Chromatic Number

Iova Daniel-Alexandru

1st year, CEN 1.2 B

Academic Year 2019 - 2020

1 Requirement:

Chromatic Number. Let G be an undirected graph and k a natural number greater than zero. We define a k-coloring of G an assignment of a color to each node of the graph such that any two adjacent nodes are colored differently. The Chromatic number of G, denoted by $\chi(G)$ or $\gamma(G)$, is the smallest k for which G admits a k-coloring. I am tasked with implementing two different algorithms that calculate the chromatic number of a given undirected graph.

2 Proposed algorithms:

2.1 Welsh-Powell-Based Chromatic Number Algorithm

WELSH-POWELL-CHROMATIC-NUMBER(G, N)

```
1  // G is the adjacency matrix of the array, N is the number of nodes
2  col = 0
3  k = 0
4  // Initialize the array A with the initial nodes and the result array with −1.
5  for i = 0 to N − 1 do
6        A[i] = i
7        result[i] = −1
8  ▷ Sort the sequence A[0 . . n − 1] in decreasing order of node degree.
9  for i = 0 to N − 1 do
10       if result[A[i]] == −1 then
11             ok = 0
12             for j = 0 to N − 1 do
13                   if G[A[i]][A[j]] == 0 and result[A[j]] == −1 then
14                         if A[j] not adjacent to nodes of current color then
15                               result[A[j]] = col
16                               ok = 1
17             if ok == 1 then
18                   col = col + 1
19 // Calculate the chromatic number as the maximum of the result array + 1
20 for i = 0 to N − 1 do
21       if result[i] > k then
22             k = result[i]
23 k = k + 1
24 Return k
```

The Welsh and Powell algorithm introduced an upper bound to the chromatic number of a graph. It is a heuristic greedy-based algorithm in which the nodes are sorted by their degree in decreasing order [1]. This provides a somewhat more optimal ordering than just randomizing the order of the nodes or just considering the order 0 to n-1 and hoping we get an optimal result. Because this is still a greedy algorithm, the result we are going to obtain is not always the optimal one, but we can expect a suboptimal one.

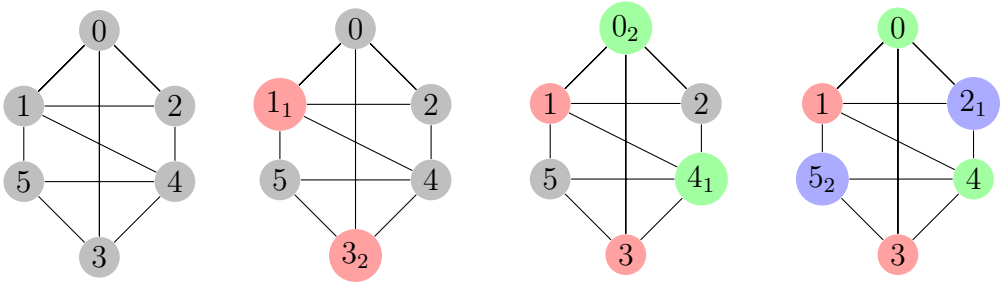
The algorithm consists of the next steps:

1. Generate array with nodes sorted by degree in decreasing order.
2. Color the first node in the array with the color 0.
3. Continue down the array and color, with the same color, all the nodes not connected to nodes of the same color.
4. Increment the color counter.
5. Repeat the two previous steps on all uncolored nodes until all the nodes are colored
6. The chromatic number will be the maximum element of the coloring array plus 1 (because we started with color 0).

The **computational complexity** is $\mathcal{O}(\mathbf{N^2})$, because we first have $\mathcal{O}$(N) for initializing the A array and result array, $\mathcal{O}$(NlogN) from sorting the array A, $\mathcal{O}(N^2)$ for when we iterate through the A array and assign colors to the result array and $\mathcal{O}$(N) for when we calculate the maximum of the result array.

The **memory requirement** is $\mathcal{O}(\mathbf{N^2})$, because we have $\mathcal{O}(N^2)$ for storing the input in the adjacency matrix G, $\mathcal{O}$(N) for storing the result array and $\mathcal{O}(1)$ for storing the chromatic number.

Example: For the following example, the order of vertices after ordering by maximum degree is [1, 4, 0, 2, 3, 5]. The resulting chromatic number is 3 (subscript numbers represent the order of coloring for each color).



