A multithreading graph coloring implementation

A.A 2020/2021

Matia Torlini s281431
Lorenzo Semeraro s288183

Indice

1. Introduction

Graph coloring is a heavily studied problem with many real world application. Formally, a (vertex)-coloring of an undirected graph G = (V,E) is an assignment of a color v.color to each vertex v V such that for every edge (u, v) E, we have u.color != v.color, that is, no two adjacent vertices have the same color. The graph-coloring problem is the problem of determining a coloring which uses as few colors as possible. Although the problem of finding an optimal coloring of a graph — a coloring using the fewest colors possible — is an NP-complete problem, heuristic "greedy" algorithms work reasonably well in practice. However, different heuristics produce different results in terms of coloring quality (number of colors used) and time to produce the coloring. In this project, we designed a parallel algorithm to make it work with different heuristics :

- the random ordering heuristic
- the smallest-degree-last ordering heuristic (SDL)
- the largest-degree-first ordering heuristic (LDF)

The first one colors vertices in a uniformly random order, while the SDL colors the vertices in the order induced by assigning a weight to each vertex. Weights depend upon the degree of each vertex (number of edges connected to a node). In case two close vertices are assigned the same weight, a random number is used to solve the conflict. The random ordering heuristic performs better in terms of time, but offers a bad-quality coloring. On the other hand, the SDL heuristic requires more time to be put in place, but offers much better coloring quality. LDF uses degree as ordering heuristic (once again conflicts are solved by randoms) in order to use a fewer number of colors with respect to the random one by sacrificing some execution time.

2. Data structures

2.1 Vertex.h

Each vertex is characterized by a long int id, which corresponds to its index in the vertices' vector in Graph.h. Each node has an integer to represent the color (int color) and a long int variable to store its random number (long int random). Neighbors are saved as a list of pointers to vertices (std::list<Vertex*> neighbors). There are 2 copies of std::list neighbours: one keeps the original neighbors for each vertex, while neighbours_updated_jp keeps the list of neighbors excluded the colored ones. An additional copy of this list neighbours_updated_sdl is used by the weighting algorithm. Moreover, a std::shared_mutex m is included for each vertex, in order to manage synchronization between thread in accessing and updating neighbors during the weighting phase. For the weighting phase also the variables int weight and int degree are saved. The class "Vertex" exposes the following methods to perform any action required :

- void set_random();
- int get_color();
- void add_neighbor(Vertex* neigh) : pushes a new vertex pointer into the list of neighbors
- bool has_biggest_random() : returns true if the vertex has the biggest random among its neighbors, false otherwise
- void assign_color() : finds and assigns the smallest color available in the neighborhood
- void delete_from_neighborhood() : delete the pointer to itself from the neighbors list of its adjacency list
- void print_vertex() : prints on standard output vertex's attributes
- void init_degree()
  initializes the degree of this vertex to the count of its neighbors
- bool compare_degrees(int degree)
  locks a shared mutex, and returns true if the degree of this vertex is lower or equal the parameter d
- void decrease_degree()
  decreases the degree of this vertex by one

- void set_weight(int w)
  sets the weight of this vertex to the parameter w
- void delete_neighbor_sdl(Vertex* n)
  deletes the neighbor pointed by neigh from the adjacency list dedicated to the weighting algorithm
- void remove_sdl()
  removes this vertex from the adjacency list of its neighbors dedicated to the weighting algorithm void has_biggest_weight()
  returns true if the vertex has weight > all the weights of its neighborhood (conflicts resolved by random numbers) void has_biggest_degree()
  returns true if the vertex has the biggest degree of all the degrees of its neighborhood (conflicts resolved by random numbers)

2.2 Graph.h

The fundamental data structure for the Graph.h class is a vector of shared pointers std::vector<std::shared_ptr<Vertex» vertices, each one pointing to a memory zone where a Vertex has been allocated through the std::make_shared call during the loading phase. A std::set is used to store the colors assigned by the algorithm. Here all the functions used in Graph.h :

