Graph Theory Project Report: An Inclusion-Exclusion Algorithm On Chromatic Number

Ziyue Wang

Computer Science
University of Manitoba
Winter 2018

Contents

Abstract

This report summarize the works contributed in COMP 7750 - Graph Algorithms in Winter 2018. In this project I implemented an inclusion-exclusion algorithm for chromatic number according to Björklund and Husfeldt [3] in C++ and evaluate the algorithm on various results.

1 Introduction

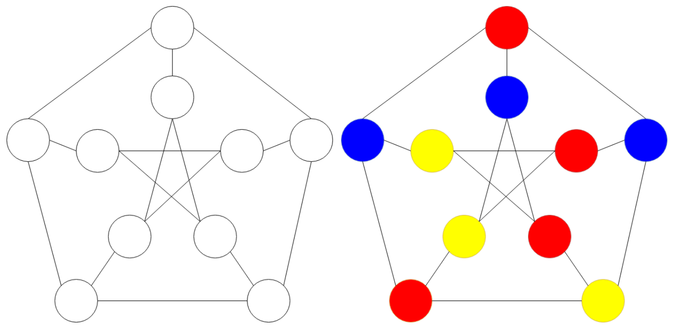
1.1 Graph Colouring

Graph colouring problems are problems in graph theory and combinatorial mathematics. The problems are usually defined as graph partition problems. Normally we have vertex colouring and edge colouring problems. The problems try to partition vertices or edges on a give graph into k independent sets, each set is given a different colour. To find the minimum k partitions of vertices, we define it as chromatic number problem, to find the minimum number of k edge partition, it is usually called chromatic index. There are also some other problems, such as chromatic polynomial is to find the number of vertex k-colouring of a graph, the focused problem in this project is chromatic number.

1.2 Chromatic Number

Chromatic number finds the minimum number of colours so that we can colouring the vertices of a graph into different colours, and no adjacency vertices have same colour. To find a chromatic number is NP-hard for any non-2 colourable graph, which means there is no algorithm can find the optimal solution efficiently. For 2-colourability, we can easily come with a polynomial algorithm to solve that. In order to find the chromatic number, there are some existing algorithms to solve that, what I will try is to use an inclusion-exclusion algorithm that was introduced by Björklund and Husfeldt [3] to solve the problem.

One of the famous graph is Petersen graph. It is a 3-regular graph with 10 vertices. Petersen graph has chromatic number 3, which means we can find a valid 3-colouring of Petersen graph.

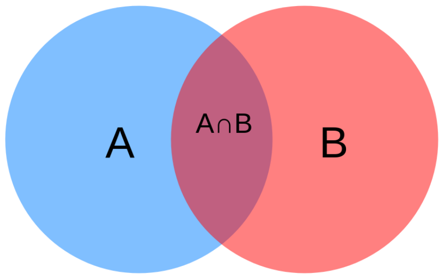


Petersen Graph

The above example is a 3-colouring of Petersen graph. 3 colours: red, blue and yellow are used to colour the graph. No vertex has same colour with it's neighbors.

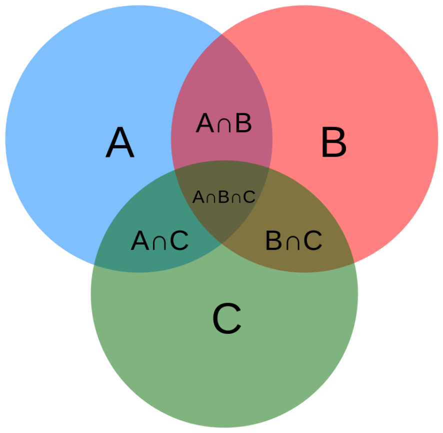
1.3 Inclusion-exclusion

Inclusion-exclusion algorithms are usually designed for complex counting problems. There some existing inclusion-exclusion algorithms apply to other graph problems such as hamiltonian path problem [6, 1]. Inclusion-exclusion algorithms are designed based on inclusion-exclusion principle.



$$|A \cup B| = |A| + |B| - |A \cap B| \tag{1}$$

The above formula 1 is an example of inclusion-exclusion principle with two sets, A and B. To get the size of A and B union, we merge the sets, and remove the duplicate items. This idea could be further extended to more sets.



