Algorithm Engineering
WS 22/23

Sanket Datta

Exploring Graph Coloring Algorithms: A Comparative Analysis of Exact, Approximation, and Heuristic Approaches

Abstract

There are four algorithms discussed in this paper for coloring the vertices of a graph such that adjacent vertices have distinct colors. The first algorithm, Largest Degree First (LDF), determines the minimum number of colors required for such a coloring. The second and third algorithms are variations of the randomized greedy algorithm that ensure a proper coloring with at most $\Delta+1$ colors, where Δ is the maximum degree of the graph. Both normal and parallel versions of the randomized greedy algorithm are implemented and analyzed for their effectiveness in achieving a better coloring result. Furthermore, an exact algorithm is presented in this paper that computes the minimum number of colors required to properly color the vertices of a graph using a bounded search tree approach.
According to the experimental findings of this research, by incorporating randomization into the greedy algorithm, the approximation method guarantees a coloring solution with at most $\Delta + 1$ colors, often resulting in superior outcomes. By considering the vertex degree during the implementation of the approximation algorithm, a slightly reduced number of colors can be achieved for certain graphs. In contrast, the exact algorithm utilizes a parameterized approach to determine if the adjacent vertices of the graph can be uniquely colored with a given value, thereby ensuring an optimal coloring solution.

1 Introduction

The graph coloring problem is a well-known problem in graph theory that has practical applications in a variety of fields, including scheduling, register allocation, and channel allocation in wireless communication networks. Given an undirected graph, the graph coloring problem aims to assign a color to each vertex such that no two adjacent vertices have the same color, using the smallest number of colors possible.
In the case of planar graphs, which are graphs that can be drawn on a plane without any two edges crossing, the graph coloring problem is known to be NP-complete. This means that there is no known polynomial-time algorithm that can solve the problem for all instances. However, several heuristics and approximation algorithms have been proposed that can find good solutions in reasonable time.
Graph coloring has been used in various real-world problems, such as scheduling tasks on a processor, assigning radio frequencies to transmitters, mapping cell towers to frequencies, designing timetables for school, bus or train routes, and coloring maps to make them easy to read. It is also applied in other areas such as biochemistry, where graph coloring is used to predict the binding properties of proteins, and computer vision, where graph coloring is used to segment images.
Research on the graph coloring problem has focused on developing efficient algorithms for solving the problem exactly or approximately, as well as exploring the complexity of the problem and identifying structural properties of graphs that can be exploited to find good solutions.
Constraints for research in this area include the difficulty of the problem itself and the large number of possible solutions, as well as the need for efficient algorithms that can handle large-scale instances of the problem. Additionally, finding good solutions often requires a deep understanding of the structural properties of graphs, which can be challenging to uncover.

2 Preliminaries

The goal is to assign a color to each vertex of the graph such that no adjacent vertices have the same color. The total number of colors used in the coloring should be minimized. The graph coloring problem is known to be NP-complete, which means that there is no known polynomial-time algorithm that can solve this problem for all instances. Therefore, researchers have developed various approximation and heuristic algorithms to find a near-optimal solution in a reasonable amount of time. The performance of these algorithms is typically evaluated based on the quality of the solution produced and the time required to compute it.

Problem definition: Given a graph G=(V, E), chromatic number or the number of colors (c_i) required to color the graph is k such that $\forall$(u,v) $\in E$:$c_u \neq c_v$ where i $\leq k$.

3 Algorithm & Implementation

This project implemented various algorithms for solving the graph coloring problem, including the Largest Degree First (LDF) heuristic algorithm, the randomized greedy approximation algorithm (both in normal and parallel versions), and the exact algorithm which finds the optimal solution using bounded search tree.

3.1 Largest Degree First (LDF) heuristic

The Largest Degree First (LDF) algorithm is a popular heuristic approach for the graph coloring problem. The basic idea behind this algorithm is to color the vertices with the highest degree first, as these are the vertices that are most likely to cause conflicts with other vertices.