- std::vector<Vertex*> vector_to_shuffle() : returns a vector of raw vertices pointer to be shuffled in the sequential greedy algorithm
- size_t get_size()
- void set_size()
- void add_vertex(const std::shared_ptr<Vertex>& v)
- void add_neighbor(long id, long neigh) : invokes the Vertex::add_neighbor() method on vertices[id] adding to its neighbors list vertices[long]
- void assign_random(long id, long rand) : assigns to vertices[id] the rand random
- bool has_biggest_random(long id) : invokes Vertex::has_biggest_random() on vertices[id]
- bool has_biggest_weight(long id) : invokes Vertex::has_biggest_weight() on vertices[id]
- bool has_biggest_degree(long id) : invokes Vertex::has_biggest_degree() on vertices[id]
- void color(long id) : invokes Vertex::assign_color() on vertices[id]
- void update_neighborhood(long id) : invokes Vertex::delete_from_neighborhood() on vertices[id]

- void number_of_colors() : prints the number of colors used
- void check_coloring() : checks sequentially if the colorization assigned is acceptable
- print_vertices_to_file(std::string output_file) : print the output of the algorithm on the file specified by output_file.
- void init_degree(long int id)
  initializes the degree of this vertex to the count of its neighbors
- void try_to_weight(long int id, int degree, int weight)
  returns true if the degree of the vertex identified by id is lower or equal the parameter d, and assigns the weight w in this case
- void remove(long int id)
  removes the vertex identified by id from the adjacency lists of its neighbors dedicated to the weighting algorithm

2.3 Smart_index.h

Whenever a cycle of colorization for an independent set is completed, the vector of vertices has to be updated excluding vertices that have been colored during that cycle. During the study of the problem, two possible solutions have been explored for addressing this problem :

- Set the random number of a processed vertex to -1
- Use a list of Vertex instead of a vector and whenever a cycle is completed, delete the nodes from the list

The first of solution implies that the exclusion condition has to be verified for each vertex at each cycle, throwing away a discrete amount of clock cycles.
The second option requires the list to be iterated for deleting each node related to a processed vertex: again a huge number of machine cycles are wasted. To avoid the problems above mentioned and try to optimize the updating phase, a support data structure has been implemented. Its most important attributes are :

- long int idx : defines the current index
- std::vector<long> next : for each index idx defines its next index
- std::vector<long> prev: for each index idx defines its previous index

It is a sort of simple "lookup-table" which allows to exclude vertices to be processed while they are colored by updating the next and prev vectors.
Example: consider a graph having 6 vertices that have to be processed using just one thread for ease of explanation. Next and prev will be:

next : 1 2 3 4 5 0
prev : 5 0 1 2 3 4

The function which updates the index void update(bool processed) updates the index differently in two cases :

- the vertex vertices[idx] has been processed (processed = true). In this case the table is updated. For example, if the vertex with id 4 has been processed, next and prev vectors will be :
  - next : 1 2 3 5 5 0 (vertices[3] next element will be vertices[5])
  - prev : 5 0 1 2 3 3 (vertices[5] prev element will be vertices[3])
- the vertex vertices[idx] has not been processed (processed = false): idx becomes next[idx]

3. Graph loading

3.1 Loader.h

The Loader class constructor takes as parameters :

- a reference to the Graph data structure where to load data
- a reference to a string representing the input file

Initially, the Loader reads the first line, setting the size of the Graph and initializing the whole vector of vertices. Then 4 threads are launched, each one of them initializes a safe_reader object, which is in charge to read one line from the file and add the neighbors for each vertex. The safe_reader are coordinated through a static std::mutex and a common atomic variable shared_cnt to preserve the order of indexes. The core function of the safe_reader is the function get_line() which is protected by a std::scoped_lock, so that one thread at a time can call the std::getline(file, line, ')́ on the input file. Each thread reads one long integer i at a time from its line, and invokes Vertex::add_neighbor(long id, long i) on vertices[shared_cnt]. Once the whole file has been read, the data structure is ready to be processed.

