Graph Mining Performance Study

Anuj Ketkar
anujketkar@tamu.edu

Wyatt McGinnis
mw8088@tamu.edu

Shurui Xu
shuruixu@tamu.edu

Khanh Nguyen
khanhtn@tamu.edu

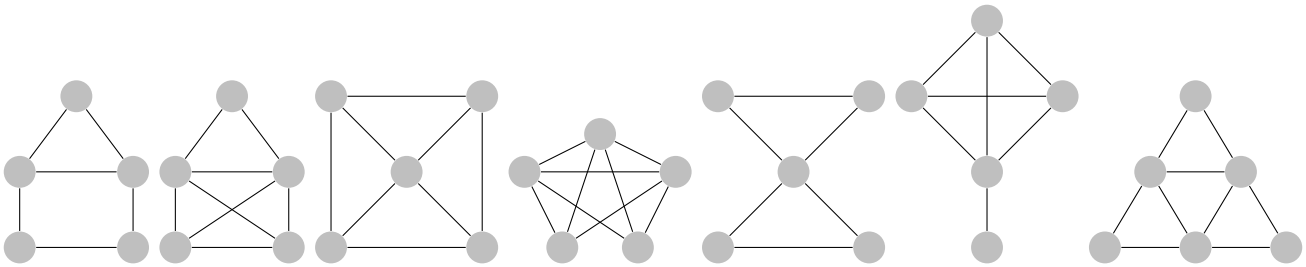
1. Overview

Analytics based on graph mining has become increasingly popular across various important domains including bioinformatics, computer vision, and social network analysis. Graph mining workloads aim to extract the structural properties of a graph by exploring its subgraph structures. Researchers have posited that a new breed of Machine Learning (ML), named graph-based machine learning (Graph AI), can lead to a significant advance in many frontiers, including cybersecurity analytics. Graph mining systems are provided as input subgraph structures of interest (i.e., the pattern) which guide the overall exploration process. It has been observed that an input pattern with different vertex ids can result in different performances in terms of execution time and memory consumption. In this study, we investigate the proliferation and severity of this problem, which, to the best of our knowledge, has not been explored before.

Systems We include Peregrine [2] and GraphPi [3] in our study as representatives for graph mining systems. Peregrine is the state-of-the-art single-machine-based system. GraphPi is a high performance distributed pattern matching system. We refer readers to their publications for details of the systems.

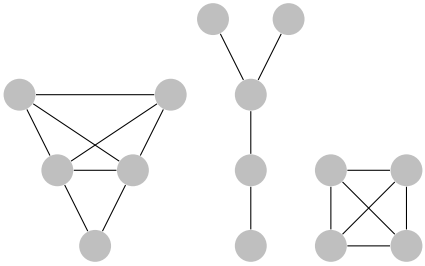
Environment All experiments were conducted on a server equipped with Intel(R) Xeon(R) CPU E5-2620, 24 cores and 252 GBs of memory, running Ubuntu 18.04.6. GraphPi was run in pseudo-distributed mode.

Patterns The patterns that were used for Peregrine are:



From left to right: House, xHouse, xBox, Apple, Hourglass, Kite, Triforce

For GraphPi, three additional patterns were used:



From left to right: Cone, Eiffel, xSquare

Workflow Each pattern of size n is run through an exhaustive generator to generate all $n!$ variants (e.g., for Triforce, which has 6 vertices, there are 720 variants generated). Each variant is then run three times and the average is reported and analyzed. As input, we used MiCo graph [1], a co-authorship graph that has 96.6M vertices and 1.1M edges.

2. Peregrine Result

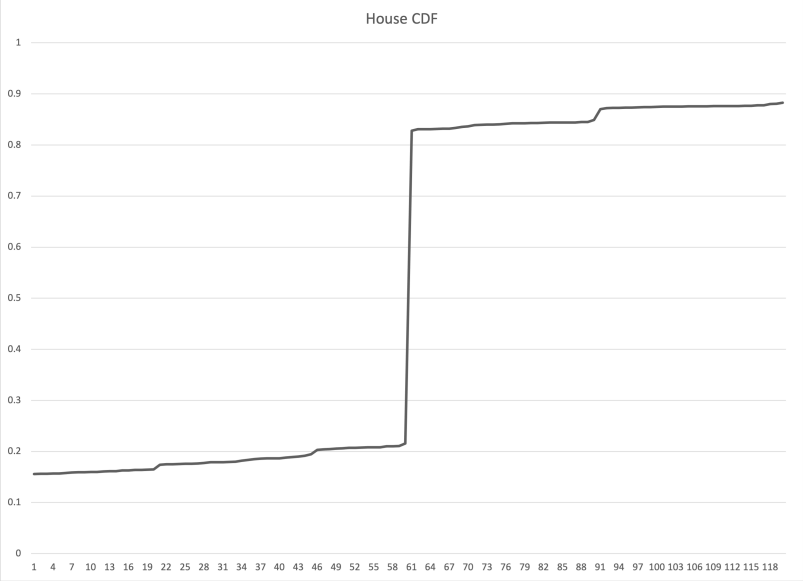
For each of the seven patterns tested, we include:

- The summary statistics of that pattern's tests
- The CDF graph for all the variants' execution time, which summarizes the behavior of the system
- The five fastest and slowest variants, for future explorations

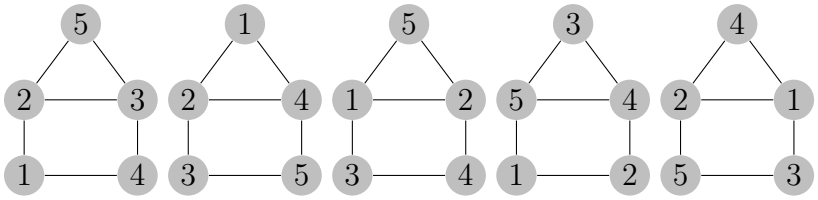
2.1 Results

2.1.1 House

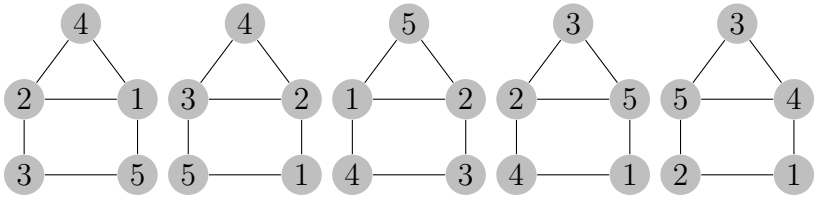
Time		**RSS Memory**		**Dirty Memory**	
Diff.	42.558%	Diff.	4.187%	Diff.	3.913%
Min	17.91503	Min	16384	Min	12472
Max	25.53933	Max	17070	Max	12960
Mean	21.69641075	Mean	16779.35	Mean	12768.2417
Std. dev.	3.461391778	Std. dev.	136.715963	Std. dev.	108.144006



Fastest patterns:

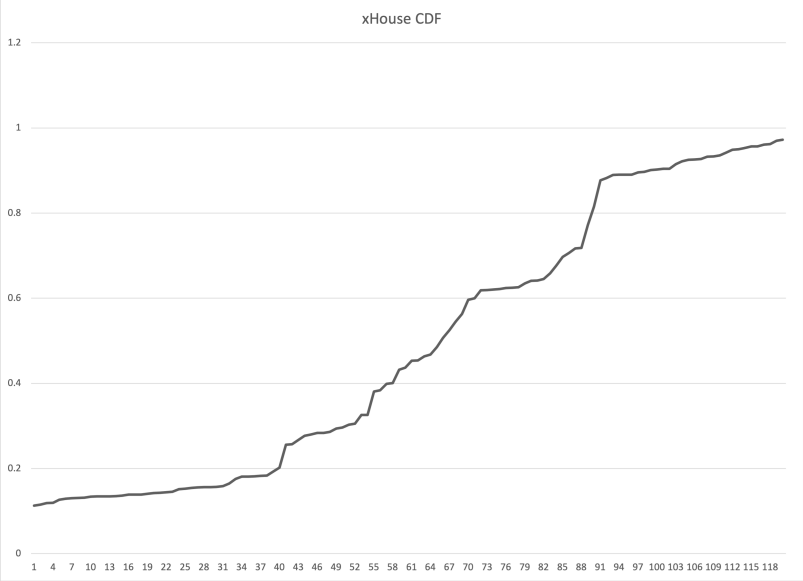


Slowest patterns:

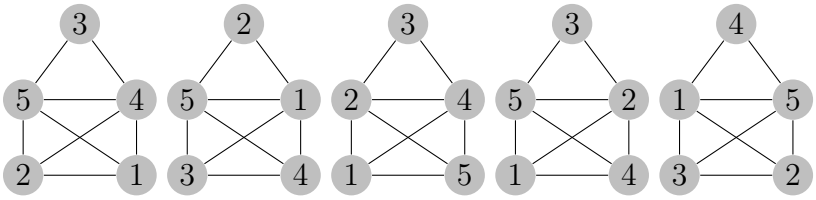


2.1.2 xHouse

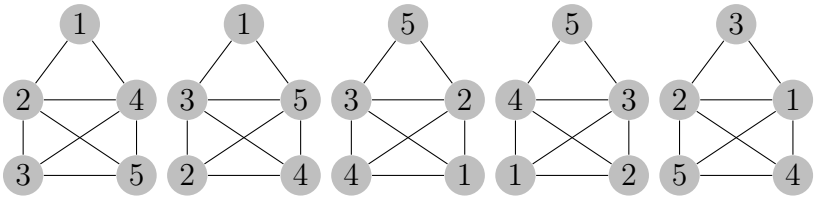
Time		**RSS Memory**		**Dirty Memory**	
Diff.	8.031%	Diff.	10.405%	Diff.	11.702%
Min	10.0602	Min	15800	Min	12024
Max	10.86813	Max	17444	Max	13431
Mean	10.37644667	Mean	17042.625	Mean	13044.4333
Std. dev.	0.258518707	Std. dev.	278.831537	Std. dev.	256.745197



Fastest Patterns:

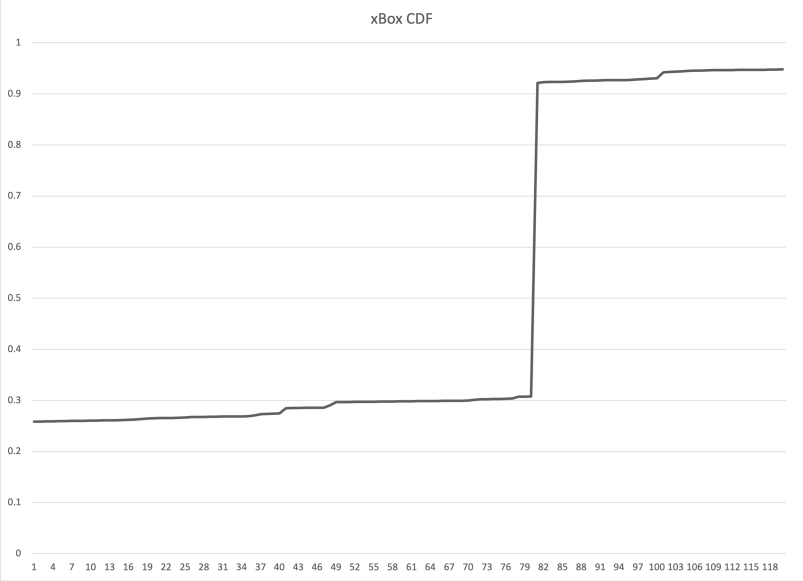


Slowest Patterns:

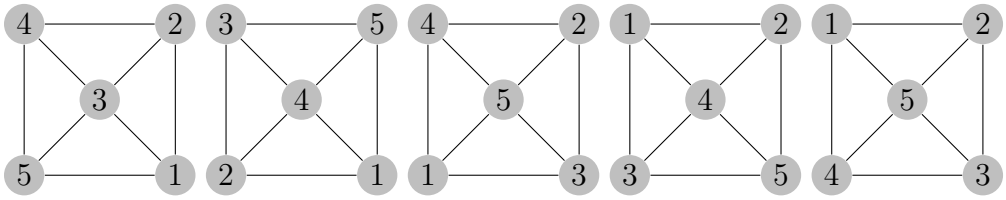


2.1.3 xBox

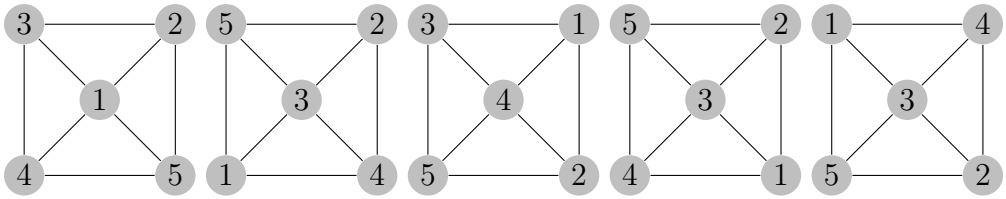
Time		**RSS Memory**		**Dirty Memory**	
Diff.	78.557%	Diff.	6.517%	Diff.	6.185%
Min	10.60676	Min	15759	Min	11948
Max	18.93916	Max	16786	Max	12687
Mean	13.43027808	Mean	16487.9667	Mean	12484.925
Std. dev.	3.663603462	Std. dev.	214.670078	Std. dev.	175.761688



Fastest Patterns:

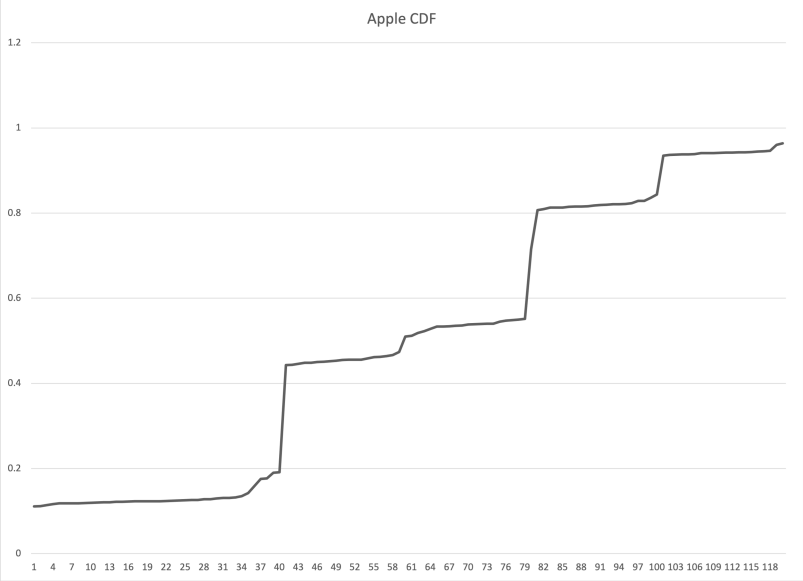


Slowest Patterns:

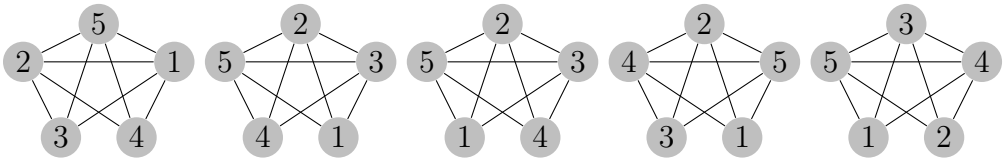


2.1.4 Apple

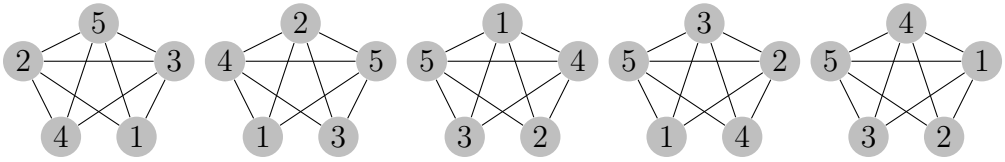
Time		**RSS Memory**		**Dirty Memory**	
Diff.	28.367%	Diff.	5.754%	Diff.	7.783%
Min	2.23518	Min	15432	Min	11461
Max	2.86923	Max	16320	Max	12353
Mean	2.500170917	Mean	15768.025	Mean	11795.7
Std. dev.	0.209946285	Std. dev.	179.651505	Std. dev.	177.164719



Fastest Patterns:

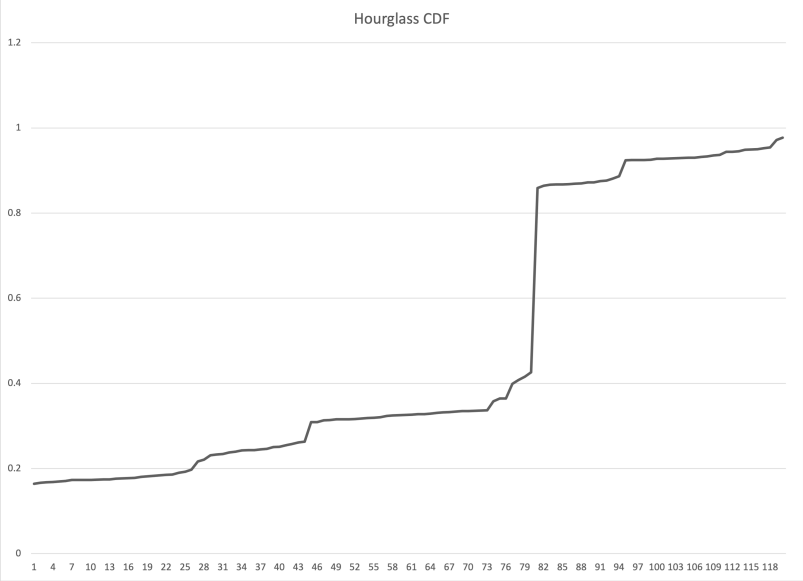


Slowest Patterns:

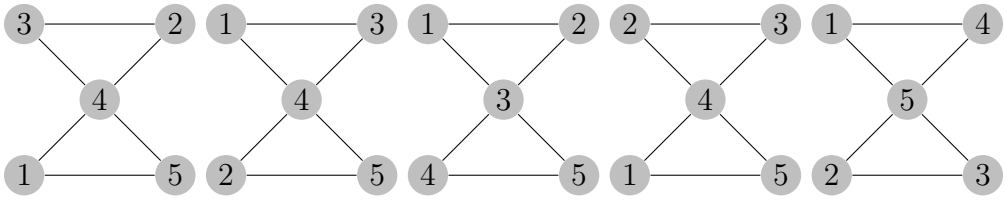


2.1.5 Hourglass

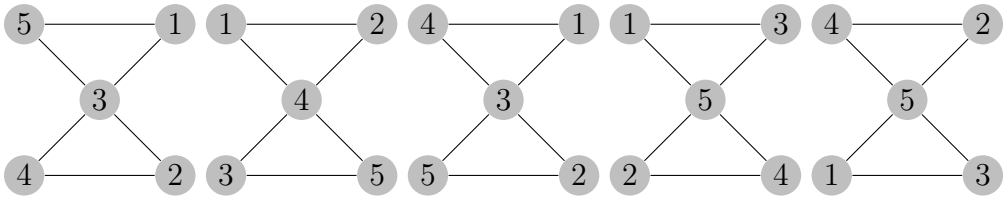
Time		**RSS Memory**		**Dirty Memory**	
Diff.	25.058%	Diff.	3.658%	Diff.	3.529%
Min	14.76633	Min	16346	Min	12439
Max	18.46653	Max	16944	Max	12878
Mean	16.02841167	Mean	16789.95	Mean	12778.2083
Std. dev.	1.242803891	Std. dev.	102.811556	Std. dev.	75.6034431



Fastest Patterns:

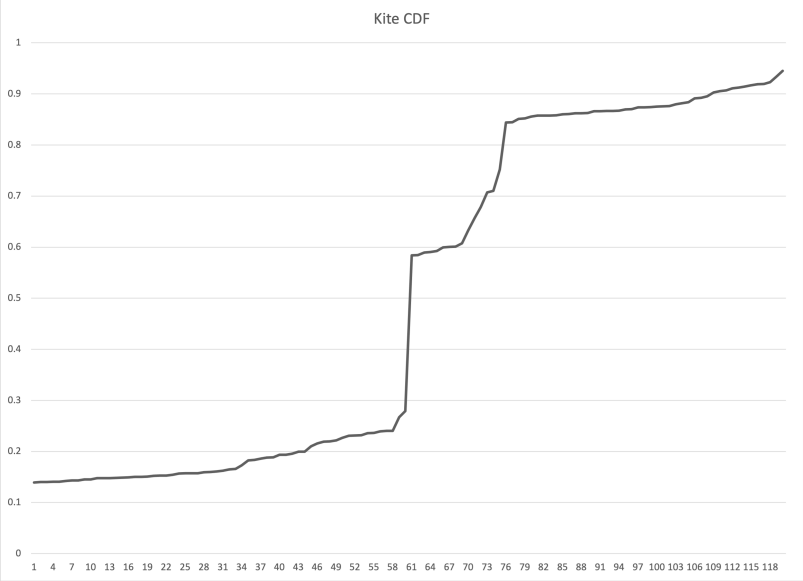


Slowest Patterns:

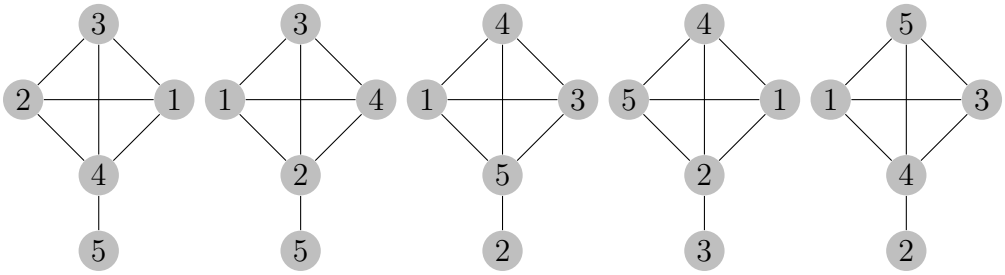


2.1.6 Kite

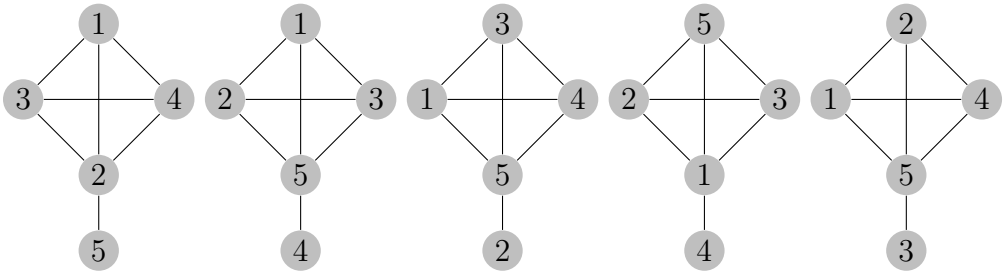
Time		**RSS Memory**		**Dirty Memory**	
Diff.	15.917%	Diff.	8.526%	Diff.	9.718%
Min	6.0563	Min	15823	Min	11988
Max	7.02031	Max	17172	Max	13153
Mean	6.455420833	Mean	16787.6833	Mean	12795.3917
Std. dev.	0.359184095	Std. dev.	256.093896	Std. dev.	226.637878



Fastest Patterns:

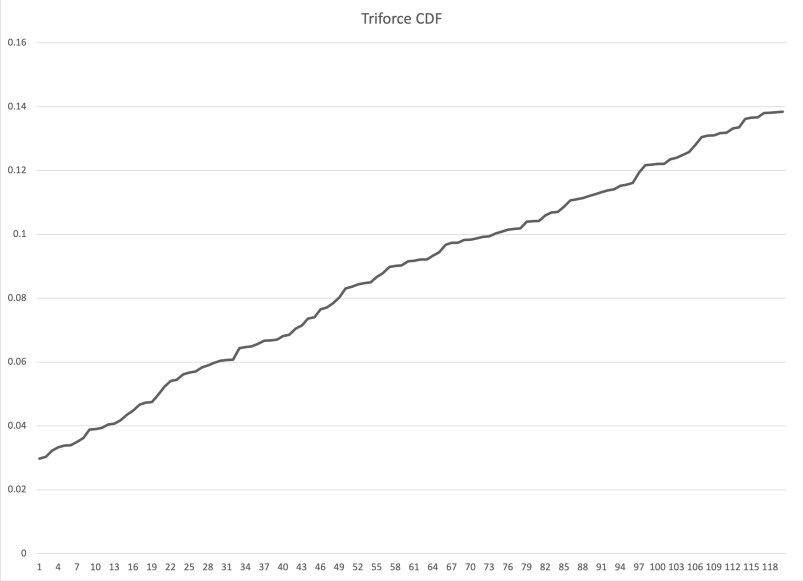


Slowest Patterns:

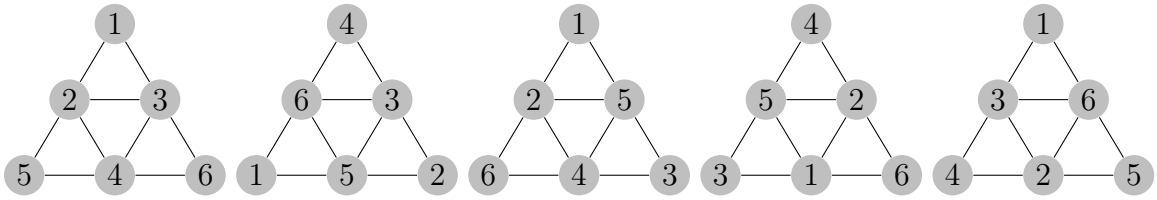


2.1.7 Triforce

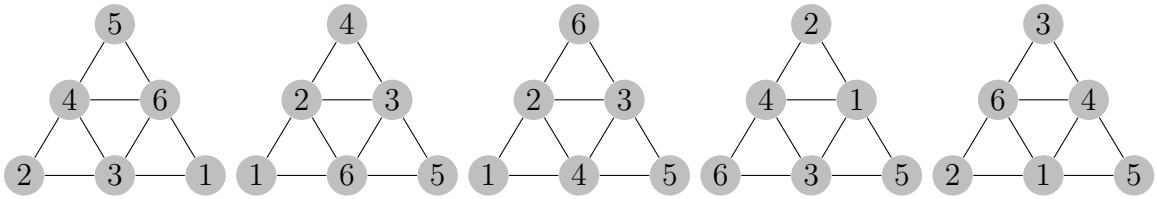
Time		**RSS Memory**		**Dirty Memory**	
Diff.	9.232%	Diff.	7.915%	Diff.	8.772%
Min	8.73159	Min	16285	Min	12505
Max	9.53773	Max	17574	Max	13602
Mean	9.105632306	Mean	17166.4097	Mean	13169.7319
Std. dev.	0.197391869	Std. dev.	229.749789	Std. dev.	200.758199



Fastest Patterns:



Slowest Patterns:



2.2 Discussion

Looking at the CDF graphs for House and xBox, there appears to be a single alteration that costs a significant difference in time (i.e., a large spike that separates the test cases into two groups). The Apple pattern may be an even more ideal test subject since there are multiple spikes throughout the distribution.

