Seminar: Advances in Dynamic Algorithms
Maximal Independent Sets

Thilo Leon Fischer

June 24, 2020

Abstract

This report presents a performance comparison of different algorithms for the dynamic maximal independent set problem. The evaluated algorithms were created by Gupta et al. [7] and Assadi et al. [5]. The algorithms were implemented in Python 3 and applied to datasets of varying size. The execution time for the different algorithm update functions is measured and compared.

1 Introduction

First, some basic definitions are made and the problem of maximal independent sets is introduced.

A graph $G = (V, E)$ is a tuple consisting of a set of nodes V and a set of edges $E \subset V \times V$. The number of nodes is denoted by n, the number of edges by m. The degree of a node v is $deg(v)$. The maximum degree is refered to as Δ.

An independent set IS is a subset of nodes with the property that no two nodes in the IS are neighbours, as formalised in equation 1.

$$\forall u, v \in IS : \{u, v\} \notin E \tag{1}$$

An independent set is considered maximal, if no node can be added to it without violating the independent set property. A maximal independent set is abbreviated as MIS in this report. A MIS is different from a maximum independent set, which is an IS with the biggest possible size.

To compute a MIS for a given graph, it is possible to loop over all nodes and consider a node to be in the MIS if no previous neighboring node is inside the MIS. This approach has a complexity of $\mathcal{O}(m)$, and is called the *Trivial* algorithm $\mathcal{A}_T$ in this report.

However, in practice edges and nodes might be removed or added dynamically to the graph. In this dynamic setting, the algorithm should calculate the set of nodes in the MIS after each update. Executing $\mathcal{A}_T$ again after every update is not optimal. Instead it is desirable to maintain information about the MIS across updates to increase calculation speed.

Alternatively, this model can be relaxed to an implicit model. Then, the algorithm is not required to maintain an explicit copy of the MIS. Instead it implements a query system, where it answers whether or not a specific node is in the MIS [7].

The next section 2 presents several different algorithms to compute a MIS. In section 3 details of the implementation are discussed. The performance of these implementations is then evaluated in section 4. Finally, a summary is contained in section 5.

All code and files relevant to this report can be accessed at:
`https://www.github.com/thilofischer/dynamic_mis`.

2 Algorithms

In this section several different algorithms will be presented to solve the MIS problem in a dynamic setting. All algorithms that are discussed here use the papers from Gupta et al. [7] and Assadi et al. [5] as reference.

The first algorithm was already mentioned in the introduction 1. It is the so called *Trivial* $\mathcal{A}_T$ algorithm.

After each update to the graph, the algorithm is executed again; It computes the MIS from scratch for every update to the graph. This means that the time complexity is $\mathcal{O}(m)$ and the adjustement complexity is $\mathcal{O}(n)$.

The second algorithm is called the *Simple* $\mathcal{A}_S$ algorithm. The approach in this algorithm is to maintain a counter for each node. The values of this counter represents how many of its neighbours are inside the MIS. If the counter for a node is zero, then this node should be inside the MIS. To keep this counter value accurate, every node that leaves or enters the MIS has to inform its neighbours so that they can update their counter value. Each update may cause one node to leave the MIS and may cause up to Δ nodes to join. These joining nodes also have to inform their neighbours. Overall the amortized update time complexity is $\mathcal{O}(\Delta)$.

Gupta et al. presented a slighty improved version of this $\mathcal{A}_S$ algorithm. In this report it is refered to as the *Improved Incremental* $\mathcal{A}_{II}$ algorithm. The difference to $\mathcal{A}_S$ is in the case of an edge insertion between two nodes that are in the MIS. $\mathcal{A}_S$ does not specify which node should leave the MIS. In $\mathcal{A}_{II}$, the node with lower degree is removed. This leads to a smaller number of neighbours that need to be informed about the state change.

The fourth algorithm $\mathcal{A}_D$ has a very fast amortized update time complexity of $\mathcal{O}(min\{\Delta, m^{2/3}\})$. This is achieved by treating nodes differently based on their degree. Heavy nodes are nodes with relatively many incident edges, while light nodes are those with fewer neighbours. The MIS is computed in two steps. In $\mathcal{A}_S$ a heavy nodes has to inform many neighbours whenever it leaves or joins. For this reason, $\mathcal{A}_D$ only maintains a counter for light nodes. The counter is maintained in the same way as in the *Simple* algorithm. At this stage the selected light nodes form an independent set but it may not be maximal. So after every update, the $\mathcal{A}_T$ algorithm is applied to the heavy nodes that are not adjacent to a light node that is in the MIS.

