Comparative Analysis of C/C++, Java, Python, and LISP Implementations of Greedy Algorithms for the Graph Coloring Problem

Student authors: Nenad Mančević[1], Igor Mihajlović[1], Nenad Andrejević[1], and Milan Đokić[1]
Mentors: Radomir Stanković[2] and Dušan Gajić[2]

Abstract – **Graph coloring is a very interesting NP-complete problem. Coloring of a simple graph is the assignment of a color to each vertex of the graph so that no two adjacent vertices are assigned the same color. It is interesting to compare performances of different variations of the greedy algorithm for graph coloring depending on parameters and vertex order. Also, it is interesting to determine how much the implementation in different programming languages affects performances of the algorithm. In this paper, we present implementations of three variations of greedy algorithm for graph coloring in several programming languages including C/C++, Java, Lisp, and Python, as well as results and conclusions drawn from experiments. These results could be helpful in tracing directions for further research towards efficient implementation of various algorithms for NP-complete problems.**

Keywords – **greedy algorithm, graph coloring problem, NP-complete problems, programming languages.**

I. Introduction

Graph theory, as one the most important fields in discrete mathematics, has many applications in modern computer science. During its rich evolution, graph theory has produced great deal of many interesting problems. Many of those problems are NP-complete, including one of the most popular problems – graph coloring.

In graph theory, we can define a vertex coloring of a graph G = (V,E) as a map c: V → S such that $c(v) \neq c(w)$ whenever *v* and *w* are adjacent. The elements of the set S are called the available *colors* [1]. The minimum number of unique colors that we could use to paint graph vertices is called a *Chromatic number* χ(G). Similarly, the minimum number of unique colors used in coloring the edges so that two adjacent edges don't share the same color gives a *Chromatic index*.

In this paper, we will focus on vertex coloring problem which belongs to a class of NP-complete problems.

Since this problem is NP-complete, many algorithms have been proposed to obtain approximate colorings in reasonable time. Those algorithms can be categorized in the following classes: greedy, partition, clique, Zykov and others [2].

In the following sections we will further present three variations from a class of Greedy algorithms. We will later discuss their software implementations in four different programming languages including C/C++, Java, Python and LISP. Finally we will show experimental results, obtained by running these implementations on random generated graphs, presenting time efficiency over different programming languages and computed chromatic number using different algorithms.

Student authors:
[1]Nenad Mančević, Igor Mihajlović, Nenad Andrejević, and Milan Đokić are with the Faculty of Electronic Engineering, Aleksandra Medvedeva 14, 18000 Nis, Serbia, E-mails: manca@elfak.rs, igor.mihajlovic1987@gmail.com, neca.87@gmail.com, Milan.djokic.87@gmail.com .
Mentors:
[2]Radomir Stanković and Dušan Gajić are with the Faculty of Electronic Engineering, Aleksandra Medvedeva 14, 18000 Nis, Serbia, E-mails: radomir.stankovic@gmail.com, dusan.gajic@elfak.ni.ac.rs.

II. Greedy Algorithms

One of the simplest methods for coloring a graph is by using a greedy approach. This approach consists of the following:

Given a graph G = (V,E) and a fixed vertex enumeration $a_0, a_1, \ldots, a_n$:

$c(a_0) = 1$

If $a_1, \ldots, a_{i-1}$ ($i \geq 1$) have already received colors, let $c(a_i)$ be the smallest color not yet used in the neighborhood of a_i.

It could be shown that the number of colors used hugely depends on the order of vertices. There are many heuristic techniques for Greedy coloring. We will now present two different methods for vertex ordering that could yield good results in terms of chromatic number.

A. Largest degree ordering

Degree based ordering is one of the easiest methods for coloring a graph. It provides a slightly better strategy than the algorithm provided above which simply picks a vertex from an arbitrary order [3].

Largest degree based ordering chooses a vertex with the highest number of neighbors. Initially graph vertices are sorted in a non-increasing order according to their degree after which former algorithm is applied. This approach produces better chromatic number but it's time consumable.

B. Saturation degree ordering

The algorithm DSATUR (Degree of Saturation) of Brelez [3] is a sequential coloring algorithm with a dynamically

established order of the vertices. The degree of saturation of a vertex *u*, $deg_c(u)$, is the number of different colors at the vertices adjacent to *u*. This algorithm at *i*-th step chooses the not yet colored vertex with the largest degree of saturation. Since degrees change dynamically through algorithm iterations, we can conclude that it is even more time consumable than the previous one.