$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C| \tag{2}$$

The above formula 2 is another example of inclusion-exclusion formula for three sets. Notice here after merge all sets, we subtract the number of overlapping items of each two sets, but some of them are subtracted multiply times, therefore we need to add them back. Follow this idea, we can keep merge sets, subtract the number of duplicated items, and add the number of items that are subtracted more than once. Repeatedly doing this until we reach the universal set.

$$X = \sum_{W \subseteq \{1,2,\ldots,n\}} (-1)^{|W|} N(W) \tag{3}$$

The above formula 3 is the closed formula for inclusion-exclusion principle with n items. Let's say we have an universal set U, contains item 1,2,...,n. W is all possible subsets of U, $N(W)$ is the number of items in $U - W$. The similar condition holds for any inclusion-exclusion algorithm. However, instead of sets, people usually define sets as a collection of properties, therefore $N(W)$ will be the number of items having none of the properties w, where $w \in W$ [6].

To extend inclusion-exclusion principle to an algorithm, the critical part is to pick a suitable $N(W)$ and its properties.

2 Related Works

Inclusion-exclusion algorithms are really suitable for count problems. Independent sets problem could be a good example. The problem is about to find the minimum number of independent subsets to cover the universal set [3]. The problem could be reduced to chromatic number problem which we will show later [6]. Inclusion-exclusion algorithms are also used for counting problems in graph theory. Such as Hamiltonian path problem [1] and counting subgraph isomorphisms [7].

2-colouring of a graph could be solve in polynomial time, 3-colouring is complex but still not that complexity, it could be solved in $O(1.3289^n)$ [2]. However, for any $k > 3$, to find the optimal solution is really challenging. A very early algorithm for chromatic number problem was introduced by Lawler [10], which has complexity of $O(2.4423^n)$. After many years, Eppstein [5] used idea from minimum independent sets to solve the problem in $O(2.4151^n)$ time. Byskov [4] improved the minimum independent sets algorithm for graph colouring problem, and reduce the time cost to $O(2.4023^n)$.

The inclusion-exclusion algorithm for chromatic number problem was invented by Björklund and Husfeldt [3]. They introduced an algorithm that uses inclusion-exclusion principle solves chromatic number problem of complexity of $O(n \times 2^n)$ with exponential space, and $O(2.2461^n)$ with polynomial space. The algorithm is also applied to minimum independent sets problem [4]. This upon reduce the complexity from previous best known complexity $O(2.4023^n)$ [4] to $O(2.2461^n)$ with only polynomial space. [11, 3].

In same year, Koivisto has developed an inclusion-exclusion algorithm for graph partition problems by using other techniques which also could be applied to graph colouring problem [9]. The paper generally introduced the algorithm concepts and give some examples on graph partition problems, in addition, Koivisto also claimed that their algorithm applies to graph coloring with details omitted. Koivisto's algorithm is more efficient in terms of complexity - $O*(2^n)$, compared with $O(n \times 2^n)$ by Björklund and Husfeldt[3, 9].

3 The Algorithms

3.1 Minimum Independent Sets

Before we touch the chromatic number problem, we could gain idea from independent set problem and see how is it reduced to chromatic number problem.

Independent set problem is the problem that tries to find a group of subsets $s \in S$, where S is the set off all subsets in the universal set U, the union of those subsets will cover U. We are given U and S as parameters. Let's look at an example. We have $U = \{a, b, c, d, e\}$, and $S = \{\{a, b, c\}, \{b, d\}, \{c, d\}, \{d, e\}\}$. A valid set cover could be $\{\{a, b, c\}, \{d, e\}\}$ or $\{\{a, b, c\}, \{b, d\}, \{d, e\}\}$. In this case $\{\{a, b, c\}, \{d, e\}\}$ will be the minimum independent sets.