The final algorithm $\mathcal{A}_{Impl}$ operates in a different model than the previous algorithms. Instead of maintaining an explicit copy of the MIS, only an independent set is kept in memory. In this report, this algorithm is called the *Implicit* algorithm. To get information about the MIS, the user of the algorithm has to query the algorithm about the state of a specific node. Then the algorithm checks whether the node is in the independent set or if it can be added to it. Here, once again, nodes are differentiated by their degree. $\mathcal{A}_{Impl}$ maintains a counter only for heavy nodes. The dynamic calculation of the count would be too slow for heavy nodes. For light nodes however, this calculation can be done inside the query function, because they have few neighbours.

In the implicit model not the amortized complexity is considered but the worst case complexity. For $\mathcal{A}_{Impl}$ the worst case complexity of updates and queries is $\mathcal{O}(min(\Delta, \sqrt{m}))$.

3 Implementation

The algorithms presented in the previous section were implemented using Python 3.8.2. To store and perform operations on graphs, the *NetworkX* package is used. To perform randomized operations, *numpy.random* is used.

The strategy design pattern is applied to give all implementations a unified interface. This

interface is visualised in the UML diagram that can be seen in figure 3. The different algorithms shown in section 2 are implemented as subclasses of *Algorithm* and implement the six abstract functions.

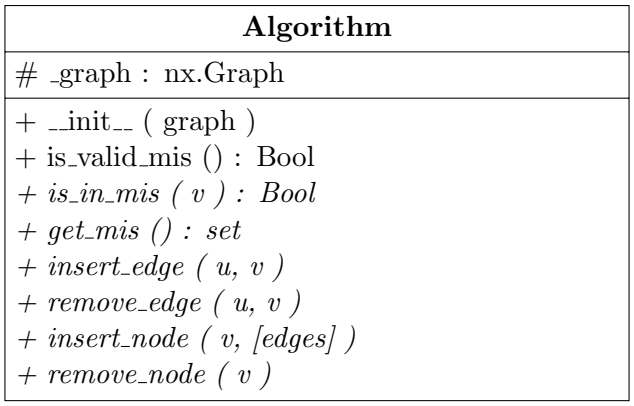

Figure 1: UML Class Diagram of the Algorithm interface

The __init__ () function takes a NetworkX graph as a parameter. In this function the algorithms perform the operations to initialize their internal data structures. The function is_valid_mis () checks whether the MIS, that the algorithm computed, is a valid MIS. This function should always return True. To access information about the MIS, the two functions is_in_mis (v) and get_mis() can be used. The former checks whether a given node is in the MIS, and the latter returns a set of all nodes in the MIS. Note that these functions have side-effects when the *Implicit* algorithm is used.

To perform an update, one of the four update functions can be used. When inserting a node, edges to existing nodes can be passed along but are optional.

3.1 Testing

To check that the implementation produces correct results, each algorithm was tested. The relevant source code is located in the file *tests/test_algorithm.py*.

To determine whether a set of nodes forms a MIS on a given graph, the function *Algorithm.is_valid_mis()* was implemented. In the testcases for this function, an algorithm object was patched using the Python module *unittest.mock* to return pre-determined values as an answer. This was done to produce valid and invalid maximal independent sets independently of any specific implementation. The function is tested against four edge cases, three of which expect the return value to be *False*.

Using the *Algorithm.is_valid_mis*, function the actual algorithm implementations are tested. To generate data for these testcases $G(n, p)$-graphs were created. To make the testcases reproducible, fixed seeds are used, so that every run of the testcase operates on the same graph.

For each algorithm four tests are performed. These testcases correspond to the four different updates: insert/remove node/edge. Each testcase consists of a loop that performs the updates until all nodes/edges are inserted or removed. For the node insertion case, the starting graph is completely empty. For edge insertions, the starting graph consists only of nodes.

After each update, it is verified that the set calculated by the algorithm is actually a MIS, and that the update has been performed on the graph. For the insertion testcases, after all updates

have been performed the final graph is compared against the original graph to check whether the resulting graph is the desired graph.

4 Evaluation

To evaluate the performance of the different algorithms, the execution time of multiple updates is measured. For this purpose five different datasets with varying size are used. The datasets are called:

1. aves-wildbirds-networks [10]
2. topology [3, 8, 12]
3. facebook-wosn-links [2, 8, 11]
4. youtube-u-growth [4, 8, 9]
5. brightkite [1, 6, 8]

The datasets 1-4 are used for edge insertion benchmarks, while dataset 5 is used to evaluate edge deletion. The size of the datasets is shown in table 1.

Table 1: Size of the datasets

Dataset	1	2	3	4	5
Nodes	202	34,761	63,731	3,223,589	52,228
Edges	11900	114,496	817,035	9,375,374	214,078

One benchmark run consists of two parts. First is the initialization of the data structures used by the algorithm. Here, the initial MIS is calculated for the starting graph, with exception of the **ImplicitMIS** algorithm, which does not keep an explicit copy of the MIS.