The algorithm starts by sorting the vertices in descending order of their degree, so that the vertex with the highest degree comes first. This takes O(V log V) time. Then, it colors this vertex with the lowest possible color that is not already used by any of its adjacent vertices. After coloring the first vertex, the algorithm proceeds to the next vertex in the sorted list and colors it using the same procedure. The process continues until all vertices have been colored. Coloring each vertex takes O(deg(V)) time, where deg(V) is the degree of vertex V. In the worst case, deg(V) can be as high as V-1, which would result in a time complexity of O(V^2). The total space complexity of the algorithm is O(V) where we need to store the degree and color of each vertex. However, In terms of the best-case running time, this algorithm has a time complexity of O($|V| * log|V|$) if the graph is a tree or a forest, where $|V|$ is the number of vertices in the graph. This is because the degree of each vertex in a tree is at most 2, so the number of available colors can be computed efficiently using a binary search.

The LDF algorithm is a greedy algorithm and does not guarantee the optimal solution, but it is simple and efficient, and often provides good results in practice. However, it may fail to color the graph with the minimum possible number of colors, especially when the degree of the vertices is not a good indicator of their conflict potential.

Algorithm 1 Largest Degree First (LDF) heuristic

```
Input: Graph g
color = A : None, B : None, ... (Initialize the coloring to all as None)
vertices = sorted vertices by decreasing order of degree
for v in vertices do
    usedcolors = Find the set used colors that have already been used by neighboring vertices, which are already colored.
    availablecolors = Find the set available colors of colors that have not yet been used by any neighboring vertices of v
    color = min(available colors)
    Assign the color to the vertex v in the coloring dictionary.
end for
Return the final coloring dictionary
```

3.2 Randomized greedy algorithm

The randomized greedy algorithm is a popular approach for solving the graph coloring problem. The

algorithm works by assigning colors to vertices in a way that minimizes the likelihood of conflicts with adjacent vertices.The algorithm starts by randomly permuting the order in which the vertices will be considered. Then, for each vertex in the permuted order, the algorithm selects the smallest available color that does not conflict with the colors of its adjacent vertices. If all available colors conflict with adjacent vertices, the algorithm chooses a new color that is not yet used by any adjacent vertex. The approximation quality of the randomized greedy coloring algorithm is similar to that of the greedy coloring algorithm, with a maximum number of colors at most $\Delta + 1$, where Δ is the maximum degree of the graph. However, the randomized greedy coloring algorithm is often more effective at finding good colorings for graphs with irregular or skewed degree distributions.
The time complexity of the shuffle operation is O(V). we iterate through each vertex in vertices, which has a time complexity of O(V). For each vertex, we iterate through its neighbors to find the colors of adjacent vertices. This has a time complexity of O(E), where E is the number of edges in the graph. We then select the smallest available color that does not conflict with the colors of adjacent vertices, which has a time complexity of O(V). Finally we update the color map and available colors which take constant amount of time. Therefore, the total time complexity of the function is $O(V * (E + V))$, which can be simplified to $O(V * E)$. The space complexity of the function is O(V), which is the space used by the two dictionaries colormap and availablecolors and vertices.
The algorithm's performance can vary depending on the graph's density. In particular, the algorithm performs better on sparse graphs, where the number of adjacent vertices for each vertex is low. This increases the number of available colors for each vertex, reducing the number of backtracking and color revisions. As a result, the time complexity of the algorithm decreases, making it more efficient. On average, the time complexity of the algorithm can be approximated by $O(V + ElogE)$ This is because, on average, each vertex has a small number of neighbors, and the available colors for each vertex form a small set. Therefore, the algorithm can quickly find a color for each vertex using a binary search on the available colors.
However, the algorithm may struggle on highly connected graphs, where the number of adjacent vertices for each vertex is high. In such cases, the number of available colors for each vertex is low, reducing the chances of finding a color that does not conflict with the colors of adjacent vertices. This leads to more backtracking and color revisions, increasing the time complexity of the algorithm. The maximum running time of the algorithm can be as high as O(VE), which occurs when the input graph is a complete graph, where every pair of vertices is connected by an edge. In this case, finding a color for each vertex requires $O(V^2)$ time in the worst case, resulting in a total time complexity of O(VE).

