Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing Group

Semester Thesis

Distrtributed Graph Coloring

Nico Eigenmann
nicoe@student.ethz.ch

Advisor: Johannes Schneider
Supervisor: Prof. Dr. Roger Wattenhofer
Distributed Computing Group

Dept. of Computer Science
Swiss Federal Institute of Technology (ETH) Zurich
Spring 2011

Contents

1 Introduction

1.1 Motivation

Graph coloring is a well known and computationally hard problem. There are different algorithms to compute approximations in a non-distributed way. The goal of this thesis is to compare different distributed algorithms. The two evolution criteria are, how many colors the algorithm uses and how many rounds of communication it requires.

2
Model

The underlying model in this thesis is the message passing model. The communication is done in rounds. In one round a node prepares a message and sends it to its neighbors. After that a node parses the received messages. Then the next round starts. In this model there are no communication failures. This means each sent message, is received in the same round. The message will also not get corrupted. The graph is fixed. There are no new nodes joining and if two nodes can communicate in the first round they can communicate in all others. Each node has an unique identifier. There is no limitation on message size.

3 Algorithms

3.1 Luby [2]

This is a randomized algorithm. Each node chooses a color at random. Then it sends this color to all neighbors. If no neighbor has chosen the same color, the node is done. Otherwise it chooses a new color in the next round.

3.2 Color Trail [3]

This algorithm is similar to Luby. But instead of choosing a color at random. A node chooses for each possible color a random integer. Then it compares the list with the ones from its neighbors. If it has the absolute maximum by one of the colors, it takes it.

3.3 Directed Edge [1]

In a first phase each edge gets a direction, from the node with the bigger ID to the one with the smaller. If a node has only outgoing edges, he can safely chose a color. In the second phase the nodes perform one round of Luby.

3.4 Splitted ID [3]

In the beginning all nodes are active. Each node splits its 32 bit ID into 4 bit packages. Then it send the packages to its neighbor. With the 8 packages it generate a 8 bit number. If its package is as big as the maximum of all the correspondent

packages, the bit is set to one. Now it distributes the new number to all neighbors and if its number was bigger than the others it chooses a color. If the number was smaller the node becomes inactive until all its neighbors are inactive or colored. If the number was as big, the node does not choose a color but remains active.

3.5 Max Clustering [4]

This algorithm tries to cluster the graph and then a leader in this cluster colors all its nodes. The nodes with the maximum ID in radius $2r+1$ is the leader of a cluster with radius r. The distance between two leaders needs to be this large otherwise two neighboring nodes can be colored from different leaders and therefore have the same color.

3.6 Delta Max

In this newly developed algorithm a node counts uncolored neighbors, after this it distributes this information and its ID. If a node has an maximum of active neighbors and the biggest ID, it computes a possible coloring for all neighbors. Then it sends the color suggestion to its neighbors. If a nodes has multiple suggestions, it ignores them all. Then all nodes compare the color suggestion with all neighbors. If there is no conflict the node accepts the color. If there is a conflict, the node ignores the suggestion. A small optimization would be that the node with the bigger ID can accept the color.

4 Implementation

4.1 Difference to Model

Instead of simulating the message sending to all neighbors, we implemented a message polling in each round. Otherwise the simulation needs to much memory to keep track of all the messages. With this method we only need to store the message once.

4.2 Graph Generation

To test the different algorithms, we implemented 3 different graph generators. The first one is for sparse graphs, it creates d-Trees. These are trees where each node has d children at most. The second generator creates unit disk graphs. These geometric graphs are typically dense. The third is a simple random graph, where each edge exist with a probability p. This graph class contains graphs in the ”middle“ of the other two, you can create sparse or dense graph by adjusting the parameter p.

4.3 Library

We decided to use the JUNG graph library for Java[1], to implement my algorithms. We used this library is because it has built in support to visualize graphs. The provided data structure to store the nodes and edges is working for small amount of nodes. Because of the limitation for large graphs, we implemented my own

[1] Java Universal Network/Graph Framework `http://jung.sourceforge.net/`

graph class that supports more nodes, but only the most basic functions. And we only used the other library for visualization.

4.4 Visualization

We decided to implement the ability to visualize the result of an algorithm. This is useful when implementing the algorithms. Its easier to see if they are working correctly. The visualization also helps to verify that all border cases are covered. Due to memory usage, as mentioned above, of the library we had to limit the visualization to graphs with no more than 50 nodes. But thats more than enough to see if the algorithm works like expected.