4. Jones-Plassman

4.1 Proposed implementation

The proposed implementation of the Jones-Plassman algorithm is inspired by the following pseudo-code The class Jones-Plassman constructor takes as parameter a reference to the

```
U := V
while ( |U| > 0 ) do
    for all vertices v ∈ U do in parallel
        I := {v such that w(v) > w(u) ∀ neighbors u ∈ U}
        for all vertices v' ∈ I do in parallel
            S := {colors of all neighbors of v' }
            c(v') := minimum color not in S
        end do
    end do
    U := U − I
end do
```

Figura 4.1: Jones-Plassman algorithm pseudo-code

graph to process and has the following attributes

- int n_threads : number of threads used
- std::vector<std::pair<long int, long int> indexes vector of n_threads indexes pair <start,stop> for assigning to each thread its portion of the vertices' vector
- std::vector<std::thread> threads
- std::vector<std::list<long int» to_remove : a vector of n_threads list where each thread can push the indexes of processed vertices

The class exposes a void function start(int n, int m) that can be used to start the computation, and where n is the number of threads and m specifies the heuristic to be used for processing the graph.

4.2 The algorithm

The algorithm works as follows : all the variables are initialized, and the graph is partitioned splitting the vertices' vector into n_threads chunks. Then the thread pool is launched. Each thread uses 5 local variables :

- long start : defines the starting index of the portion of the graph vector to process
- long stop : defines the ending index of the portion of the graph vector to process
- long chunk_size : defines the size of the graph portion to process
- long uncolored : number of vertices uncolored
- Smart_index idx(start,stop)

Each thread then creates an instance of a Random_assigner object, and launches it to assign a random number to each vertex in its portion. The Random_assigner uses a Mersenne-Twister-19937 random number generator (it has been proved to be 1.8x times faster with respect to the rand() function, which moreover is not suitable for being parallelized) initialized with a seed which depends on the starting index and on the ending index of the vector portion to which it has to assign randoms. To synchronize during this phase, the threads use a std::barrier without completion function. Once all the threads arrive at the barrier sync_point0 the real Jones-Plassman algorithm starts. While there are uncolored vertices in the vector chunk, the thread iterates on the vector using the smart_index, looking for vertices which comes first depending on the heuristic used. If the vertex comes first, then it is colored, its index is pushed in the thread's to_remove list, the uncolored local variable is decremented and the smart_index is updated. Once the first iteration ends, the thread waits on a std::barrier sync_point1 for the other threads to end. Once all the threads ended their iteration, the function void update_graph is invoked, and all the vertices whose id is in any of the to_remove lists removes its pointer from the copy of neighbors adjancecies list of its neighbors through the function Vertex::delete_from_neighborhood(). This process is reiterated until the number of uncolored vertices becomes zero.

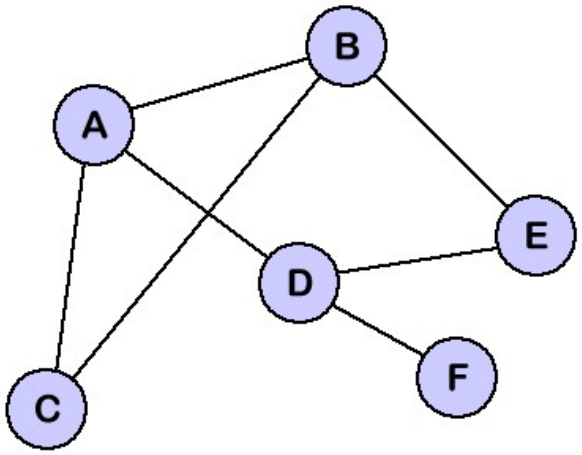
4.3 Considerations

Smart_ptrs vs RAW ptrs

During the development of the project different solutions have been explored. At first, raw pointers weren't used, weak_ptrs where used instead. However, a substantial difference in terms of performances has been observed. On graph 18 (260k nodes) an average gain of 800ms (17% of the total time) has been recorded using raw pointers. Moreover, each raw pointer is part of a shared_pointer wrapped object and it is destroyed when the object itself is destroyed, avoiding the problem of having dangling pointers.

No lock required

It's worth noting that Jones-Plassman implementation does not use any kind of locks, since the concept of independent-set is exploited. We can take as an example the graph in figure. In the case that one thread is processing vertex A and another thread is processing



vertex D, it is impossible that both are colored in the same iteration, since one of them have the biggest random number for sure.
However, the only lock is used to protect the set of colors in the graph, so that all the threads can insert element in a consistent manner.