Algorithm 2 Randomized greedy algorithm

```
Input: Graph g
colormap←emptyset,vertices←[list of vertices]
availablecolors←each vertex maps to a set of all possible colors
vertices ← Randomly shuffle all vertices
for v in vertices do
    neighborcolors ← Find the colors of all adjacent vertices to v that have already been colored
    colormap[v] ← min(availablecolors[v] - neighborcolors)
    Update colormap to assign the chosen color to vertex v
    Remove the assigned color from availablecolors for vertex v.
end for
return colormap
```

3.3 Parallel Randomized greedy algorithm

This is an implementation of the randomized greedy coloring algorithm in parallel using multiprocessing in Python. The main difference compared to the earlier one is that this one splits the vertices into chunks and processes each chunk in a separate process, and then process them in parallel.
In the parallel version, a multiprocessing manager is

created to share dictionaries between processes. A shared dictionary is used to store the vertex-color mapping, and another shared dictionary is used to store the available colors for each vertex. A lock object is also created for thread safety during dictionary updates to prevent race conditions.
The code first checks if the number of processes specified is greater than or equal to 1. If it is greater than or equal to 1, the vertices are split into chunks, and a pool of processes is created using the multiprocessing module. The colorchunk function is called for each chunk of vertices and passed the shared dictionaries and lock object. The starmap method is used to map the colorchunk function to each chunk of vertices. The results are stored in a list, and the pool is closed and joined.
The sequential version has a time complexity of $O(V^2)$, Where V is the number of vertices in the graph. This is because for each vertex, we need to iterate over all of its neighbors to find their colors and then choose the smallest available color that does not conflict with the colors of adjacent vertices.
The parallelized version, on the other hand, divides the set of vertices into smaller chunks and processes them in parallel. Assuming there are p processes, the time complexity becomes $O(V^2/p + p)$, Where V is the number of vertices in the graph. The $O(V^2/p)$ term comes from dividing the vertices into p chunks and processing each chunk independently, while the $O(p)$ term comes from the overhead of managing the processes and distributing the work.
As p increases, the $O(V^2/p)$ term decreases, leading to a faster overall runtime. However, as p increases, the $O(p)$ term increases, leading to more overhead and potential performance degradation. Thus, there is an optimal value of p that minimizes the total time complexity.
Theoretically, Parallel version should have a better performance than original version for large graphs with many vertices because it can utilize multiple CPU cores and process the vertices in parallel. However, the performance improvement may depend on several factors, such as the number of processes, the size of the graph, the chunk size, and the speed of the CPU. If the graph is small, or the chunk size is too small the overhead of creating multiple processes may outweigh the performance gains from parallel processing.

3.4 Exact algorithm

This code solves the graph coloring problem using an exact algorithm that searches for a valid coloring by recursively trying all possible colorings for the vertices.
The algorithm starts by initializing the colors of all vertices to None. Then, it recursively tries all possible colors for the first vertex and checks if the resulting coloring is valid by checking if adjacent vertices have different colors. If the coloring is valid, it recursively tries all possible colorings for the second vertex and continues until all vertices are colored. If at any point, no valid coloring is found, the algorithm backtracks and tries another color for the previous vertex. If a valid coloring is found, the algorithm returns the coloring.
The time complexity of the code depends on the size of the search space. The average and maximum running times for this algorithm depend on the size of the input graph and the number of colors used. In the worst case, the algorithm must try all possible colorings, which takes exponential time. In practice, the algorithm may terminate much faster if a valid coloring is found early in the search. Specifically, the worst-case time complexity of this code is $O(K^V)$, where V is the number of vertices in the graph and K is the maximum number of colors allowed. To see why the time complexity is exponential, we need to consider that for each vertex, we are trying K different colors. Therefore, the total number of possible colorings is K^V. In the worst case, we have to try all possible colorings before finding a valid coloring, so the time complexity is $O(K^V)$. The space complexity of the code is $O(V)$, where V is the number of vertices in the graph. This is because we store the colors of each vertex in a dictionary with n entries, and we only need to store the current coloring of the graph and the adjacency list of the graph during the search.