The second part of a benchmark run is a loop that calls an *Algorithm* function to perform an update. In listing 1, a code snippet of the benchmarking code is shown.

Listing 1: Benchmark Code Snippet

```
1 def execute():
2         algo = algo_cls(graph)
3         for e in edges:
4             algo.insert_edge(*e)
5
6 t = timeit.timeit(execute, number=1)
```

In order for the measured execution time to be reliable, five runs of a benchmark are done. The results are then averaged. Averaging is chosen over a boxplot, because the individual times only show very minor deviations. For the detailed output, including the time of every single run, refer to the files inside the **log/** folder in the repository.

Note that in listing 1, on line 6, the number of runs is set to one. Because the graph is modified during a run, a new graph instance needs to be used for each run. However, the copy operations to create a new graph do not depend on the algorithm under evaluation. So these operations should not contribute to the run time of the algorithm, as they would only introduce more unknown variables. Instead the *timeit* function only performs one benchmark run at a time and the code shown in listing 1 is executed five times to arrive at the final result.

4.1 Results

In this subsection, the results of the benchmark runs are presented. The benchmarks were executed on an Intel Core i7-1065G7 with 16GB of memory. The two updates that are benchmarked are edge insertions in section 4.1.1 and edge deletions in section 4.1.2.

4.1.1 Edge Insertions

In table 2 the average execution times are shown. The values are the average over five runs, in seconds. In case the execution took too long, the table contains a **DNF**, denoting that the algorithm did not finish. The threshold for this is 2 minutes. This cut-off was chosen to keep the time required to perform five iterations of each algorithm manageable. One evaluation of the *Youtube* dataset for the three fastest algorithm takes over 20 minutes.

Table 2: Time for Edge Insertions

Data	Trivial $\mathcal{A}_T$	Simple $\mathcal{A}_S$	Incremental $\mathcal{A}_{II}$	Dynamic $\mathcal{A}_D$	Implicit $\mathcal{A}_{Impl}$
Wildbirds	1.281	0.013	0.012	0.064	0.020
Topology	DNF	0.430	0.504	3.205	0.696
Facebook	DNF	2.582	2.228	DNF	4.042
Youtube	DNF	67.058	60.695	DNF	95.488

First note the execution times for the *Trivial* algorithm. It is considerably slower than the other algorithms on the smallest dataset. And for the next biggest dataset it does not compute the result in 2 minutes. This clearly demonstrates the need for a dynamic approach to solve the MIS problem in an environment with many updates.

Now compare the execution times of all algorithms on the *Wildbirds* dataset. *Simple* is the second fastest after the improved version of it, the *Improved Incremental* algorithm. It makes a better choice for removing a node from the MIS for the cost of an additional degree comparison. In the largest dataset, the improvement to the *Simple* algorithm manifest in an improvement of around 9%, when comparing the *Improved Incremental* to the *Simple* algorithm.

The *Dynamic* algorithm presented in [7] performs worse than the simpler algorithms. Profiling the code did not reveal any obvious bottlenecks for this algorithm. However, inspecting the datasets showed in the *Youtube* dataset only six nodes are considered heavy in the full dataset. In the complete graph of the *Facebook* dataset no nodes are heavy. The same is true for the *Wildbirds* network. In the *Topology* dataset three nodes are heavy in the full graph. Although it is not as severe, the *Implicit* algorithm is also slower.

Thus it is likely that for these specific datasets, the more sophisticated algorithms can not realize their full potential. In the contrary, the more complex code adds overhead and slows the execution down.

Another case where more complex code slowed the execution down noticeably was in the implementation of $\mathcal{A}_{II}$. It is almost identical to the *Simple* algorithm. Thus the first implementation used inheritance. This approach is shown in listing 2. However the additional **if**-statement and function call negated the speed improvement from differentiating between the lower and higher degree node. The algorithm that is faster in theory was actually slower in practice.

Instead, it is faster to inline the code from *Simple*. This inlined implementation was used for the benchmarks. Then, an improvement compared to $\mathcal{A}_S$ can be seen in three of the four datasets.

Listing 2: Slow implementation

```
1 class ImprovedIncrementalMIS(SimpleMIS):
2   def insert_edge(self, u, v):
3     lower_deg, higher_deg = (u, v) if self._graph.degree(u) < self.
          _graph.degree(v) else (v, u)
4     SimpleMIS.insert_edge(self, lower_deg, higher_deg)
```

4.1.2 Edge Deletions

To evaluate the performance of edge deletions, the *Brightkite* [1, 6, 8] dataset is used.