4.5 Design

The implementation uses the strategy pattern for the coloring algorithms. Each algorithm must inherit the methods from an abstract coloring class. It has its own creating routine, because the algorithms uses different parameters. But all must implement the abstract functions. There are different graph containers, all are subclasses of the basic graph container. This container has the visualization function already implemented. The containers differ in the graph creation function, one container for each graph type. The main class parses the command line input, starts the graph creation, creates the different strategies. Then they are tested on the graph.

5 Results

We tested the algorithms only on the random graph with different probabilities for the edges to create sparse and dense graphs. We used different number of nodes, but we present the results based on the statistic with 1000 nodes. Because the statistics are similar for sufficiently many nodes (i.e. about 1000), it is enough to show the differences between the different algorithms.
We grouped the algorithm into deterministic and randomized algorithms. One big difference between the algorithms we compared is the fact that the randomized need more colors. This is because the nodes choose randomly one color. On the other hand the deterministic algorithms makes sure that the node who chooses a color can pick any color safely, this means they take the smallest free color and reduce the number of used colors. The performance of the randomized algorithms depends on the number of used colors, deterministic are independent.
We have two different types of figures to present our results. The Figures 5.1, 5.2 and 5.3 shown the number of rounds needed, when the algorithm could choose colors in the specified range. In the Figures 5.4, 5.5 and 5.6 one can see the actual number of colors the algorithm has used.

5.1 Randomized Algorithms

In dense graphs the Directed Edge algorithm needs usually more rounds than Luby as seen in Figures 5.1, 5.2 and 5.3. Directed Edge uses Luby every second round and in the other only the nodes with outgoing edges are colored. In dense graphs the one with outgoing edges can block a lot of other nodes. As with Luby they have the chance to get a color. In sparse graphs their performance is about the same. A node has low degree and thus does not block many other nodes. The Color Trail

algorithm is faster than the other two. In Luby the nodes only choose one color in each round compared to Color Trail where they try all at once. Because of that there is the higher probability that there is at least one color that is safe for each node.

5.2 Deterministic Algorithms

The performance of the Max Clustering algorithm depends on its radius. In graphs, like trees, a small radius is better. In all other cases a bigger radius is better (The graph in Figure 5.1 is alreday too dense). With this algorithm a lot of nodes are not allowed to choose a color, because they are blocked by a node with a larger ID in their $2r + 1$-neighborhood. These larger ID nodes are possibly also blocked by some bigger node in their neighborhood. This leaves a large part of the graph inactive. In tree like graphs a big radius can still only color a small number of nodes and leaves the bigger part untouched. With a small radius there is the possibility that more nodes can be active, in the same phase. In dense graphs one node can also block a large part of the graph, but their r-neighborhood also gets bigger and therefore it can color a lot of nodes (Figure 5.3).
Delta Max performs equally well for sparse and dense graphs. In sparse graphs (Figure 5.1) the problem with a node blocking another is not possible, because it only uses a small radius. In dense graphs (Figure 5.3) it colors always a big amount of the graph, as the node chosen by the algorithm has the biggest neighborhood.
The Splitted ID algorithm performs reasonably on sparse graphs (Figure 5.1). The node with the biggest number has only as small neighborhood and this allows other nodes to chooses a color too. The performance in dense graphs is bad (Figure 5.2 and 5.3). The problem is the less significant bits of the ID are discarded in the first round. Nodes with a large ID do not need to have a big number regarding only the less significant bits. There could be a node with a small ID, but a great number in the less significant bits. The nodes with big IDs discard now there less significant bits. This leads to many nodes with the same new value. So the algorithm needs another phase, with the small ID passive, until all bigger IDs are done.