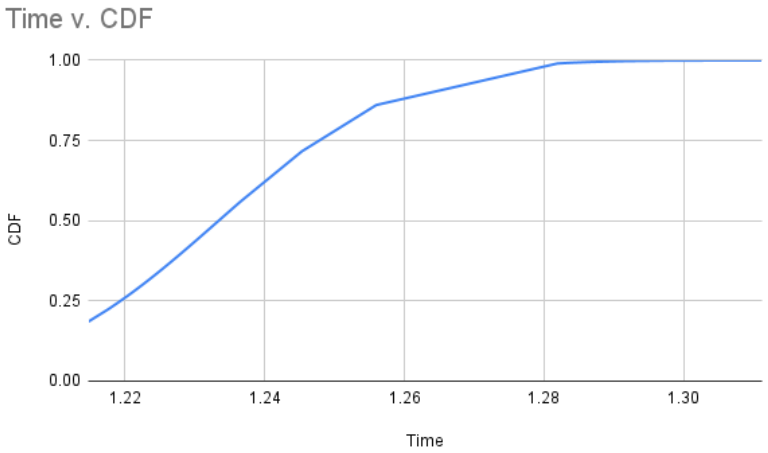
3. GraphPi Result

The data included in this Section is similar to that of Peregrine

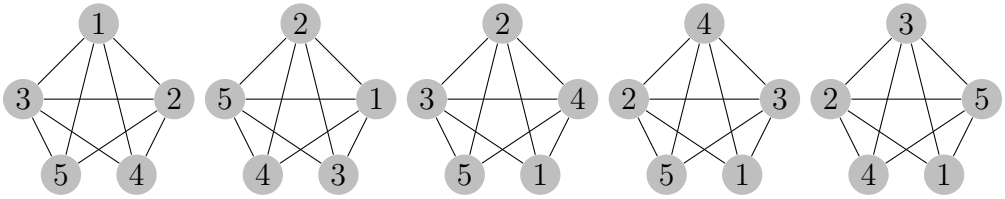
3.1 Results

3.1.1 Apple

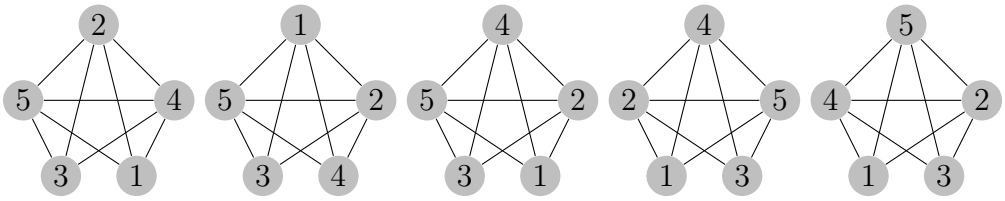
Time		**RSS Memory**		**Dirty Memory**	
Diff.	7.94155176%	Diff.	46.73564558%	Diff.	55.0514863%
Min	1.21475	Min	13724	Min	10294
Max	1.31122	Max	20138	Max	15961
Mean	1.233589669	Mean	18019.4876	Mean	13917.03306
Std. dev.	0.02084348007	Std. dev.	928.3474935	Std. dev.	879.6941697



Fastest Patterns:

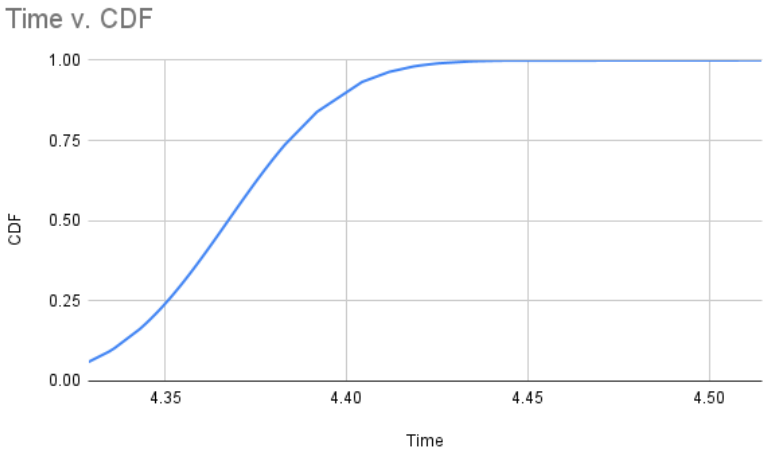


Slowest Patterns:

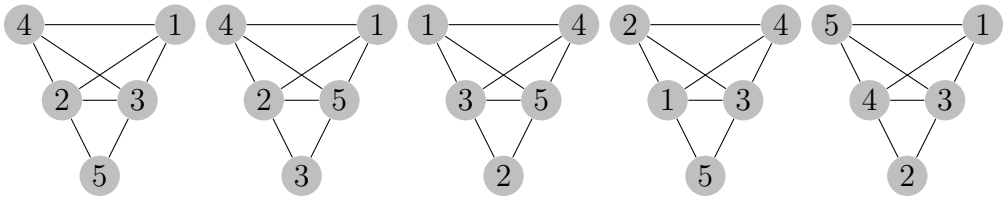


3.1.2 Cone

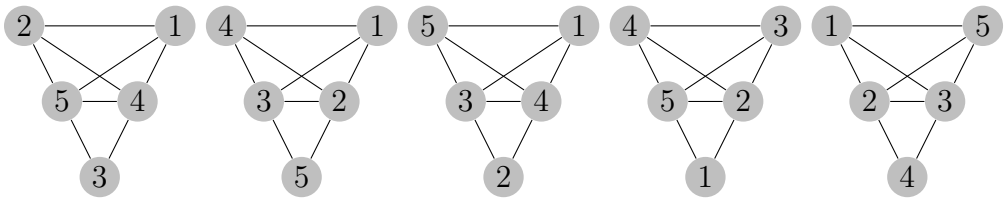
Time		**RSS Memory**		**Dirty Memory**	
Diff.	4.288888992%	Diff.	12.01756413%	Diff.	14.73167045%
Min	4.32886	Min	17308	Min	13230
Max	4.51452	Max	19388	Max	15179
Mean	4.367454793	Mean	18023.92562	Mean	13895.69421
Std. dev.	0.02470596872	Std. dev.	448.389826	Std. dev.	438.4434179



Fastest Patterns:

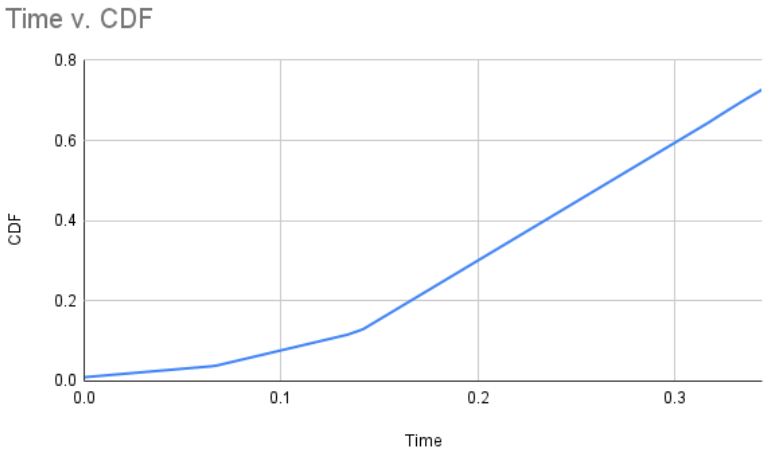


Slowest Patterns:

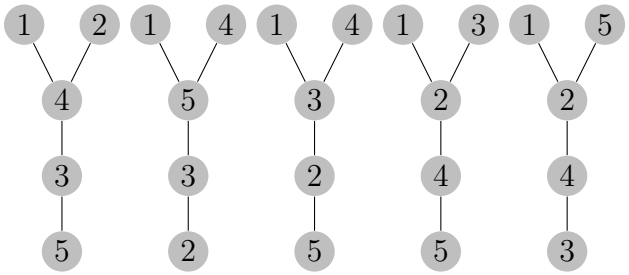


3.1.3 Eiffel

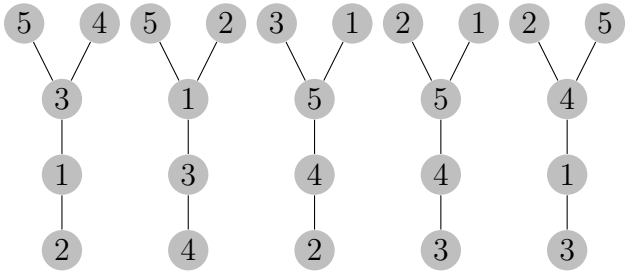
Time		**RSS Memory**		**Dirty Memory**	
Diff.	100.00%	Diff.	66.23874625%	Diff.	100.0631552%
Min	0	Min	11996	Min	7917
Max	0.34408	Max	19942	Max	15839
Mean	0.2701636364	Mean	16886.93388	Mean	12799.78512
Std. dev.	0.116925831	Std. dev.	1469.485872	Std. dev.	1465.178608



Fastest Patterns:

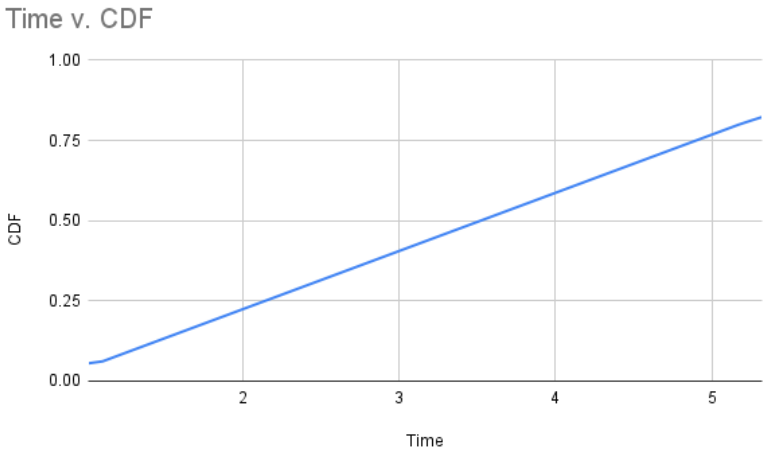


Slowest Patterns:

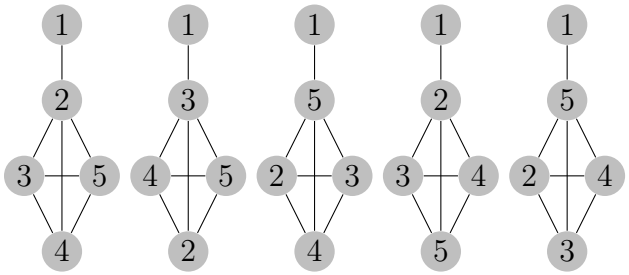


3.1.4 Kite

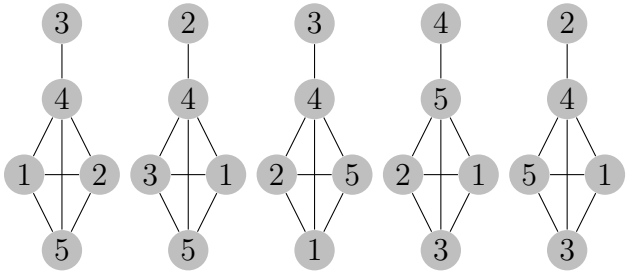
Time		**RSS Memory**		**Dirty Memory**	
Diff.	424.9324364%	Diff.	71.41957539%	Diff.	114.3613975%
Min	1.01386	Min	11116	Min	6984
Max	5.32208	Max	19055	Max	14971
Mean	4.363522397	Mean	17475.84298	Mean	13350.97521
Std. dev.	1.708609991	Std. dev.	1650.53975	Std. dev.	1608.27245



Fastest Patterns:

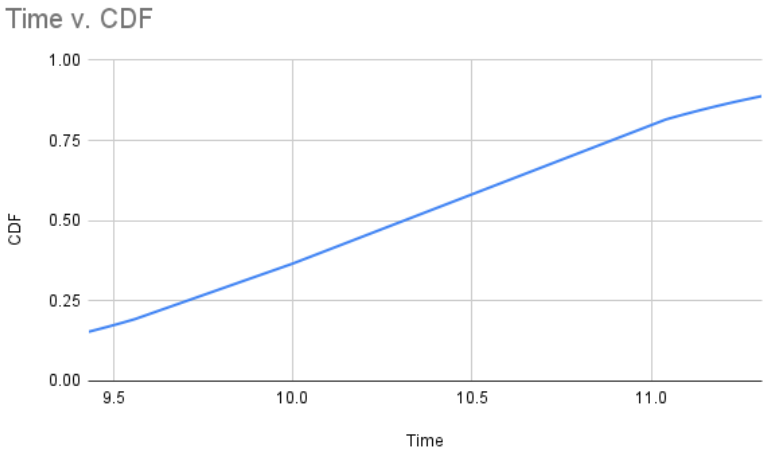


Slowest Patterns:

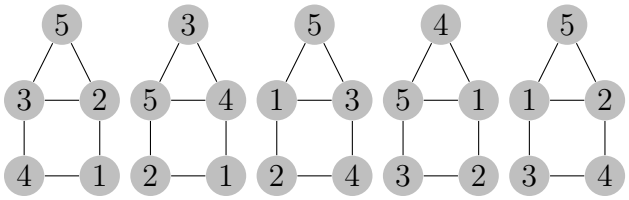


3.1.5 House

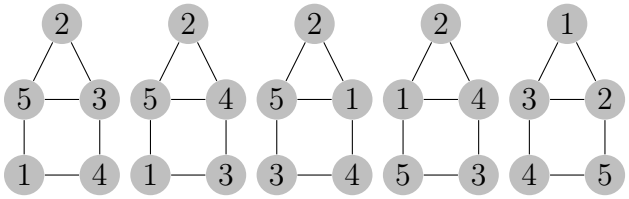
Time		**RSS Memory**		**Dirty Memory**	
Diff.	19.91712244%	Diff.	9.578344612%	Diff.	10.98669873%
Min	9.43078	Min	17289	Min	13307
Max	11.30912	Max	18945	Max	14769
Mean	10.32121446	Mean	18309.8595	Mean	14179.52066
Std. dev.	0.8388583459	Std. dev.	262.7996482	Std. dev.	260.1276001