Algorithm 3 Exact algorithm

search(G, C, K, V)
G: Graph,
C: dictionary of colors
K: bound
V: counter
if $v = countofvertices$ **then**
  return colors
end if
for color in range of K **do**
  Set the color of the current vertex to the current color.
  Check if the current coloring is valid using the isvalid function.
  if the current coloring is valid **then**
    increase v by 1
    result←recursively call search function with increased v value
    if result not equal to None **then**
      return result
    end if
  end if
end for
If no valid coloring has been found set the vertice color to None and backtrack
return None

graphColoringExact(G, C, K, V)
G: Graph,
K: bound
colors ← A: None, B: None,.... Initialize all vertices to None color in a dictionary
for $i = 1, 2, ..., K + 1$ **do**
  result = search(graph, colors, i, 0)
  if result not equal to None **then**
    return result
  end if
end for
If no valid coloring has been found set the vertice color to None and backtrack
return None

4 Experimental Evaluation

4.1 Hardware

Table 1 shows the specifications of the computer.

Table 1: Computer Specifications

Chip	Intel core i5 8300H(2.3 GHz)
Core	8 CPU
Memory	LPDDR5 16GB
Storage	1TB HARD DISK

4.2 Preliminary Tests

First, as a preliminary test, graph coloring for the small graphs shown in Fig.1 were searched for to make sure that the four algorithms shown below worked correctly.(BST: Exact algorithm using BST, APX: Approximation Algorithm, PAPX: Parallel Approximation Algorithm, LDF: Largest Degree First Heuristic.)

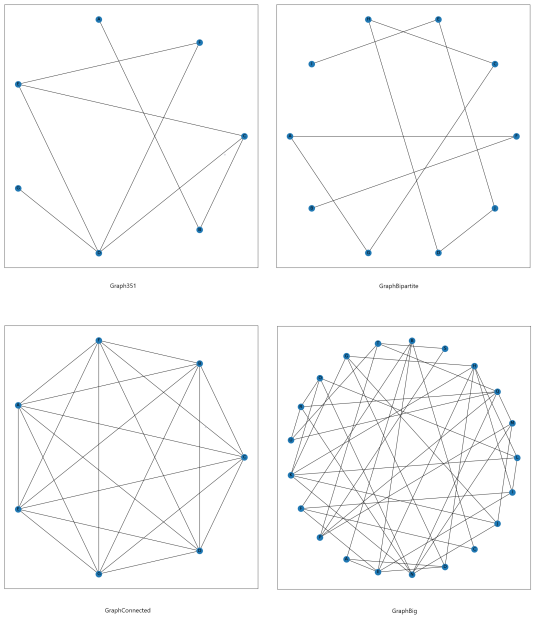

Figure 1: Graphs for preliminary tests

Table 2: Graph coloring

Graph	BST	APX	P-APX	LDF
351	3	3	3	3
Bipartite	2	3	3	2
Connected	7	7	7	7
Big	3	4	4	4

Table 3: Time Consumption (ms)

Graph	BST	APX	P-APX	LDF
351	197	0.995	769	0.998
Bipartite	201	0.996	857	0.787
Connected	166	0.010	812	0.702
Big	135	0.997	625	0.998

4.3 Data

As a benchmark for the implemented algorithms, the Parameterised Algorithms and Computational Experiments 2019 (PACE 2019) (https://pacechallenge.org/2019) benchmark was used. This benchmark included 200 undirected graphs. Due to system limitation 100 files out of them were selected for the experiment. These graphs vary greatly in size, with a minimum of 210 vertices and a maximum of 98,128. Additionally, the number of edges in the graphs ranges from 600 to 161,357.