The edges to be removed were chosen randomly but in a reproducible way. So every benchmark run performs exactly the same operations. The first benchmark removes 1,000 edges, the second 10,000, the third removes 100,000 edges. For edge deletions, the *Improved Incremental* algorithm is not used, because it only supports incremental updates. For deletions, it is identical to the *Simple* algorithm.

Table 3: Time for Edge Deletions

Removals	Trivial $\mathcal{A}_T$	Simple $\mathcal{A}_S$	Dynamic $\mathcal{A}_D$	Implicit $\mathcal{A}_{Impl}$
1000	77.246	0.136	0.158	0.100
10,000	DNF	0.157	0.266	0.166
100,000	DNF	0.361	1.252	0.793

In this scenario *Trivial* performs even worse than in the insertion benchmark in the previous section. While the three more sophisticated approaches are fairly evenly matched for few updates. When stepping up from 1,000 deletions to 10,000, the execution time does not increase very much. This is caused by the slow initialization phase. In the insertion case, the initial graph consists only of nodes without their edges. So the initialization is basically a loop that adds all nodes to the MIS, and thus is relatively fast. However, in the deletion case, the initial graph is the full dataset. As shown in figure 1, the brightkite dataset has 52,228 nodes and 214,078 edges.

To initialize, $\mathcal{A}_S$ performs a single execution of the $\mathcal{A}_T$ algorithm. This alone takes ca. 0.140 seconds, 89% of the runtime for 10,000 updates. When only few updates are performed, the *Implicit* algorithm benefits from the low initialisation overhead. Again, the *Dynamic* algorithm from [7], is the slowest. In theory, the *Dynamic* algorithm is faster by treating nodes differently based on their degree. However, after performing the benchmark, inspection showed, that in this dataset no node is considered to be heavy. A similar observation as for the datasets 1-4.

5 Summary

In this paper five different algorithms to solve the dynamic MIS problem were evaluated on five datasets. Each of these algorithms belongs to a different complexity class. The *Trivial* algorithm perfomed very poorly compared to the other algorithms. This demonstrated that applying a static algorithm to a dynamic use case does not yield acceptable results. Also the $\mathcal{A}_D$ and $\mathcal{A}_{Impl}$ algorithm did not perform as expected. They were slower, in some cases by orders of magnitude, than their more simple counterparts. However, this observation might be caused by the specific datasets that were employed for the benchmarking. $\mathcal{A}_D$ and $\mathcal{A}_{Impl}$ have a lower amortized complexity because

they treat nodes with a high degree differently. The threshold for a node to be considered heavy is so high that, for these datasets, no improvement could be observed. Rather the more complex code added overhead which slowed the execution down.

A conclusion from this observed behavior could be that when chosing an algorithm in practice, not only should the theoretical complexity be a factor but also the data it will operate on. And whether the algorithm is suited for the specific kind of data. In the future, it would be interesting to evaluate the algorithms on datasets with a higher density of edges. This could show how $\mathcal{A}_D$ and $\mathcal{A}_{Impl}$ perform on data that allows them to leverage their lower theoretical complexity.

References

[1] Brightkite network dataset – KONECT, September 2016.

[2] Facebook friendships network dataset – KONECT, September 2016.

[3] Internet topology network dataset – KONECT, September 2016.

[4] Youtube network dataset – KONECT, October 2016.

[5] Sepehr Assadi, Krzysztof Onak, Baruch Schieber, and Shay Solomon. Fully dynamic maximal independent set with sublinear in n update time. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1919–1936. SIAM, 2019.

[6] Eunjoon Cho, Seth A. Myers, and Jure Leskovec. Friendship and mobility: User movement in location-based social networks. In *Proc. Int. Conf. on Knowledge Discovery and Data Mining*, pages 1082–1090, 2011.

[7] Manoj Gupta and Shahbaz Khan. Simple dynamic algorithms for maximal independent set and other problems. *arXiv preprint arXiv:1804.01823*, 2018.

[8] Jérôme Kunegis. KONECT – The Koblenz Network Collection. In *Proc. Int. Conf. on World Wide Web Companion*, pages 1343–1350, 2013.

[9] Alan Mislove. *Online Social Networks: Measurement, Analysis, and Applications to Distributed Information Systems*. PhD thesis, Rice University, 2009.

[10] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[11] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P. Gummadi. On the evolution of user interaction in Facebook. In *Proc. Workshop on Online Social Networks*, pages 37–42, 2009.

[12] Beichuan Zhang, Raymond Liu, Daniel Massey, and Lixia Zhang. Collecting the Internet AS-level topology. *SIGCOMM Computer Communication Review*, 35(1):53–61, 2005.