Graph balancing

It has been observed that in some cases one or more thread completes its cycles much before some others, avoiding the software to use the whole computational power of the processors. This probably is due to the fact that the probability for a vertex to have the biggest random among its neighbours is not uniform in the data chunks assigned to each thread, causing some of them not to work, while others have to work more.

5. Smallest Degree Last

5.1 Proposed implementation

The proposed weighting algorithm for Smallest Degree Last approach is based on the following pseudo-code:

$k := 1$
$i := 1$
$U := V$
while ($|U| > 0$) **do**
    while $\{\exists$ `vertices` $v \in U$ `with` $d^U(v) \leq k\}$ **do in parallel**
        $S = \{$`all vertices` v `with` $d^U(v) \leq k\}$
        for all `vertices` $v \in S$, $w(v) := i$
        $U = U - S$
        $i := i + 1$
    end do
    $k := k + 1$
end do

Figura 5.1: Smallest Degree Last algorithm pseudo-code.

For its implementation a class called Smallest_Degree_Last was created. It consists of the following attributes.

- std::vector<std::thread> threads
  vector of thread objects to execute the weighting in parallel
- int n_threads
  number of threads used
- Graph& graph
  reference to the graph to weight
- std::atomic<bool> processed_flag
  flag to indicate whether at least a vertex was weighted in the last weighting iteration

- std::atomic<int> current_weight
  weight to assign to vertices during the current weighting iteration
- std::atomic<int> current_degree
  degree to compare during the current weighting iteration
- std::vector<std::list<long» to_remove
  list containing identifiers of vertices weighted during the last weighting iteration, that need to be removed from the graph
-

5.2 Algorithm

The main thread initializes the main variables (current_weight and current_degree to 0, processed_flag to false) and dimensions the vector of threads. At this point it defines two functions to use with a std::barrier, which is a class that models a synchronization point between threads:

- auto thread_task = [this, &barrier] (int thread_id)
  is the function to pass to the threads (which are launched right after) so it can be executed in parallel. At the start of this block, the function split_indexes(graph.get_size() is called in order to obtain the identifiers of the first and last vertex inside the own graph fragment to process. Moreover, a Smart_Index object is constructed to manage the iterations in an efficient way. Then, the function thread_weight, responsible for a weighting iteration, and the function barrier.arrive_and_wait(), that allows to wait all the other threads' weightings before proceeding, are launched inside a while loop. on_phase_completion will be executed once all threads are done with their graph fragment. Outside the loop, controlled by a condition that checks if there are still vertices to weight, barrier.arrive_and_drop() decrements the number of threads to wait for at the end of each cycle.
- auto on_phase_completion = [this]() noexcept
  is passed to the constructor of the barrier and contains the code to execute at the end of each weighting iteration. In particular, its job is to update current_weight and current_degree basing on the value of the atomic processed_flag, and to remove (sequentially) all the nodes weighted in the last weighting iteration. This is done looping the vector to_remove containing the lists associated to individual threads.

The single weighting iteration works as the following:

- void thread_weight(int thread_id, long& unweighted, Smart_index & si, std::list<long>& to_remove)
  All threads start, for each vertex in their own associated graph fragment, the function graph.try_to_weight, and in case of positive return they set the shared processed_flag and push in their own list of nodes to remove the identifier of the just processed ones. Then, they update the counters and the Smart Index to manage the loop and proceed to the next vertex.

6. Conclusions

Even though the parallelization of the code should result in a speed-up, in this case the sequential greedy algorithm has better performances in terms of execution time, probably due to OS control over the hardware. Here is some data concerning the random ordering heuristic. About the quality of coloring, it has been experienced a better quality with random numbers heuristic ordering with Jones-Plassman wrt to the sequential greedy only in particular cases. For example on the graph 19 (525k nodes), launching the program with 4 threads the colors used were 18, while the sequential greedy used more. Here we report some graphs which represent the execution times on different graph sizes. All the tests where performed on a 12 processors Intel-Core i7-8750H 2.2GHz. Time is expressed in milliseconds. All the statistics are an average of 15 executions.