4.4 Results

In Figure 2 and 3, there are charts that compare the results of two methods: heuristic and approximation, used for coloring graphs. The heuristic approach used fewer colors to color the graphs so that no two connected shapes had the same color compared to the approximation approach. Furthermore, the heuristic approach took less time than the approximation approach, especially when dealing with larger graphs.
In Figure 4, the chart represents the relation between maximum degree of the graph and the graph coloring used by approximation algorithm. As theoritically proven it was supposed to give less than the maximum degree of any graph. In experiment results we can see that the approximation algorithm indeed guarantees an upper bound of $\Delta + 1$.
In Figure 5, two approaches are compared, the approximation (greedy) and the parallelized approximation algorithm. The parallelized algorithm is expected to perform better in practice by distributing the work among multiple processors, which is supported by theoretical analysis. However, experimental results show that the parallelized algorithm takes significantly more time to execute in Python. This finding suggests that the performance gain from parallelization is limited by the overhead associated with process synchronization and data sharing in Python.
The results depicted in Figure 6 demonstrate the performance of the Exact algorithm on all files, employing a bound of 2 and a timeout of 30 seconds. It is evident that despite the utilization of a bound of 2, the algorithm exceeded the allotted timeout for numerous files. Notably, throughout the entire experiment, the algorithm failed to complete computations with a bound greater than 2 within the designated time limit.

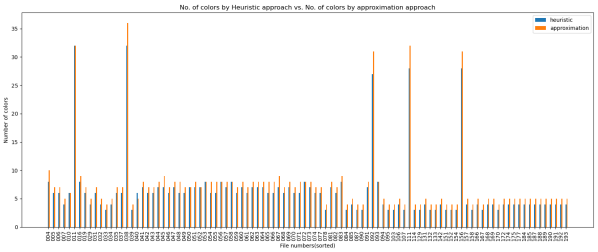

Figure 2: No. of colors used for graph coloring - heuristic vs approximation

5 Discussion and Conclusion

5.1 Heuristic vs Approximation

This experiment employs one heuristic approach, two approximation algorithms, and one exact algorithm

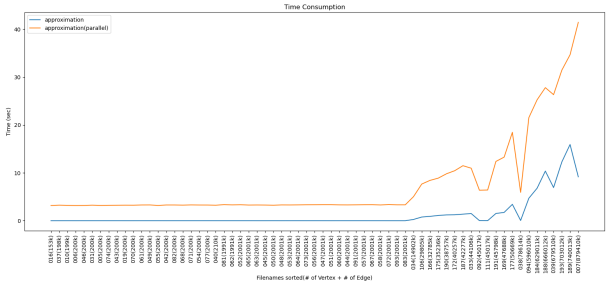

Figure 3: Time consumption for graph coloring - heuristic vs approximation

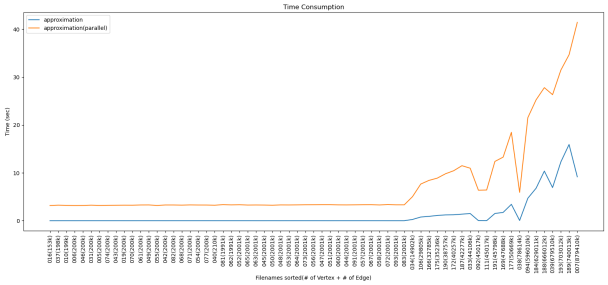

Figure 5: Time consumption - Approximation vs Parallelized approximation

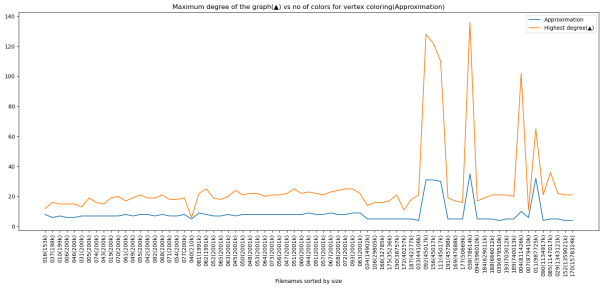

Figure 4: Relation between Maximum degree and min vertex color(Approximation)

