored, the time difference will be negligible.

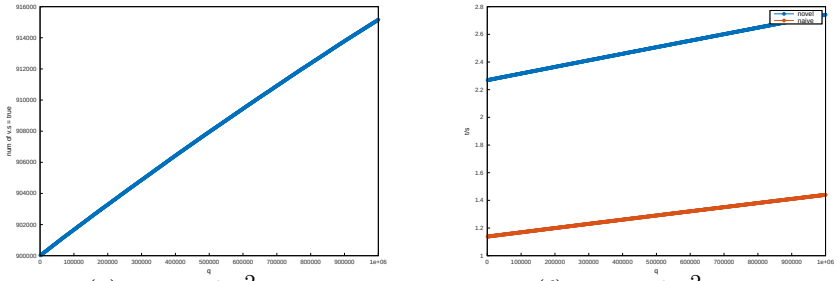
It was also attempted to find the optimal *crit* value, but I am unable to find a clear trend of running time for varying *crit*.



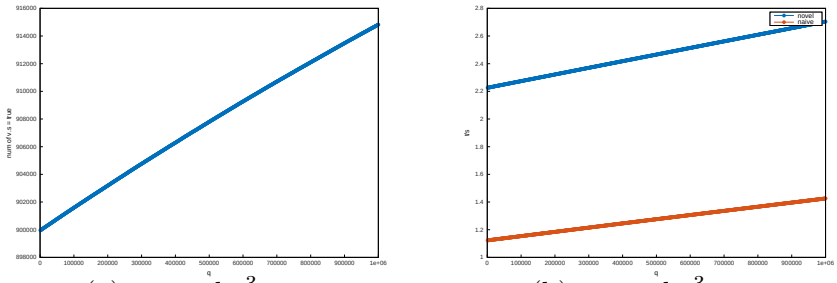
(a) All w_v are equal (b) All w_v are equal



(c) $w_v = deg_v$ (d) $w_v = deg_v$

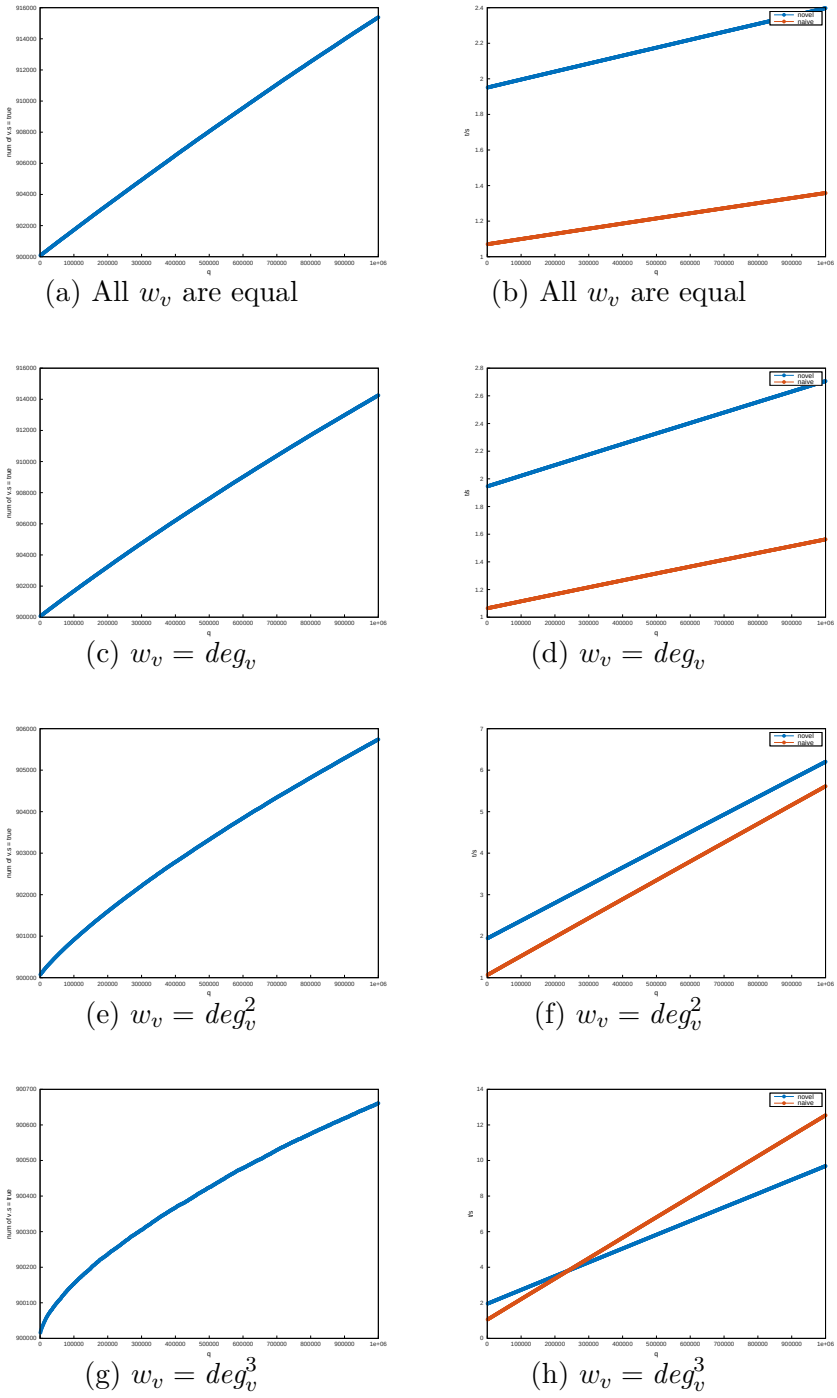


(e) $w_v = deg_v^2$ (f) $w_v = deg_v^2$



(g) $w_v = deg_v^3$ (h) $w_v = deg_v^3$

Figure 3.3: Left: number of $v.s = true$ after executing q operations on $G_{\text{rand}}(10^6, 10^7)$; Right: time in seconds after executing q operations on $G_{\text{rand}}(10^6, 10^7)$; All tests done by repeating on 5 graphs and 5 sets of operations for each graph and averaging the results.