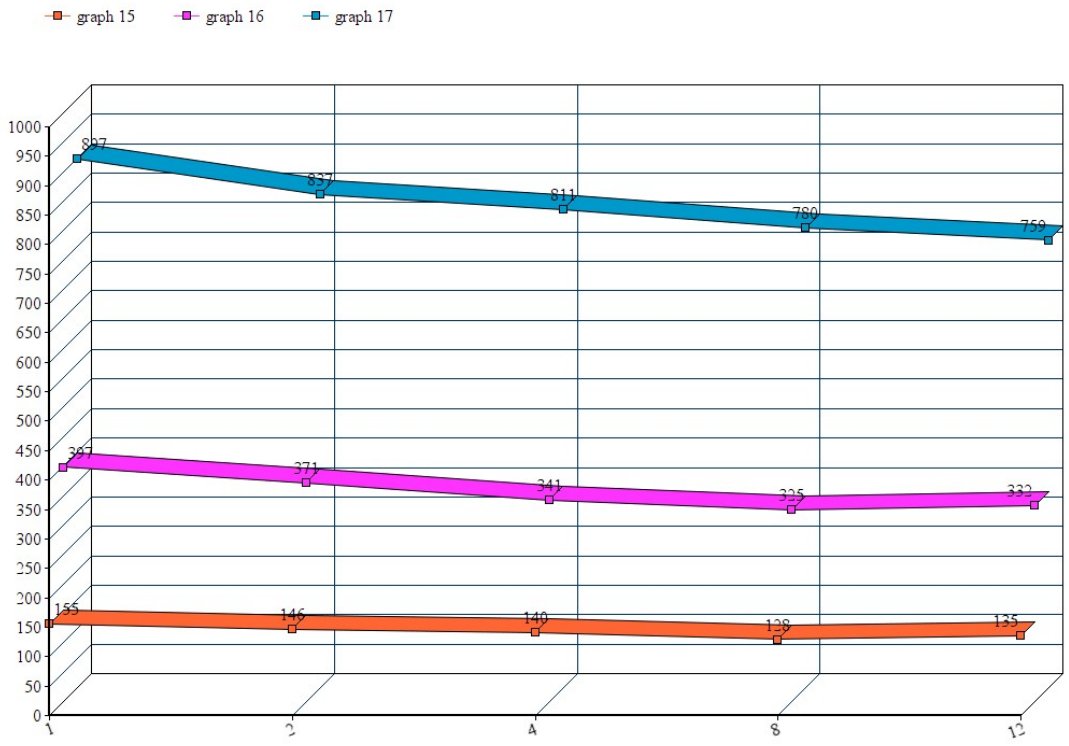


Figura 6.1: Sizes of graph varying from 32k to 130k nodes

It is possible to see how the overall speed-up progressively goes to zero using more and more threads, and in some cases (e.g. graph 22) performance even decrease using more than 4 threads.