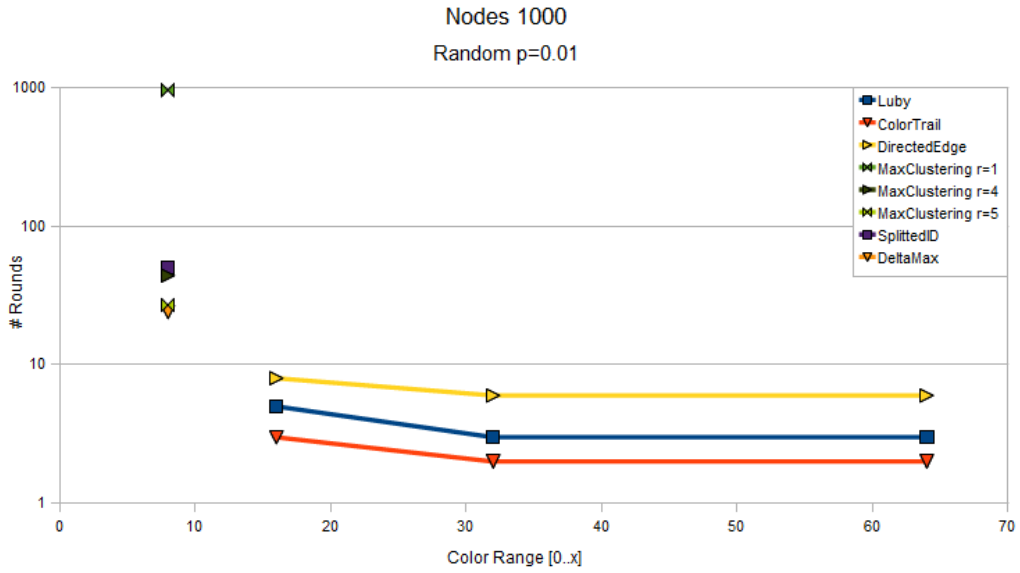

Figure 5.1: Random Graph 1000 nodes, edge probability p=0.01. Number of rounds needed, when the algorithm chooses colors in the specified range.

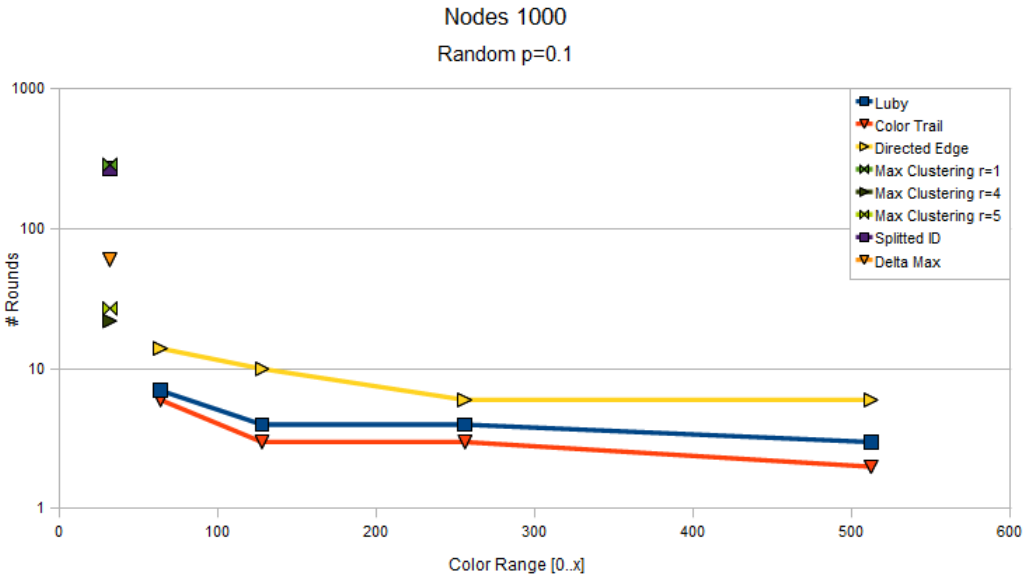

Figure 5.2: Random Graph 1000 nodes p=0.1. Number of rounds needed, when the algorithm chooses colors in the specified range.