III. Software Implementations

We implemented above discussed algorithms using four different programming languages. We will now present a short pseudo-code [4] that is the basis for each of them and then show particular implementation techniques in each of the programming languages.

```
Color(V,E):
1. color[v[0]] = 1
2. for i ← 1 to |V|
3.    do  ColorsTaken [] ← 0
4.    for each u ∈ V
5.        do if (u,V[i]) ∈ E
6.             ColorsTaken[color[u]] ←1
7.    k ← 1
8.   while ColorsTaken[k]=1
9.        do k ←k+1
10   color[v[i]] = k
```

In line one first color is assigned to the first vertex. Lines 2-10 algorithm iterates through the rest of the vertices. In line 3 ColorsTaken for current vertex is reseted to zero. In lines 4-6 neighbors list of current vertex is explored and colors assigned to each of them are marked as taken. In lines 7 - 9 first non-used color is chosen and assigned to current vertex in line 10.

Depending on the variation of the algorithm used input set of vertices V is sorted accordingly.

A. C++ and Java implementations

An array of adjacency lists is used to represent graph G.[5]The i^{th} adjacency list is integer array of size *d(i)*,which is degree of vertex *i*, where each entry represents vertex adjacent to *i*. Above given algorithm is used to color vertices. Merge-sort [5] is used for sorting vertices in a non-increasing order according to their degree. In DSATUR algorithm [3] ColorsTaken matrix is used instead of array in former algorithm where ColorsTaken(*i*,*j*) is equal to 1 if and only if node *i* has neighbor colored with color *j*. In this way both the smallest available color for vertex *i*, and its degree of saturation could be easily calculated. For purpose of testing random number generator is created which generates random graphs with given number of vertices and edges.

Same logic was used for implementation of the above algorithm in Java. To use all the benefits that Java offer, adjacency list is represented using built-in data structure ArrayList, encapsulated in class AdjacentList, which has easy methods for adding and removing element from the list. For vertices sorting Heap-sort was used [5]. Tests were performed on same random generated graph parsed to fit input format of given implementation.

B. LISP implementation

List of vertices is used to represent graph G. Vertices are represented as a list, each containing vertex ID, its color and list of adjacent vertices IDs. Algorithm given above is used to color vertices. To sort vertices according to their degree Merge-sort [5] is used. In its modified version, it sorts vertices by their degree of saturation to determinate a vertex that will be colored next in DSATUR algorithm [3].

C. Python implementation

For this implementation we used already existing library for graph manipulation – *networkx* [6]. Adding set of edges to the Graph structure creates graph. We use built-in methods for manipulation with set of vertices. The aforementioned algorithm in its adapted version is used to color vertices. To sort the vertices, built-in quick-sort algorithm is used.

IV. Experimental Results

We will now show obtained results from above mentioned implementations. For purpose of testing random graph generator was created which generates random graphs for a given number of vertices and edges.

It is worth mentioning that the performance in terms of chromatic number of the implemented algorithms will vary due to different sorting algorithms used in different programming languages.

In the next four tables we will show running times for C++, Java, Python and LISP implementations considering different algorithms and graph sizes, respectively. Finally the comparison between the average chromatic numbers will be shown, as well as the best performed implementation.

TABLE I
Running time [ms] of different algorithms on various graph size using C++ implementation

Number of Vertices	Number of Edges	Naïve Greedy	LDO	DSATUR
1000	10000	0	0	16
1000	50000	0	4	16
1000	100000	0	8	18
5000	50000	0	8	162
5000	100000	0	16	172
5000	500000	31	32	193
5000	10^6	62	78	246
10000	10^5	15	31	671
10000	$5*10^5$	78	83	718
10000	10^6	125	140	796
10000	10^7	1373	1382	2718
20000	$2*10^5$	47	78	2527
20000	10^6	219	254	2543
20000	$2*10^6$	421	468	2730
20000	10^7	2122	2169	4633
20000	$2*10^7$	4228	4290	6254

As it could be seen from the results running time of Naïve Greedy varies a little from LDO while DSATUR has far worse performance. On the other hand Naïve and LDO much more depend on the number of edges than the DSATUR.

In Table II the running time results are given considering the same test input on the same graphs as for the previous implementation, but in Java.