Initial Graph | Apply 1st color | Apply 2nd color | Apply 3rd color

2.2 Modified M-Coloring Algorithm

M-Coloring-Chromatic-Number(G, N)

```
1  // G is the adjacency matrix of the array, N is the number of nodes.
2  // G is used when testing if it safe to color the nodes for the color config.
3  K = 1
4  ▷ Generate color configuration array with K colors.
5  if color configuration incomplete then
6        while color configuration incomplete do
7              K = K + 1
8              ▷ Generate color configuration array with K colors.
9  Return K
```

The m-Coloring problem requires that we test if a given undirected graph is colored with a given m. It uses backtracking to generate a color configuration for the given m, and if it is incomplete, returns false, otherwise returns true [2].

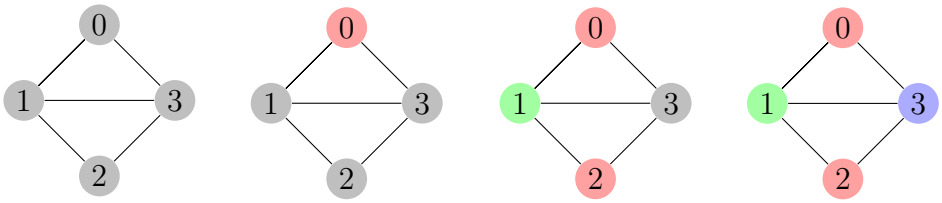
I have modified this algorithm by making m a variable that starts from 1 and is incremented until the generated color configuration is complete.

The color configuration is generated by coloring each vertex, starting from vertex 0, such that, before assigning a color, we check if the node is adjacent to any nodes of the same color. If it is safe to color the node, assign it a color and add it to the current color configuration. If no colors are assigned, backtrack and return false.

The **computational complexity** of the algorithm is $\mathcal{O}(\sum_{i=1}^{K} i^N)$ because there are K^N possible combinations, with K starting from 1 and incrementing itself until it reaches the chromatic number.

The **memory requirement** of the algorithm is $\mathcal{O}(\mathbf{N^2})$, because we have $\mathcal{O}(N^2)$ for storing the input in the adjacency matrix G, $\mathcal{O}(N)$ for storing the color configuration array and $\mathcal{O}(1)$ for storing the resulting chromatic number.

Example: For the following example, the chromatic number (denoted by K) starts from 1 and increments itself until a complete combination is found. The order in which the nodes are colored is irrelevant, so the example shows one of the possible combinations (color 0 - red, color 1 - green, color 2 - blue).



Initial Graph | Returns False | Returns False | Returns True

3 Experimental data:

The non-trivial data is created using an algorithm which firstly generates a random number of nodes N. Then, for a random number of lines, generates two integers in range [0..N-1] and adds the pair to an array of pairs. The array is then cleaned up by removing inverted duplicates (i.e. we have "a b" and "b a" in the array so we remove either one of them) and pairs with same value for both members (i.e "a a", "b b"). In the end we print N, followed by the lines of pairs. The data is output to a file which will become one of the input files for the two algorithms.

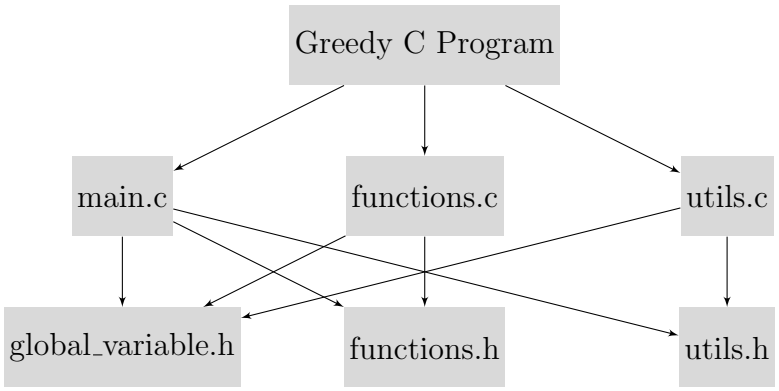
Because the data is to be run on both algorithms and one of them is a backtracking algorithm, the input cannot consist of big values (the number of distinct edges cannot be greater than ≈70 edges for the Python implementation and ≈120 edges for the C implementation). On the other hand, the greedy algorithm can run inputs of more than ≈800 edges while still outputting a sub-optimal or even optimal result. The downside to running big inputs on the greedy algorithm is that, because this problem is only correctly solved by a brute-force/backtracking approach [3], we will never be able to test if the result is correct or not in a reasonable amount of time (or at all).

The generated data is significant for testing because it shows the difference in performance and speed between the two algorithms, but also because it shows the difference in speed between the Python and C implementations.

4 Experimental application development:

Following are the high-level structures of both algorithms in their C implementations along with descriptions of their modules and procedures.