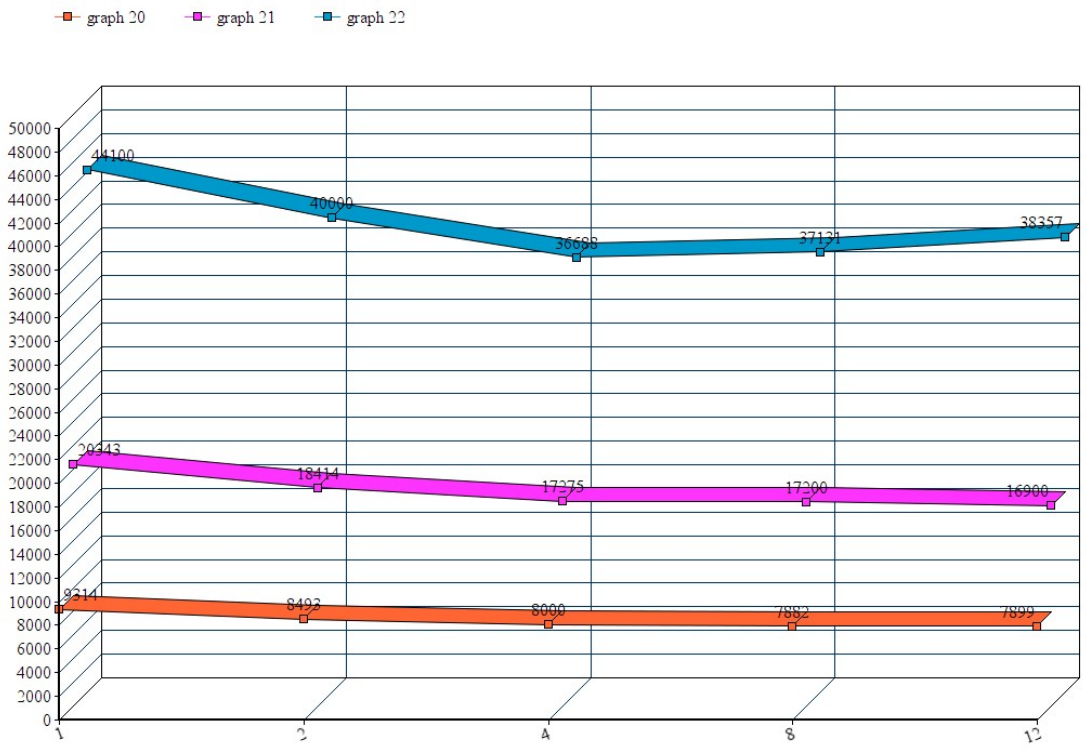


Figura 6.2: Sizes of graph varying from 1mln to 4mln nodes

Since the algorithm is multithreading, the number of colors is not deterministic for each execution. However, we collected several data from different executions, and selected the number of colors which was used the most for each graph. Here a comparison between the sequential greedy and the JP random heuristic ordering quality of coloring.

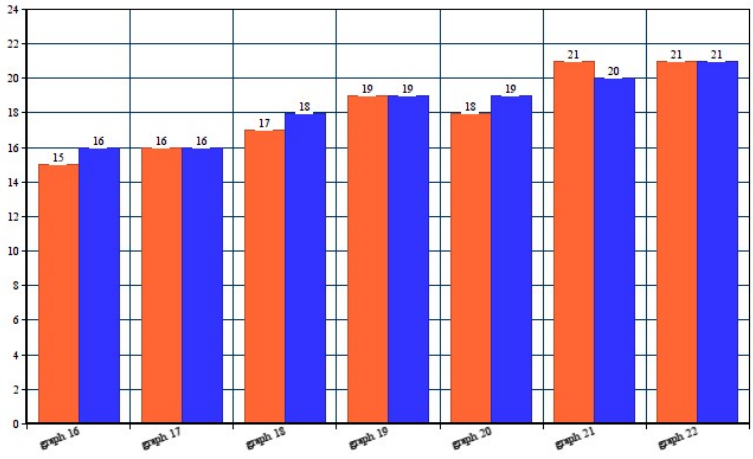


Figura 6.3: In orange the number of colors assigned by the sequential greedy, in blue the number of colors assigned by the Jones Plassman algorithm with random ordering heuristic

As for the "smallest degree last" heuristic, the number of colors used by the algorithm is always less or equal than the number of colors used from the other ones, as expected. The following graphs show execution times for the algorithm using the "smallest degree last"

heuristic run on a 12 cores Intel i7-10750H, 2.6 GHz. Time is expressed in milliseconds. All the statistics are an average of 15 executions.