TABLE II
RUNNING TIME [MS] OF DIFFERENT ALGORITHMS ON VARIOUS GRAPH SIZE USING JAVA IMPLEMENTATION

Number of Vertices	Number of Edges	Naïve Greedy	LDO	DSATUR
1000	10000	52	53	75
1000	50000	30	39	41
1000	100000	31	38	42
5000	50000	33	47	325
5000	100000	34	58	320
5000	500000	48	69	391
5000	10^6	59	76	436
10000	10^5	71	91	1203
10000	$5*10^5$	86	106	1301
10000	10^6	88	112	1430
10000	10^7	213	245	NA
20000	$2*10^5$	221	248	NA
20000	10^6	255	285	NA
20000	$2*10^6$	257	278	NA
20000	10^7	369	417	NA

TABLE III
RUNNING TIME [MS] OF DIFFERENT ALGORITHMS ON VARIOUS GRAPH SIZE USING PYTHON IMPLEMENTATION

Number of Vertices	Number of Edges	Naïve Greedy	LDO	DSATUR
500	5000	6	15	113
500	10000	12	12	120
500	25000	26	27	128
1000	50000	57	59	480
1000	100000	119	122	521
5000	50000	173	176	11169
5000	10^5	223	229	11572
5000	$5*10^5$	710	731	11613
5000	10^6	1520	1580	12077
10000	10^5	590	588	45161
10000	$5*10^5$	1156	1056	45480
10000	10^6	1794	1699	45766
20000	10^6	4011	4034	NA
20000	$2*10^6$	5415	7012	NA

From the experimental results in Java implementation we can notice that the running time is not that much worse than in C++. For example, for 20000 vertices and 10^7 edges give much better running time in Java than in C++. However, Java cannot compute chromatic number using DSATUR algorithm for more than 10000 vertices and 10^6 edges due to memory limitations.

TABLE IV
RUNNING TIME [MS] OF DIFFERENT ALGORITHMS ON VARIOUS GRAPH SIZE USING LISP IMPLEMENTATION

Number of Vertices	Number of Edges	Naïve Greedy	LDO	DSATUR
500	20000	244	295	400
500	28000	442	493	672
1000	30000	623	661	1029
2000	11000	420	428	537
2000	25000	1101	932	1588
5000	33000	3342	3504	4760
10000	25000	7492	9250	10656

On the other hand Python's implementation using already built-in library performs considerable worse than Java and C++. Maximum number of vertices and edges that this implementation could handle on our test system was 20000 and 10^6, respectively. Also, the same rule applies as for C++ and Java implementations that the first two algorithms perform much faster than the DSATUR algorithm.

Finally, LISP implementation gives the worst results. Although it has dynamic typing as Python, it performed much worse due to its recursive structure. That implies vertex number limitation for our tests.

In Table V we show the average chromatic number obtained from all three implementations for given test results, performed on all three algorithms used.

TABLE V
AVERAGE CHROMATIC NUMBER FOR DIFFERENT IMPLEMENTATIONS ON THREE ALGORITHMS

Number of Vertices	Number of Edges	Naïve Greedy	LDO	DSATUR
500	5000	11	7.6	7.6
500	10000	17	11.5	11.5
500	25000	31.5	21.8	22.5
1000	50000	31.6	30.3	29.1
1000	100000	54	51.3	51
5000	50000	11.33	10.8	9.8
5000	10^5	17	15.6	15.5
5000	$5*10^5$	51	49.3	47.8
5000	10^6	87.6	85	83.3
10000	10^5	11.8	10.5	10
10000	$5*10^5$	33	29.6	28.6
10000	10^6	51.3	48.3	47.6
20000	10^6	32	30	NA
20000	$2*10^6$	52	48.6	NA

From the Table V we observe that the LDO algorithm performs the best comparing the runing time seen in Tables I, II, III and IV. DSATUR algorithm is more time consuming and does not provide the expected results. Hence, we can state that from our tests LDO is recommended algorithm for greedy graph coloring.

V. Conclusion

In this paper we presented a comparative analysis of four different programming language implementations of three variations of a greedy algorithm applied on the graph coloring problem.

First we introduced graph coloring as an important field in modern computer science and discussed different approaches in solving this NP-complete problem. We presented greedy coloring as one of the popular methods that gives good results for most graphs. This paper shows three different variations of basic greedy algorithm: Naïve greedy, largest degree ordering and DSATUR.

Comparing the results obtained from all four implementations, we can notice that C++ and Java gave the best running times for most of our tested graphs, whereas Python and LISP were considerably slower due to their dynamic typing and interpreting nature of program execution. We can conclude that the second algorithm produces the best chromatic number in terms of time consumption for most randomly generated graphs. LDO algorithm is not much slower than the Naïve algorithm because it uses sorting algorithms of $O(n\log n)$ complexity.

References

[1] Reinhard Diestel, *Graph Theory*, 2000.

[2] Joseph C. Culberson, "Iterated Greedy Graph Coloring and the Difficulty Landscape", Technical Report TR 92-07, June 1992.

[3] Walter Klotz, *Graph Coloring Algorithms*, 2002.

[4] James A. Anderson, *Discrete Mathematics with Combinatorics*, 2nd Edition, 2003.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to algorithms*, 2nd Edition, 2001.

[6] Networkx 1.4 Library (http://networkx.lanl.gov/), 10.04.2011.