To reduce minimum independent sets problem to graph colouring problem, we using the input graph G as the universal set U, and all the independent vertices sets I to be the subsets S. Thus the input would be G and S. Therefore a k-cover of U and S will become a k-partition of G and I. Sincere the sets in I are all independent, by assigning a colour to each set, we could get a valid k-colouring. We count the number of k-colouring of G, if there is at least one possible solution, we could conclude G is k-colourable. We have shown the reduction from minimum independent sets to chromatic number problem, therefore the algorithms used for minimum independent set problem could be used for chromatic number [6, 3].

3.2 Inclusion-exclusion Algorithm

There is an inclusion-exclusion algorithm could be applied to both minimum independent sets problem and chromatic number problem. By using the idea of inclusion-exclusion principle. We are going to show that the number of ways to pick k sets from I could be computed by following formula.

$$c_k = \sum_{W \subseteq G} (-1)^{|W|} s(W)^k \tag{4}$$

First of all, we define $s(W)^k$ to be the number of ways to choose k sets in I from subgraph $G - W$ with replacement. And for the sets we pick, they could be same sets. Therefore, to pick k items in $I \in G - W$, we have $s(W)^k$ ways to do that.

To compute $s(W)$, we can do this recursively. For easiness, we donate $g(X) = s(G - W)$. We can easily conclude that $g(\emptyset) = 0$ since empty graph has no subset. We randomly (or by the order) pick one vertex v in G, consider it as an independent set itself, in the remaining graph, there are $g(G - v)$ number of independent sets. On the other hand, if v is considered as a part of an independent set, all its neighbors have to be in other sets, which is $g(G - v - N(v))$, where $N(v)$ is the set of v's neighbors. In addition, there is an empty set. Thus could get a recursive function.

$$g(G) = g(G - v) + g(G - v - N(v)) + 1 \tag{5}$$

By recursively compute the value for all subgraphs, we could get a recursive formula.

$$g(X) = g(X - v) + g(X - v - N(v)) + 1 \tag{6}$$

Which is equivalent to the following formula in $s(W)$ representation.

$$s(W) = s(W \cup v) + s(W \cup v \cup N(v)) + 1 \tag{7}$$

By the recursive function, we compute $s(W)^k$ for all $W \in G$ by given k. Compute the ways of pick k independent sets c_k by formula 4. If we obtain $c_k > 0$, it means that it is possible to partition G into k independent sets, which gives a valid k-colouring of G. To find the minimum k for $c_k > 0$, we could do a binary search of k. The pesudocode is given as following.

Algorithm 1: Find the chromatic number ($NumColour(G)$)

```
Input: Input Graph $G$
Output: Chromatic Number $k$
foreach subgraph $W \subset G$ do
    $s[W] = S[W + v] + S[W + v + N(v)] + 1$;
end
while binary search $k$ do
    $sum = 0$;
    foreach subgraph $W \subset G$ do
        $sum += -1^{|W|} * s[W]^k$
    end
    if $sum > 0$ then
        Return $k$;
    end
end
```

Notice the algorithm does find the k-colouring of the graph. But once we found the minimum k for the graph, we can extend the algorithm to find a valid colouring. The algorithm is able to extend to find the chromatic polynomial. There two extensions are excluded in this project due to the time limitation.

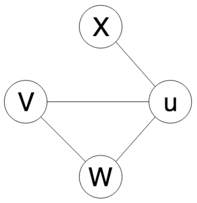
3.3 Complexity

The algorithm computes $s(W)$ for all subsets of G. Assume there are n vertices in G, we will have 2^n of $s(W)$ values. If the exponential space is allowed, by the idea of dynamic programming of recursion, we can store all $s(W)$ in an array. Since there are 2^n of $s(W)$, the space requirement will be exponential.