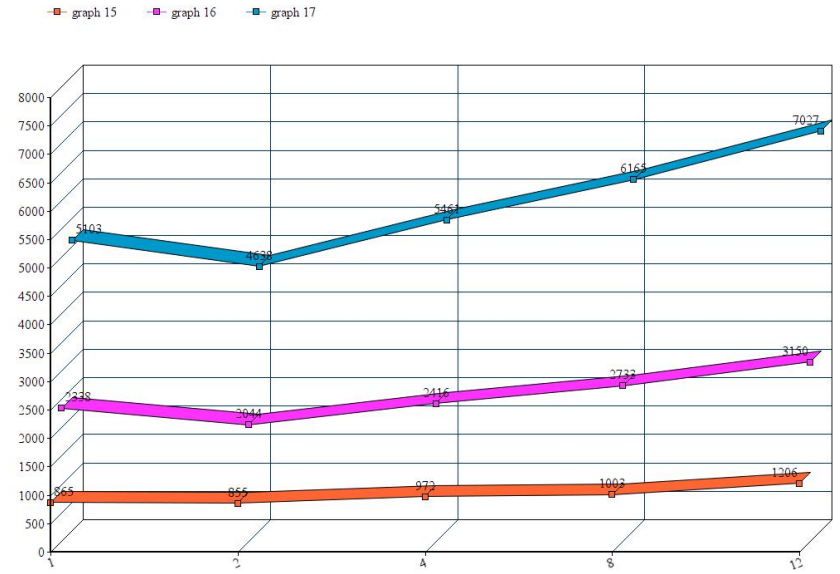


Figura 6.4: Sizes of graph varying from 32k to 130k nodes

As mentioned above, this approach needs a weighting phase to be performed before the actual coloring: this extends computational times, bringing them to about five times the previous algorithm.

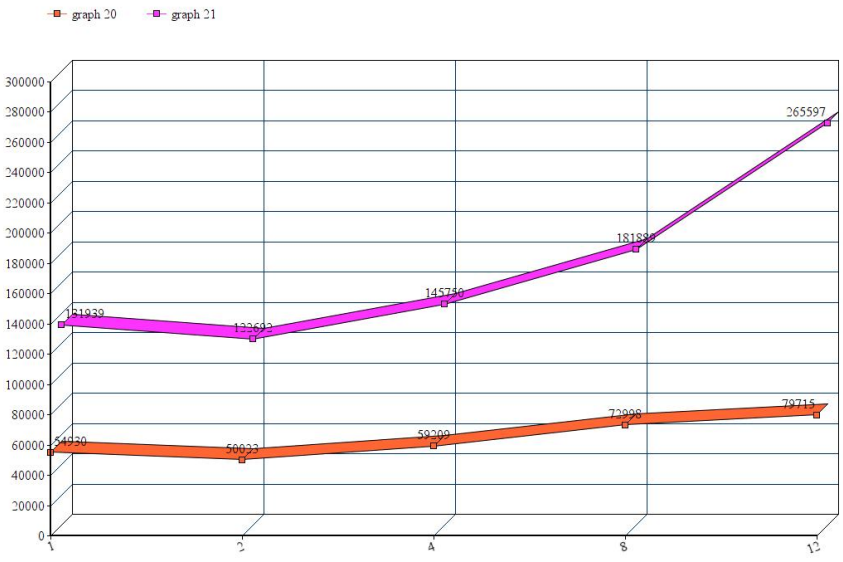


Figura 6.5: Sizes of graph varying from 1mln to 4mln nodes

Overall, the best performances are associated to executions with 2 threads. A possible reason is the use of std::mutex: many synchronization points force the algorithm to slow down.
For what concerns the "lowest degree fist" heuristic, it is possible to observe a much more regular trend about the number of threads: the more it is, the more this algorithm performs. These graphs show execution times for the algorithm using the "lowest degree fist" heuristic run on a 12 cores Intel i7-10750H, 2.6 GHz. Time is expressed in milliseconds. All the statistics are an average of 15 executions.