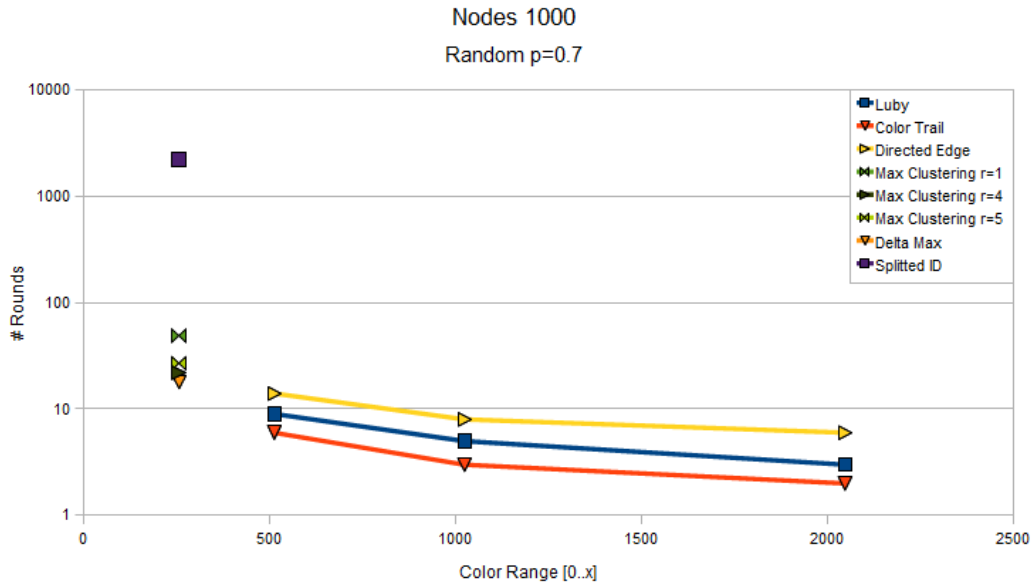

Figure 5.3: Random Graph 1000 nodes p=0.7. Number of rounds needed, when the algorithm chooses colors in the specified range.

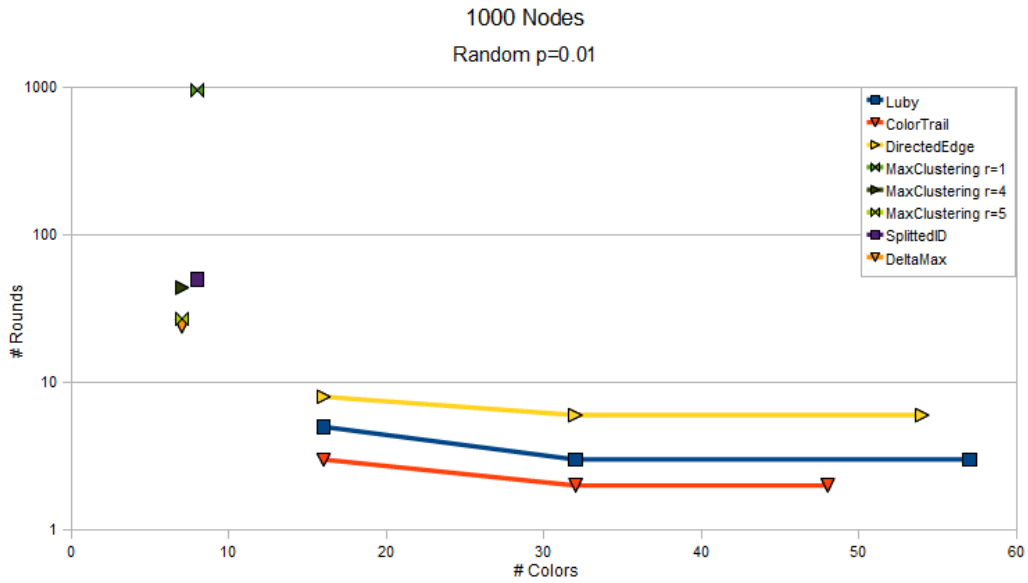

Figure 5.4: Random Graph 1000 nodes, p=0.01. Number of rounds needed and the actual number of colors the algorithm has used.

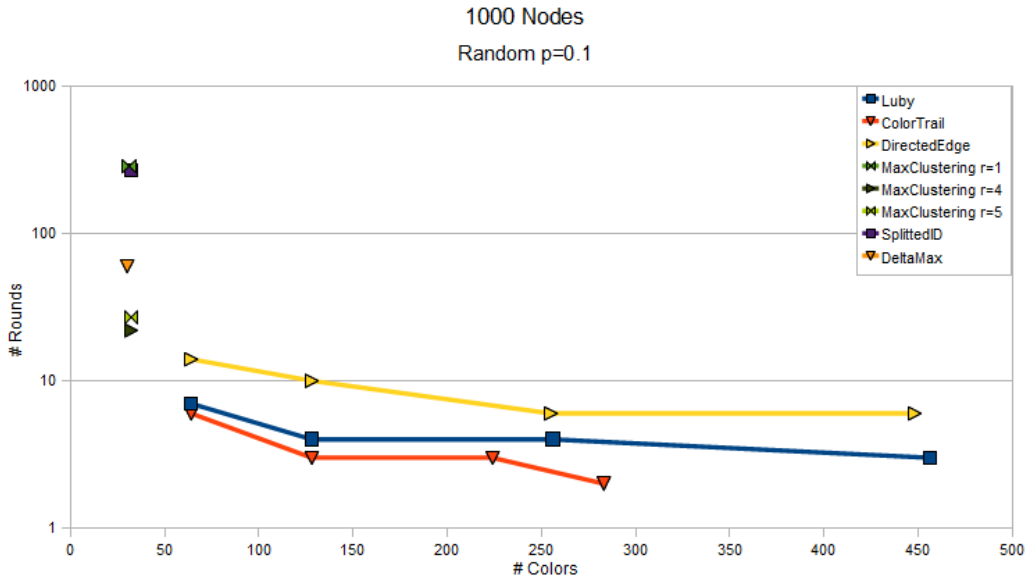

Figure 5.5: Random Graph 1000 nodes p=0.1. Number of rounds needed and the actual number of colors the algorithm has used.