4.1 1st Algorithm:



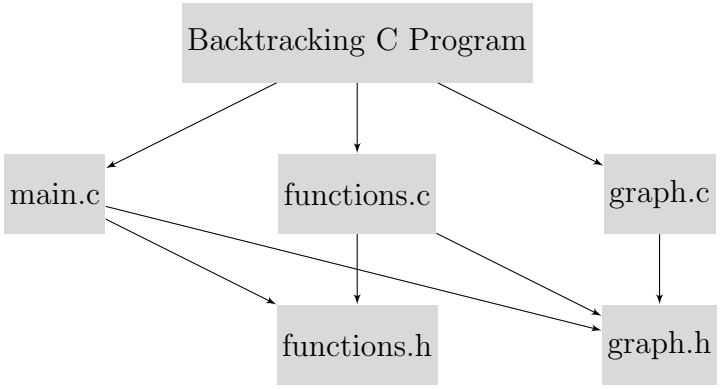
Modules and their respective procedures:

- *main.c*
  - Main program module
  - Procedures:
    - **int main()** - main function of the program
- *generate_array.c*
  - Module with procedures that generate the array with sorted nodes
  - Procedures are declared in the header *generate_array.h*
  - Procedures:
    - **int degree_of_node(int node)** - returns degree of given node

    - **int cmp(const void *a, const void*b)** - comparator function for qsort which uses previous function for comparison
    - **void generate_list_by_degree(int array[], int no_nodes)** - generates the array using the previous functions and C's qsort

- *utils.c*
  - Multi-purpose module which includes functions that range from input-output handling to assigning colors to the result array
  - Procedures are declared in the header *utils.h*
  - Procedures:
    - **int check_connected_same_color(int node, int color)** - checks if any node adjacent to the given node has the same color as given color
    - **void assign_color(int node, int vertex_order[])** - procedure to assign colors to the result array
    - **void read_data(const char* file)** - reads data from given file and assigns it to the global variables
    - **void print_result(const char* file)** - computes and prints the chromatic number to given file

4.2 2nd Algorithm:



Modules and their respective procedures:

- *main.c*
  - Main program module
  - Procedures:
    - **int main()** - main function of the program in which the final coloring function is called until a valid combination is found; prints the computed chromatic number to an output file
- *functions.c*
  - Module responsible for I/O functions
  - Procedure is declared in the header *functions.h*
  - Procedure:
    - **void read_data(struct Graph* g, const char* file)** - reads input from file and assigns it to a "Graph" structure g
- *graph.c*
  - Module solely involved with graph related procedures

- Procedures and structure "Graph" are declared in the header *graph.h*
- Procedures:
  * **int safe_to_color(struct Graph g, int vertex, int coloring_result[], int color)** - checks if given vertex is safe to color (no adjacent node has the same color)
  * **int graph_color_utility_function(struct Graph g, int chromatic_number, int coloring_result[], int vertex)** - recursive utility procedure used to build the coloring_result array
  * **int final_coloring_function(struct Graph g, int chromatic_number, int coloring_result[])** - final coloring function in which the coloring result is built by calling the previous function

The Python implementation of this algorithm uses a class, with everything inside C's "graph.c" being procedures of said class.

4.3 Common Traits:

To change what input file is to be run:
- For the **C** implementations:
  - Change the "INPUT_INDEX" macro inside "main.c".
  - Build and Run the program.
- For the **Python** implementations:
  - Change the "input_index" variable inside "main.py".
  - Save and Run the program.

Input data consists of values in range of [1..100] nodes and [0..150] edges. It is of the following format:
$\boldsymbol{n}$
$\boldsymbol{x_0}$ $\boldsymbol{y_0}$
$\boldsymbol{x_1}$ $\boldsymbol{y_1}$
$\boldsymbol{x_2}$ $\boldsymbol{y_2}$
...
where $\boldsymbol{n}$ is the number of nodes and the number of edges ($\boldsymbol{x_i}$ $\boldsymbol{y_i}$) is random. Output data is an integer representing the computed chromatic number.