Fastest Patterns:

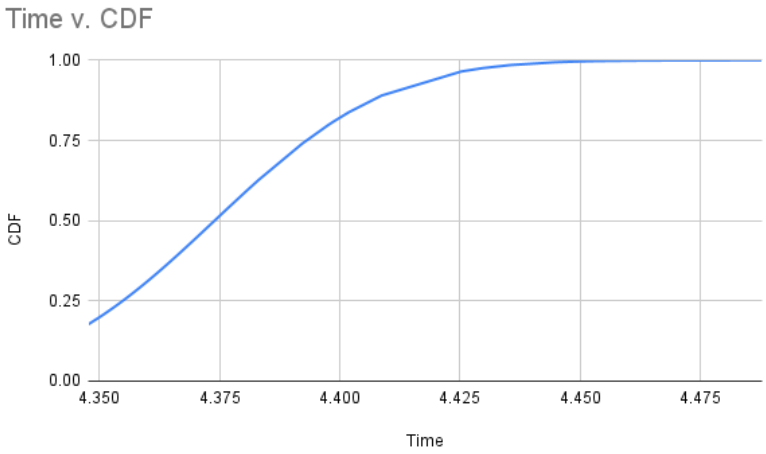


Slowest Patterns:

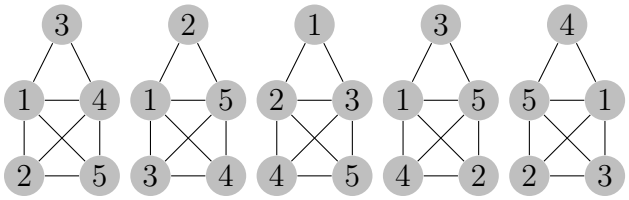


3.1.6 xHouse

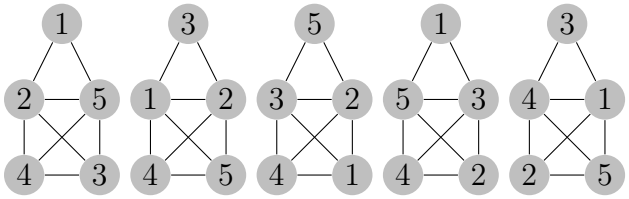
Time		**RSS Memory**		**Dirty Memory**	
Diff.	3.224255751%	Diff.	10.08185382%	Diff.	12.9725667%
Min	4.34767	Min	17348	Min	13305
Max	4.48785	Max	19097	Max	15031
Mean	4.374022893	Mean	17977.28099	Mean	13839.10744
Std. dev.	0.02833383697	Std. dev.	446.3755561	Std. dev.	443.8182962



Fastest Patterns:

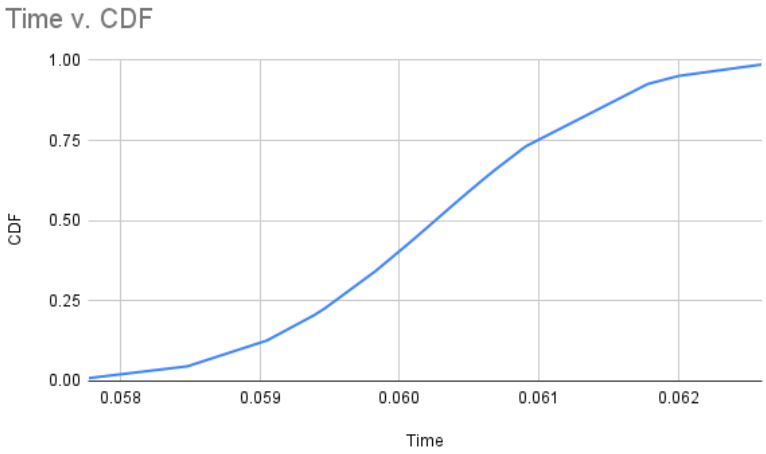


Slowest Patterns:

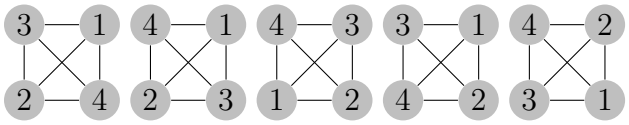


3.1.7 xSquare

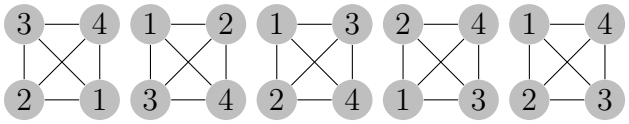
Time		**RSS Memory**		**Dirty Memory**	
Diff.	8.360740869%	Diff.	54.98050608%	Diff.	61.24054162%
Min	0.05777	Min	13081	Min	10044
Max	0.0626	Max	20273	Max	16195
Mean	0.0602644	Mean	14768.08	Mean	11438.12
Std. dev.	0.001054079693	Std. dev.	2213.238199	Std. dev.	1806.82314



Fastest Patterns:

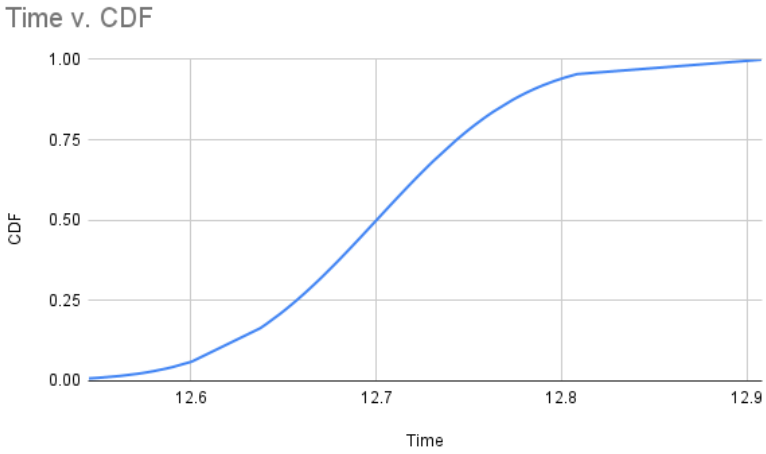


Slowest Patterns:

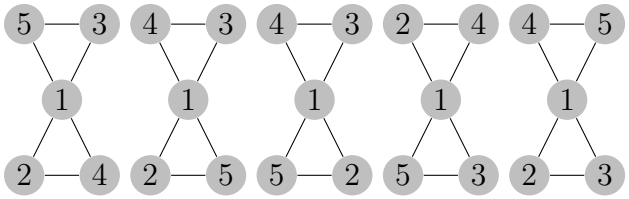


3.1.8 Hourglass

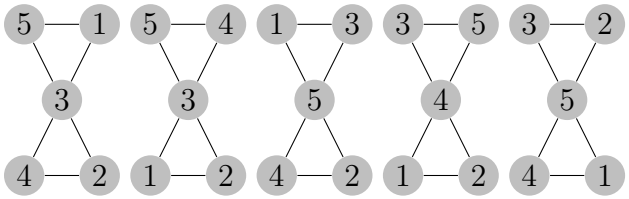
Time		**RSS Memory**		**Dirty Memory**	
Diff.	9.083564174%	Diff.	9.964173757%	Diff.	2.896829351%
Min	12.54475	Min	17328	Min	13398
Max	12.90815	Max	18902	Max	14733
Mean	12.70033669	Mean	18335.09917	Mean	14186.06612
Std. dev.	0.06407778739	Std. dev.	249.1605709	Std. dev.	237.4201878



Fastest Patterns:

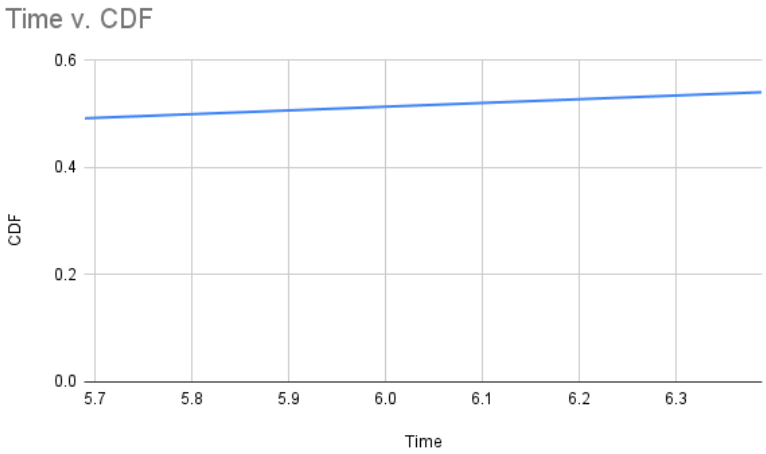


Slowest Patterns:

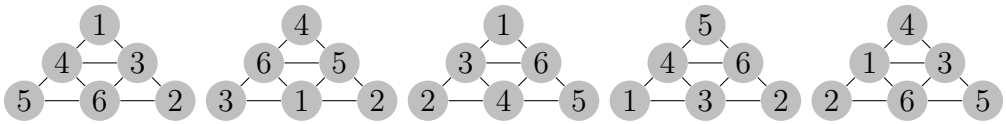


3.1.9 Triforce

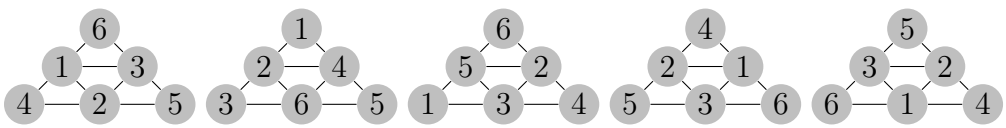
Time		**RSS Memory**		**Dirty Memory**	
Diff.	12.30943119%	Diff.	17.53926702%	Diff.	18.89168766%
Min	5.68889	Min	16426	Min	12704
Max	6.38916	Max	19307	Max	15104
Mean	5.741784189	Mean	18305.26075	Mean	14166.78225
Std. dev.	0.06195568222	Std. dev.	423.3537839	Std. dev.	407.939124



Fastest Patterns:

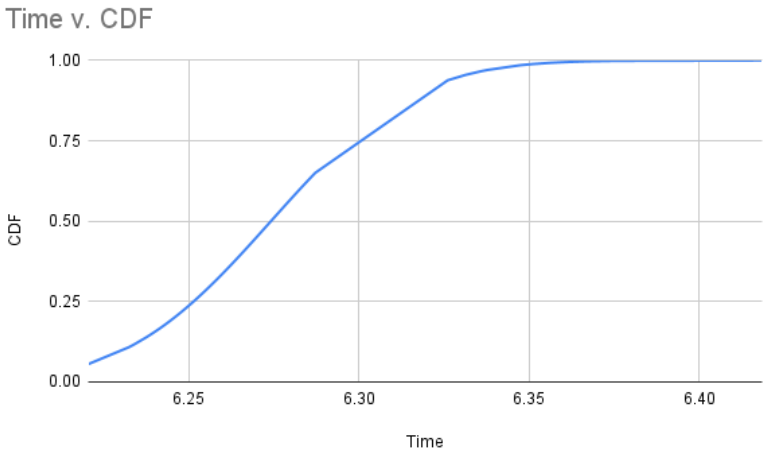


Slowest Patterns:

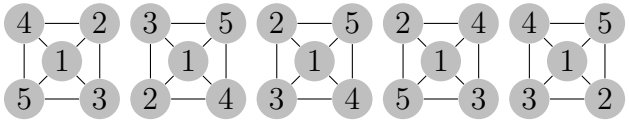


3.1.10 xBox

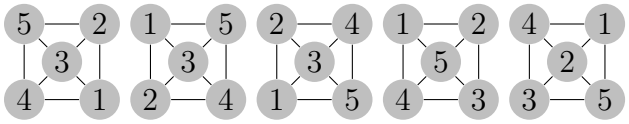
Time		**RSS Memory**		**Dirty Memory**	
Diff.	3.182426882%	Diff.	16.34726532%	Diff.	19.09199522%
Min	6.22041	Min	16437	Min	12555
Max	6.41837	Max	19124	Max	14952
Mean	6.274158182	Mean	18005.38017	Mean	13895.03306
Std. dev.	0.03372494645	Std. dev.	609.3543339	Std. dev.	537.8263495



Fastest Patterns:



Slowest Patterns:



3.2 Discussion

GraphPi appears to have addressed the discrepancy in execution time. Unlike Peregrine's, all CDF graphs of GraphPi do not have spikes. The difference in range is also negligible (e.g., 2-10%). Interestingly, GraphPi does have large discrepancies in memory consumption, which calls for future investigation.

4. Future Work

The preliminary result confirms the existence of the problem in graph mining systems. There are multiple extensions for this work, some of which are:

1. To make the study more comprehensive, experiments on more systems and input graphs are required. With the current setups, we do not anticipate any significant challenges here.

2. Unfortunately we have not identified the root cause of this phenomenon. This requires an in-depth investigation, potentially specialized for each target system. However, because all systems share a common step of generating a pattern-matching plan, we speculate this is the root cause. It is reasonable to assume if we optimize this matching plan (e.g., by building a cost model), we could alleviate the issue, avoiding worst-case performance.
3. Orthogonally, with the abundance of empirical data, it would be of interest to train an ML model to predict the performance of a variant and recommend (or automatically decide) a better variant to the users.

Acknowledgement This study is supported by an NSA grant D9104-S6

References

[1] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow.*, 7(7), 2014.

[2] K. Jamshidi, R. Mahadasa, and K. Vora. Peregrine: A pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[3] T. Shi, M. Zhai, Y. Xu, and J. Zhai. GraphPi: High performance graph pattern matching through effective redundancy elimination. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.