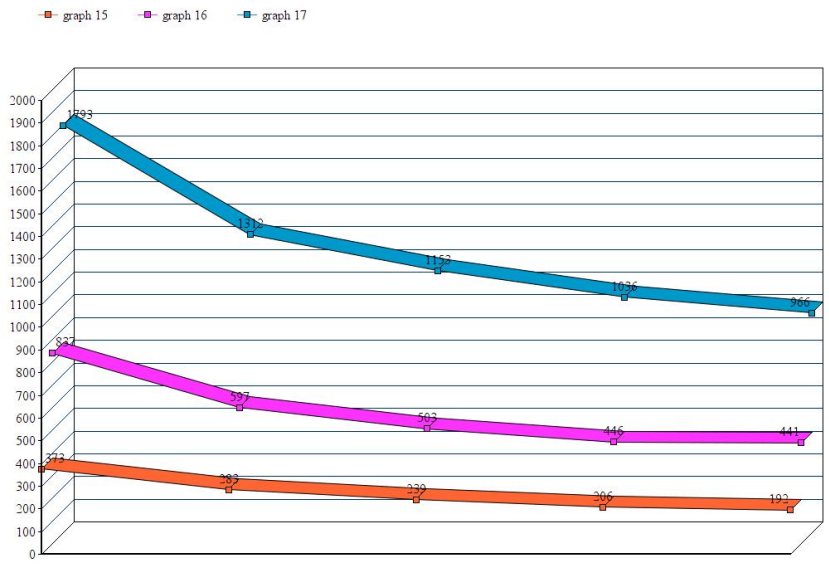


Figura 6.6: Sizes of graph varying from 32k to 130k nodes

This last graph compares instead the color efficiency of the approaches based on the vertex degree. Surprisingly, LDF shows a color efficiency that is comparable with SDL, but still offers much better performances: it represents a good compromise between performance and number of color used for parallel solutions.

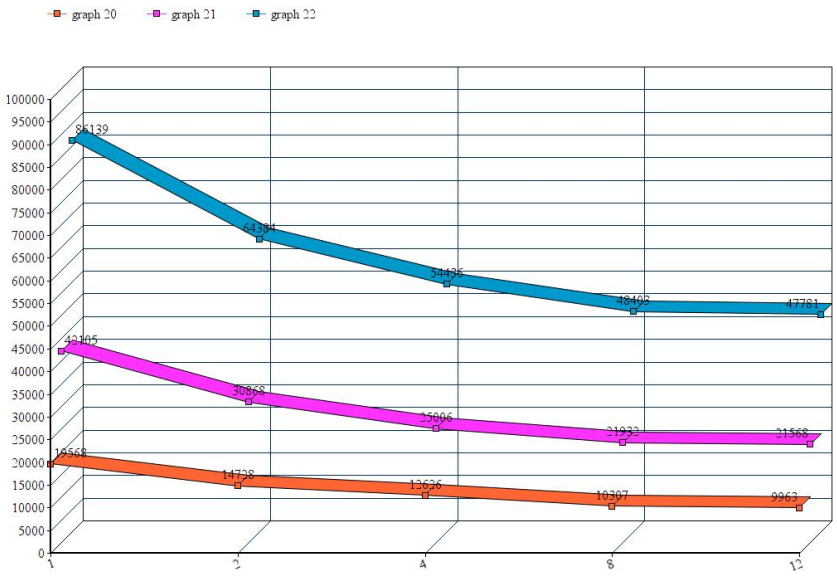


Figura 6.7: Sizes of graph varying from 1mln to 4mln nodes

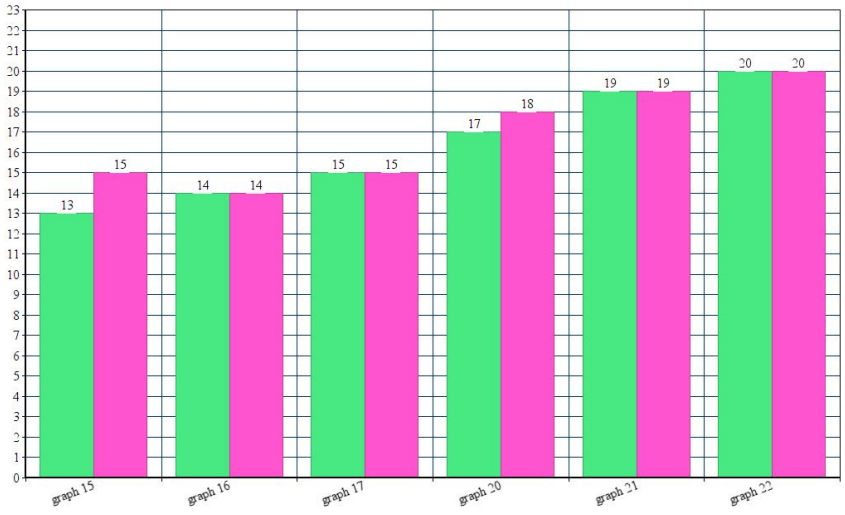


Figura 6.8: In orange the number of colors assigned using the SDL heuristic, in blue the number of colors assigned using the LDF heuristic