5 Results and Conclusions:

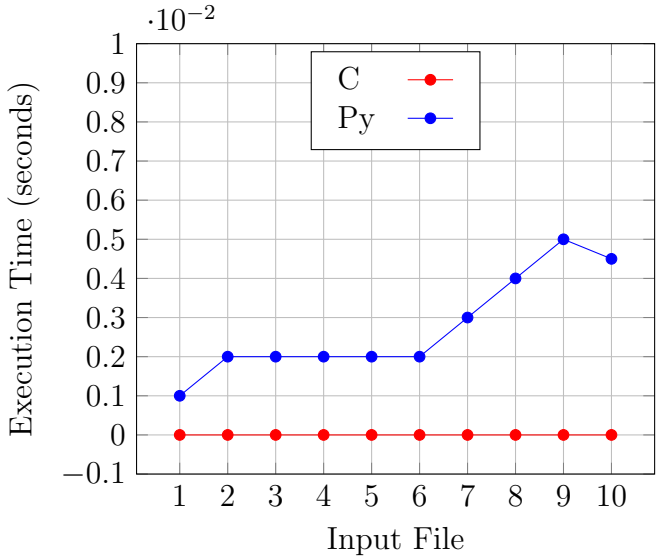
Both algorithms output a single integer: the chromatic number of the given graph. Running the algorithms on 10 input files, we can see in the table below that the greedy algorithm has a 60% success rate, while the backtracking one returns the optimal answer everytime. With that being said, the wrong outputs are still sub-optimal.

To test the correctness of their output, I used an online tool called "SageMathCell" [4], which is an online wrapper for the mathematics software system SageMath.

Input file	1	2	3	4	5	6	7	8	9	10
Chromatic number	4	2	3	3	3	3	3	4	4	3
Greedy algorithm output	4	2	3	3	4	3	3	5	5	4
Backtracking algorithm output	4	2	3	3	3	3	3	4	4	3

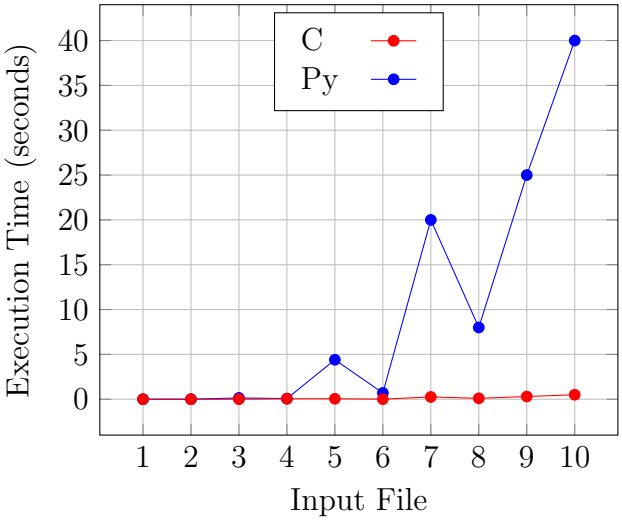
Below are graphs of the execution time differences between the C and Python implementations of each algorithm. I chose to split the representation in 2 different graphs because this allows me to better show the difference. If I were to represent all of them on the same graph, the first algorithm's representations would be glued to 0 on the X axis.

Greedy algorithm execution time comparison



For this first graph, we can see a clear difference between the execution times of the Python and C implementations. This is a negligible difference though, with the values being in the range of 0 to 0.005 seconds.

Backtracking algorithm execution time comparison



For this second graph, we can see the massive difference between the execution times. It seems Python is continuing to be the slowest between the two, getting up to 80 times slower than the C implementation. In comparison, Python's slowness starts to show in the first algorithm when the node count is $\approx 10^5$ with more than ≈ 500 edges (getting around 3 times slower than it's C counterpart).

The results show the staggering difference between the two algorithms (reaching up to an 8000 times difference between their worst cases). For the first one we trade correctness for speed, while for the second we do the exact opposite. Even though we cannot test inputs of larger size it is still interesting to see how the greedy algorithm starts to give faulty answers or how the backtracking one struggles with time under a regular-sized input.

This was an interesting and complicated project to create, with the most difficult part of it being the C implementation. With the lack of classes and the easiness of writing Python code, writing in C felt more like a chore. It did pay off though, with the blazing fast execution times. This cannot be said about the 2^{nd} implementation as much, with one of the future goals of this project being to optimize the backtracking algorithm more so that it can process larger inputs. All in all it was a great learning experience which helped my algorithmic skills grow.

References

[1] Aslan Murat, and Baykan Nurdan. *A Performance Comparison of Graph Coloring Algorithms*. International Journal of Intelligent Systems and Applications in Engineering, 2016.

[2] Article on a backtracking m-Colorability algorithm, https://www.includehelp.com/algorithms/graph-coloring-problem-solution-using-backtracking-algorithm.aspx

[3] Article on NP-Completeness, https://www.britannica.com/science/NP-complete-problem

[4] SageMathCell, web interface to the free open-source mathematics software system SageMath,
https://sagecell.sagemath.org