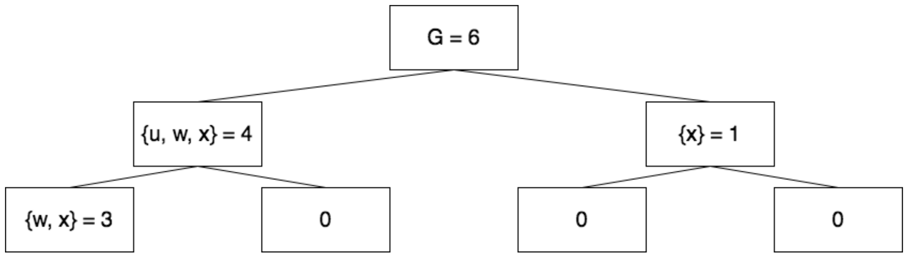
To compute each $s(W)$, since we are using dynamic programming, it is just an addition of 3 values. However, depending on the subset generate strategy, it might take $O(\log n)$ time to generate the subset. No matter what strategy we choose, it won't be more than polynomial time. Thus for each $s(W)$, we require $O(n^{O(1)})$ time. We need to compute 2^n of $s(W)$ and sum them up, therefore the dominating complexity will be $O(2^n)$ for some given k. To find the minimum k, we perform a binary search, which takes $O(\log n)$ time, since n vertices must be n-colourable. The final complexity we get for exponential space is $O(n^{O(1)} \times 2^n \times \log n) \in O(n^{O(1)} \times 2^n)$.

3.4 Example Walk-though

This section will show an example how the algorithm works. The example was take from Husfeldt [8]. Let's look at graph G has 4 vertices.



After we read the graph, the next we are going to do is compute all $s(W)$ without consider the k. There are 4 vertices in this example, therefore we have $2^4 = 16$ entries. To compute each one of them, we use a recursive function.



The above tree shows a recursion of G. Let's say we remove v from G, the remaining graph will be $\{u, w, x\}$. And since u and w are adjacent to v, the other recursion is $\{x\}$. Besides them, add an empty set to them. To compute $\{x\}$, we can easily find both of its recursion is 0, so $\{x\} = 0 + 0 + 1 = 1$. For $\{u, w, x\}$, we need more recursion. Let's say we remove u from subgraph $\{u, w, x\}$. Since all others vertices in this subgraph adjacent to u, we found this is one empty set, which is 0. On the other hand, we still have $\{w, x\}$ left. Follow this idea, we could find $\{w, x\} = 3$. $\{u, w, x\} = 4$, and finally $G = 6$. This is an example of top-down recursion. However, in practical, since we need all possible subsets, we will use a down-top approach, but following same recursion idea.

Table 1: Example

X	g	k=2	k=3
$\emptyset$	0	0	0
{u}	1	-1	-1
{v}	1	-1	-1
{w}	1	-1	-1
{x}	1	-1	-1
{u, v}	2	4	8
{u, w}	2	4	8
{u, x}	2	4	8
{v, w}	2	4	8
{v, x}	3	9	27
{w, x}	3	9	27
{u, v, w}	3	-9	-27
{u, v, x}	4	-16	-64
{u, w, x}	4	-16	-64
{v, w, x}	5	-25	-125
{G}	6	36	216

After compute all $g(X)$, we get table 1. If we look at original graph directly, we can easily find there is a triangle, and therefore it is not 2-colourable. Let's try $k = 2$ in this example. By applying the formula, we get a value for each subset. Sum up them, we found $c_2 = 0$, which means the graph is no 2-colourable. So let's try $k = 3$. By applying the exact formula, we could $c_3 = 18 > 0$, thus the graph is 3-colourable. For large graph, we will use a binary search to find the k.

4 Experimental Result

4.1 Benchmarks

The hardware I am using for evaluate the datasets is the Linux machine cluster at University of Manitoba. Everyone has a computer science account at University of Manitoba can access these machines. Table 2 shows the details of the hardware. The programming language is C++ with g++ complier version 4.8.5. For the datasets, I will test various graphs with different number of edges and vertices. Let K_n donate the complete graph of n, $L(G)$ be the line graph of some G, and C_n donate a cycle of length n.

4.2 Results

The table 3 shows the result of a couple of graphs.