(a) All w_v are equal (b) All w_v are equal

(c) $w_v = deg_v$ (d) $w_v = deg_v$

(e) $w_v = deg_v^2$ (f) $w_v = deg_v^2$

(g) $w_v = deg_v^3$ (h) $w_v = deg_v^3$

Figure 3.4: Left: number of $v.s = true$ after executing q operations on $G_{\text{BA}}(10^6, 10)$; Right: time in seconds after executing q operations on $G_{\text{BA}}(10^6, 10)$; All tests done by repeating on 5 graphs and 5 sets of operations for each graph and averaging the results.

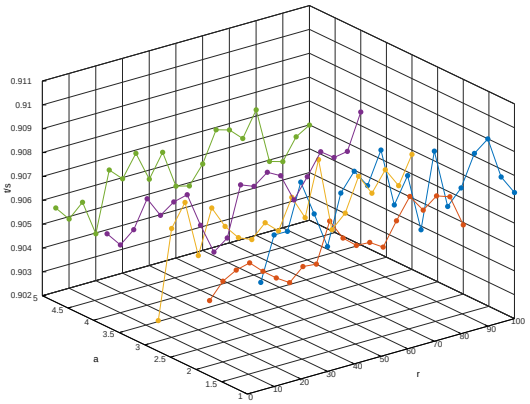


Figure 3.5: Time in seconds for varying a and r; for each data point, let $w_v = deg_v^a$ and $crit = r\sqrt{m}$, then execute 10^6 operations on $G_{\text{BA}}(10^6, 10)$; tests done by repeating on 7 graphs and 7 sets of operations for each graph and averaging the results.

Chapter 4

Applications and Modifications

I suspect this algorithm might have some use in machine learning and artificial neural networks, but I was yet to find a case that might require locking the states of vertices. I do observe that the technique of **label propagation** bears some similarity to this problem.[7] But instead of going from a *false* state to a *true* state, states can now be of many different labels, but that does not change the nature of the problem since the states always go from null label to some label. What it was really lacking was the need to lock the states of vertices.

This algorithm could also have some applications on the transmission of information on graphs of interpersonal ties, for their nature of being approximately Barabási-Albert graphs, and locking the vertices could represent that some persons are unavailable.[8] Also to note is that more "influential" persons (vertices of large degree) tend to be more active, so this might give my algorithm an edge over the naive one.

A similar application would be modelling the spreading of disease, which was also the algorithm's original background.[9] In this case, locking a vertex can represent the variables that prevent disease transmission. But such problem is much more complicated since more variables need to be taken into consideration, the edges now may have weights, and each vertex could be in one of many different states with some criteria to enter another state. Such simulation will also take time into consideration. Thus it would be necessary for the algorithm to have some kind of event processing, and the queues will store timestamp to allow simulation in time steps.

Chapter 5

Conclusions

This paper described an algorithm to simulate state propagation on graph while allowing the vertices to be locked. It was also found that this algorithm will perform better than the naive implementation only when vertices of high degree were accessed much more often. Some possible applications were discussed about this algorithm. How exactly weighing the vertices by the power of their degrees will be applicable, the better ways to find the optimal *crit* value, and how the probability distribution of the type of the operations will affect the performance of this algorithm will be left for further studies. It might also be good to have an adaptive version of the algorithm to increase its performance in different situations better.

Bibliography

[1] D. Shanks, Class number, a theory of factorization and genera, Proc. Symp. Pure Math, 20, Pages 415—440. AMS, Providence, R.I., 1971.

[2] Erdős, P.; Rényi, A, On Random Graphs. I, Publicationes Mathematicae, 6, Pages 290–297, 1959.

[3] Albert, Réka; Barabási, Albert-László, Statistical mechanics of complex networks, Reviews of Modern Physics, 74 (1), Pages 47–97, 2002.

[4] Freeman, Linton, A set of measures of centrality based on betweenness, Sociometry, 40 (1), Pages 35–41, 1977.

[5] Brin, S.; Page, L., The anatomy of a large-scale hypertextual Web search engine, Computer Networks and ISDN Systems, 30 (1–7), Pages 107–117, 1998.

[6] Korn, A.; Schubert, A.; Telcs, A., Lobby index in networks, Physica A, 388 (11), Pages 2221–2226, 2009.

[7] Zhu, Xiaojin, Learning From Labeled and Unlabeled Data With Label Propagation, 2002.

[8] Granovetter, Mark, The Impact of Social Structure on Economic Outcomes, Journal of Economic Perspectives, 19 (1), Pages 33–50, 2005.

[9] Rasim Alguliyev, Ramiz Aliguliyev, Farhad Yusifov, Graph modelling for tracking the COVID-19 pandemic spread, Infectious Disease Modelling, Volume 6, Pages 112-122, 2021.