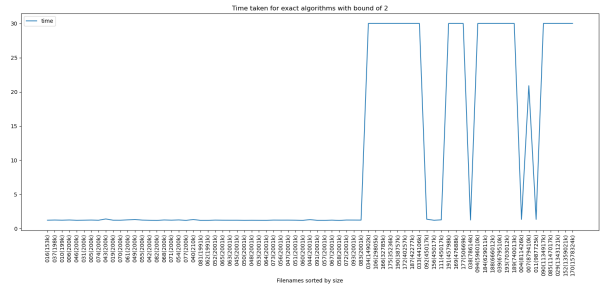

Figure 6: Time consumption - Exact algorithm with bound of 2

to solve the graph coloring problem. The objective of the graph coloring problem is to assign colors to the vertices of a graph such that no two adjacent vertices share the same color. Our experimental results show that the heuristic approach outperforms the approximation approach in terms of both the number of colors used and the execution time. Specifically, the heuristic approach requires fewer colors to uniquely color the graphs and executes faster, particularly when dealing with larger graphs. These findings demonstrate the effectiveness of heuristic methods for solving graph coloring problems.

5.2 Accuracy of Approximation algorithm

The maximum degree of vertices in a graph is represented by Δ . Theoritically, Both approximation algorithms guarantee an approximation factor of $\Delta + 1$. Our experimental study indicates that both approximation algorithms indeed achieve this theoretical approximation factor in practice for all instances tested. These results provide empirical evidence supporting the validity and effectiveness of these approximation algorithms for solving graph problems.

5.3 Limitations of parallelization while using python

Parallelization is a widely used technique to enhance the performance of algorithms by distributing workload among multiple processors. However, in practice, parallelizing certain algorithms in Python may not always result in better performance due to unexpected behavior such as race conditions and inconsistencies in the results when multiple processes modify shared variables concurrently. This issue can be resolved using multiprocessing Lock, which synchronizes access to the shared variables, allowing only one process to modify them at a time. Although this approach ensures correct results, it can reduce the effectiveness of parallelization, particularly for graph coloring algorithms where only one process can color a vertex at a time. Our experiments show that even for larger graphs, a single-threaded algorithm may perform better than parallelized graph coloring algorithms due to the additional time required to synchronize access to shared variables using multiprocessing Lock. Furthermore, Python's lack of built-in multi-threading support necessitates the use of external packages, which can further increase processing time when additional CPUs are added or removed. While parallelization can enhance algorithm performance, unexpected behavior when modifying shared variables concurrently can negatively impact the effectiveness of parallelized algorithms. Thus, careful consideration should be given to the design of parallel algorithms, particularly for graph coloring, where the use of multiprocessing Lock can significantly reduce performance gains. Additionally, the limitations of Python's multi-threading support should also be considered when deciding to parallelize an algorithm.

5.4 Limitations of exact algorithm

The approach used in this experiment is theoretically superior to brute force methods because it reduces the time complexity of the algorithm. The number of nodes in a bounded search tree is much smaller than the total number of possible solutions, making the search process much more efficient. This is especially true for problems with large solution spaces where a brute force approach would be impractical due to the sheer number of solutions that need to be explored. The exact algorithm utilises bounded search trees that systematically search all possible color assignments until a valid coloring solution is found. However, the number of possible color assignments in a graph increases exponentially with the number of vertices, making the search space intractably large for larger graphs. In experiments we found that for bound more than 2, it didn't provide the result within the time bound of 30 seconds. Therefore, exact algorithms that use bounded search trees cannot be run on larger graphs due to the prohibitive computational resources required to explore the exponentially large search space. As a result, approximation algorithms that sacrifice optimality in exchange for faster computation times have become popular alternatives for solving large-scale graph coloring problems.

6 References

References

[1] Matula, D. W., Beckwith, R. I. (1983). A comparison of five heuristic algorithms for graph coloring. SIAM Journal on Computing, 12(2), 281-297.

[2] Karimi, S., Ranjbar, M. (2017). A New Randomized Greedy Algorithm for Graph Coloring Problem. Journal of Discrete Mathematics, 2017, 1-9.