Table 2: Hardware and System Settings

CPU:	Intel(R) Pentium(R) CPU 1403 @ 2.60GHz * 2
Architecture	x86_64
Memory	7927636 kB
System	Scientific linux 7

Table 3: Results

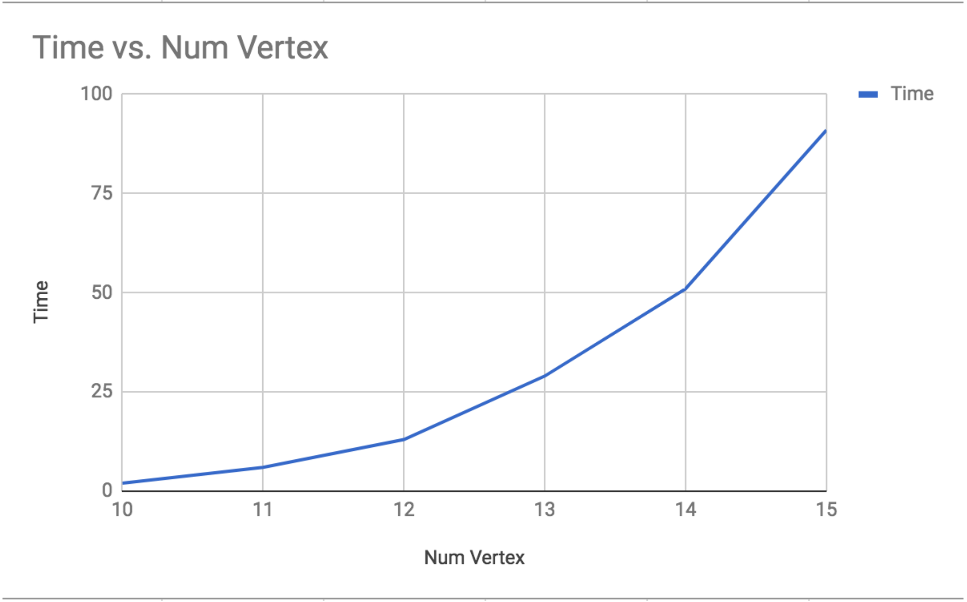
Graph	Number Vertices	Chromatic Number	Time (milliseconds)
C10	10	2	2
K10	10	10	2
Petersen	10	3	2
L(K(5))	10	5	2
C11	11	3	6
K12	12	12	13
C13	13	3	29
K14	14	14	51
K15	15	15	91
L(K(6))	15	5	98
L(K(6))-Match	15	5	98

4.3 Results Analysis

Before really touching the table, there are some other observations I found. First of all, when a graph has less number of edges, there will be more independent sets, therefore the value of $s(W)$ will be large. A complete graph has lowest value on each $s(W)$. The second observation is when $s(W)$ approaches to very large, normal built-in data type overflow. For example, $C14$ graph will overflow when we use long long type in C++ which is 64 bits int. To accept further larger values, boost library supports 128 and 256 bits int. However, to work with extremely larger values, a string based int is recommended and waiting for test. However, use a string based int will lower the performance of the algorithm. Due to this problem, the algorithm is better for the graphs have more number of edges.

By looking at two groups, $\{C10, K10, Petersen, L(K(5))\}$ and $\{K15, L(K(6)), L(K(6))-Match\}$, each graph has same number of vertices, most of them have different number of edges. By looking at the time costs, we can see that complete graph is a little better than others. This is due to that we evaluate all subsets, so the complexity of the graph does not affect the amount of computation. On the other hand, the smaller number of edges there is, the higher $s(W)$ value is. The computation cost of a larger number is higher. Thus a complexity graph (e.g. complete graph) is more efficient than a simple graph (e.g. cycle graph). This is the other reason that the algorithm is better for the graphs have more number of edges.

With in each group, although the performance is a little different, it is not that sufficient when we compare with different groups. $L(K(6))$ and $K15$ have same vertices, their computation time is only 7 milliseconds different. K15 is one more vertex than K14, the computation time is almost doubled. This is due with the increasing number of vertices, the number of subsets is exponentially increased.



The above chart shows the exponential increasing time cost of the algorithm. By increasing the number of vertices by 1, the time cost is about double every time. This is reasonable for a exponential complexity $O(n^{O(1)} \times 2^n)$. Although we could string based ints to solve the overflow problem, use non-dynamic programming to avoid exponential space requirement, at some point the time cost will be extremely high and we could not get the result in reasonable time.

5 Conclusion

In conclusion, we have shown an algorithm that solves chromatic problem problem by using inclusion-exclusion principle. The algorithm is introduced by Björklund and Husfeldt [3]. The algorithm beat down the best known algorithm at that time by reducing the complexity from $O(2.4023^n)$ to $O(n^{O(1)} \times 2^n)$.

