Graph Coloring

Ingeborg Ødegård Oftedal, Paul Philip Mitchell, Hallvard Jore Christensen, Kyrre Laugerud Moe

UCSB CMPSC 290B
Spring 2015

Abstract

This report was written during the final project for the course CSPMSC 290B: Java-Centric Cluster & Concurrent Computing at the University of California, Santa Barbara. The purpose of this report is to describe the requirements, technical issues, architecture and experiments performed in our project, which was to solve the Graph Coloring problem using distributed computing.

Introduction

The Graph Coloring problem consists of coloring all adjacent vertices with different colors, using as few colors as possible. The smallest number of colors required to fulfill this requirement is called the chromatic number. Computing the chromatic number is NP-hard and the best known approximation algorithm yields the following complexity:

$$O(n\ (\log n\ )^{-3}\ (\log \log n\ )^{2})$$

For this project we have decided to stick to deciding whether or not a given graph admits a *k*-coloring for a given input domain size of colors, *k*. A coloring of a graph using at most *k* colors is called a *k*-coloring. Deciding whether or not a graph admits to a k-coloring is NP-complete except for the cases k = 1 and k = 2, and by using the principle of inclusion-exclusion and Yates' algorithm, the computational complexity is;

$$O((2\text{^}n)n)$$

We think that our project is of interest because Graph Coloring can be used to solve a variety of problems in many applications, including scheduling problems such as assigning aircrafts to certain flights, register allocation in compilers, and more recreational problems such as Sudoku solving.

Our primary goal is to color a graph where the chromatic number is known *a priori*. An additional goal is to calculate the chromatic number of the graph. A suggested approach is to start with a color domain size *k* that is probably too low to color the graph successfully. When no valid solution is found, *k* is incremented. This process continues until a valid solution is found. Another solution is to start with a *k* that is probably too big and decrement when a solution is found. This is probably a better approach, primarily because discovering a graph is un-colorable can be very time consuming. In our solution, which utilizes Best-First Search to reach a successful coloring fast, this would mean we would have to explore every combination. This would counteract the point of using Best-First Search with a good heuristic, and would use an excessive amount of time.

Functional Requirements

1. Successfully color a graph with a known chromatic number. To test this, we will run our system on graphs with known chromatic numbers, and validate that the proposed solution follows the constraints of the graph coloring problem.
2. Update a client side GUI representation of the graph, every time the “best partial solution” is updated. By best partial solution, we essentially mean the state that has yielded the lowest heuristic (where lower is better) at any point in the calculation. To test this we will simply run the simulation, and see if the GUI updates accordingly.

Performance Requirements

1. Be able to color randomly generated graphs with 100 vertices in less than 1 minute, only using one computer. To test this we will run the system on multiple randomly generated graphs containing 100 vertices and evaluate the roundtrip time as seen from the client. We will also validate that the system returns a valid solution.
2. Check idle time of each computer for 1 and 2 computers. Here we wish to reach an ideal time of no more than 25 %. To test this we will implement a logger that logs idle time for each instance of a Computer. The system will then run on a randomly generated graph.

Key Technical Issues

1. The primary technical issue will be to get the system to a working state. This is because our system has become somewhat complex as a result of keeping the design as close as possible to the *Cilk* architecture. Graph coloring is also quite theoretical. There will be a bunch of data structures, and states that will need to be managed as the system evolves. As things are moving a lot around in our distributed system, debugging a unsuccessful algorithm.

2. For Graph coloring the shared object is not very useful for our algorithm itself, since we're looking for a valid solution and the Computers are not very dependent on each other to find it. Still, the shared object is really useful for another issue of ours, namely continuously updating the graphics. As the domains of the vertices are reduced to singletons, better partial solutions are found.We plan to check these against the shared-object and update if it is better than its current state. The client will be notified when the shared object is updated for it to redraw the graphics. This will happen continuously throughout the process giving the consumer an indication of

what progress the Computers has made.

Architecture

The general architecture of the API is basically the same as for the homeworks; since it was general enough to adapt the graph coloring problem there is little we had to alter.

Like most groups we make use of the proxy design pattern for the computers populating the Space. Another thing we did was to make a SpaceProxy also inheriting from space, to support asynchronous calls with higher fault tolerance. SpaceProxy starts a Thread handling a waiting queue for calls to the space implementation instance. All communication from the computers to the space implementation goes through SpaceProxy.

While implementing we tried to follow the Cilk paper as closely as we could. We ended up with wrapping the Task in a closure, which space receives and runs. When a child task has a result a continuation will be sent to the parent Closure. When this was used to solve TSP, the solution was composed when all children had finished their calculation. But in Graph Coloring, children are looking for valid solution in different subsections of the search tree. This means that if a child finds a solution it is already “composed”. This means that the compose method’s function is simply to send the solution up through the stack. This is somewhat impractical, and lowers performance by a tiny margin. We chose to ignore this small inconvenience as it allowed us to maintain the Cilk architecture.

As mentioned before, all of the calculations happen in the decompose method. This involves deducing any domain reductions on the current state of the graph. When no further deductions can be performed the vertex with the smallest domain that is not a singleton is selected. An assumption for each of its domain colors is made, and a child is generated based on this assumption.

The system uses Best-First search. The heuristic we have used for graph coloring using Best-First Search is as follows. Sum the domain size minus one for each vertex in the graph.

Our shared-object "global" is partially distributed through piggybacking between the space and the computers when a Task is sent to a Computer. If a computer wishes to update the shared-object, it makes a call to a synchronous method in SpaceProxy.

Experiments

1 Computer	
rand-100-4-color1.txt	**rand-100-6-color1.txt**
8.365396% idle time	19.868371% idle time
14114 ms roundtrip time	16033 ms roundtrip time

2 Computers	
rand-100-4-color1.txt	**rand-100-6-color1.txt**
17.593712% idle time	27.456592% idle time
23044 ms roundtrip time	29439 ms roundtrip time

Conclusions [and Future Work]

In conclusion we would like to say that although the system is far from optimized, the overall performance exceeded our expectations. As we described in the performance requirements section, we wanted to make a system that could solve a *k*-graph containing 100 vertices in less than 1 minute. As you can see from our experiments we achieved this with a wide margin.

Because of time concerns we were not able to implement the part that allows us to color graphs where the chromatic number is unknown. As mentioned in an earlier section we believe that the best solution would be to start with a large *k*. This would find a valid solution relatively easily. Finding and coloring the graph with its chromatic coloring would however take longer time. Once the graph is colored with its chromatic number, the system would have to verify that this is the chromatic number. This would (in our system) involve trying to color the graph where k = c -1, where c is the chromatic number. As described earlier this would involve checking every permutation of the graph, which is very time consuming. On the bright side the graph would be colored before this process, and it is simply verifying that it is colored with the chromatic color that would take a while.