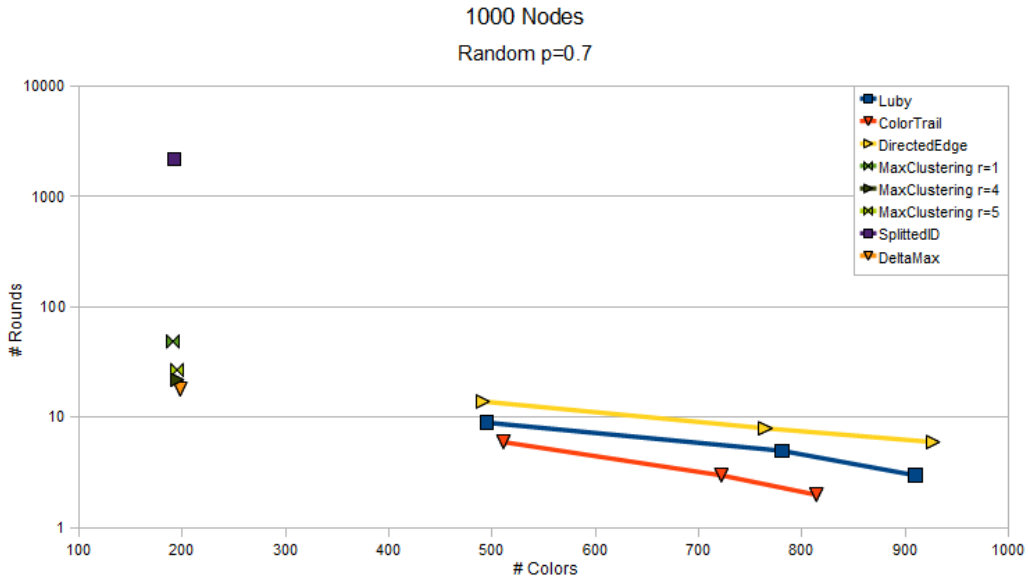

Figure 5.6: Random Graph 1000 nodes p=0.7. Number of rounds needed and the actual number of colors the algorithm has used.

6 Conclusion

6.1 Future Work

We see two major parts for future work. One is the design of new algorithms. Which then can be tested and compared to the already existent ones. We think there should be an algorithm, that performs good in both dense and sparse graph. Furthermore the algorithm could be tested with more constraints, like the limitation of message size or perhaps on computational power on each node. In sensor networks minimal power consumption is always important. And sending huge messages between nodes costs the most energy. We believe some of these algorithms could be adapted to small messages with small changes. Another small addition which could be useful is to read a special graph from a file. With this one could create some graphs with well defined properties and perhaps compute the result before running the simulation. And the verification is easier.

6.2 Difficulties

One big problem was the invention of a new algorithm, it is not easy to verify that it works for all different cases. There are a lot of small problems we did not think of until, we implemented the algorithms and started testing. The implementation had two problems, the first is to verify your algorithms are implemented correctly. And the other was to reduce the memory usage, by changing the model to pull the messages and not to push.

6.3 Improvements

Because we had some problems with the graph library, the next time we would implement it on our own and only use a library for visualization. So the library would be more specific for this task, but perhaps more efficient. The second change we could imagine is a framework for the algorithms, which would simplify the message transfer. So someone can implement a new algorithm more easily, by simply specifying the message creation and how to parse the messages.

Bibliography

[1] Kishore Kothapalli, Christian Scheideler, Melih Onus, and Christian Schindelhauer. Distributed coloring in O($\log^{1/2}$ n) bit rounds. In *International Parallel & Distributed Processing Symp. (IPDPS)*, 2006.

[2] M. Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. *SIAM Journal on Computing*, 15:1036–1053, 1986.

[3] Johannes Schneider and Roger Wattenhofer. A New Technique For Distributed Symmetry Breaking. In *Symp. on Principles of Distributed Computing(PODC)*, 2010.

[4] Johannes Schneider and Roger Wattenhofer. Distributed Coloring Depending on the Chromatic Number or the Neighborhood Growth. In *SIROOCO*, 2011.