The algorithm is introduced by reducing minimum independent sets problem to chromatic number problem. The algorithm find the number of ways to colour the graph by counting the number of ways to pick k sets in each subgraph, and sum them all. If there is a least one way to k-colour the graph, we can say graph is k colourable. A binary search is used to find minimum k. Each subgraph takes polynomial time to compute by using dynamic programming technique, and there are 2^n subgraphs. By multiplying a factor $O(\log n)$ for binary search into the complexity, we found the final complexity is $O(n^{O(1)} \times 2^n)$ with exponential space requirement.

After evaluate the results, we can see that the algorithm is more suitable for complex graphs. This is due the complex graphs has lower number of independent sets in each subgraph, therefore it is less likely overflow and the int addition operation time is less. The time cost of the algorithm is exponentially increasing both theoretically and practically. In theory, we have an exponential complexity of $O(n^{O(1)} \times 2^n)$, and same in practice, the time cost is about doubled when we increase number of vertices by 1.

6 Future Works

As the potential revisit of this project in the future, I will write some notes of potential future works and the problem I found here.

1. There are many algorithms could generate subset efficiently. Although it only effects the polynomial part of complexity, it is desire to research which one is the best in this case, since we are not generating subsets but subgraphs.

2. If we do not have exponential space available, the algorithm still works with a higher complexity, this could be evaluated as well.

3. Only a little modification is needed to find the chromatic polynomial of a graph, this could be an extension of the project.

4. I only used long long type to store the values, how does the string based int affect the performance?

5. By using string based int, it is possible to evaluate the graphs have larger size. In my experiment, I only used some small graphs and the time costs are low.

6. Koivisto [9] improved the complexity of algorithm from $O(n^{O(1)} \times 2^n)$ to $O(2^n)$. Details are omitted in their paper. This could be another extension of the project.

7. Consider of my specialization, I should take parallel computing into this project. Recursion is not each for MPI and not possible for CUDA, therefore I could start with OpenMP-task.

There are so many stuffs still desire to research and try on this topic.

References

[1] Eric T Bax. “Inclusion and exclusion algorithm for the Hamiltonian path problem”. In: *Information Processing Letters* 47.4 (1993), pp. 203–207.

[2] Richard Beigel and David Eppstein. “3-coloring in time O (1.3289 n)”. In: *Journal of Algorithms* 54.2 (2005), pp. 168–204.

[3] Andreas Björklund and Thore Husfeldt. “Inclusion–exclusion based algorithms for graph colouring”. In: *Electronic Colloquium on Computational Complexity (ECCC).* Vol. 13. 044. Citeseer. 2006.

[4] Jesper Makholm Byskov. “Enumerating maximal independent sets with applications to graph colouring”. In: *Operations Research Letters* 32.6 (2004), pp. 547–556.

[5] David Eppstein. “Small maximal independent sets and faster exact graph coloring”. In: *J. Graph Algorithms Appl.* 7.2 (2003), pp. 131–140.

[6] Fedor V Fomin and Dieter Kratsch. *Exact exponential algorithms.* Springer Science & Business Media, 2010.

[7] Zachary Frenette. “Parameterized Algorithms for the Subgraph Isomorphism Problem”. In: (2014).

[8] Thore Husfeldt. “Graph colouring algorithms”. In: *arXiv preprint arXiv:1505.05825* (2015).

[9] Mikko Koivisto. “An O*(2ˆ n) Algorithm for Graph Coloring and Other Partitioning Problems via Inclusion–Exclusion”. In: *Foundations of Computer Science, 2006. FOCS’06. 47th Annual IEEE Symposium on.* IEEE. 2006, pp. 583–590.

[10] Eugene L Lawler. “A note on the complexity of the chromatic number problem”. In: *Information Processing Letters* 5.3 (1976), pp. 66–67.

[11] Gerhard J Woeginger. “Exact algorithms for NP-hard problems: A survey”. In: *Lecture notes in computer science* 2570.2003 (2003), pp. 185–207.