# COMS W4995 Parallel Functional Programming
# Parallel Genetic Algorithm for Graph Coloring

Milen Ferev

December 2021

## Graph Coloring

The Graph Coloring Problem is an NP-Complete problem which requires the vertices of a graph to be colored such that no two adjacent vertices have the same color. More formally, given a graph $G = \{V, E\}$ and a function $\delta : V \to C$ that maps nodes to a set of colors C, it is required that $|C|$ is minimal and that

$$\forall (u, v) \in E : \delta(u) \neq \delta(v)$$

The chromatic number $\chi(G)$ of a graph $G$ is defined as the minimum number of colors required to correctly color the graph. Due to the complexity of the problem, common approaches in literature focus on estimating bounds for $\chi(G)$, or searching over a set of values, trying to minimize an estimate $\bar{(}\chi(G))$. Techniques used in literature include Ant Colony Algorithms, Simulated Annealing, Genetic Algorithms, Backtrack Search, and others.

The Graph Coloring problem has many applications revolving around resource/task scheduling. To do this, every task is represented as a vertex and if two tasks cannot be executed at the same time an edge is drawn between them. After computing the coloring, all the tasks of the same color are non-conflicting and can be executed together.

## Genetic Algorithms

A Genetic Algorithm is an iterative search technique inspired by evolution. Over its runtime, an algorithm maintains a population of individuals, where each individual is a candidate solution. At each iteration a new population is generated using the current one. The individuals are altered via three operators: mutation, crossover, and selection. The mutation operator causes a random mutation in an individual. The crossover operator uses individuals from the current population to create a new individual. The selection operator is used to select individuals for crossover operator to be used. The quality of each solution is measured by a fitness function. The implementation details of each of the operators and the fitness function are specific to the problem. For this project, implementation details are outlined in Implementation Details.

# Problem Formulation

For this particular implementation, the Genetic Algorithm (GA) is used as a search over possible values of $\chi(G)$. The approach consists of fixing the number of colors $k$ and running the search algorithm in order to find a valid coloring. The bound considered is $1 \leq \chi(G) \leq |V|$. A linear search is performed over this range.
At every iteration a selection operation is applied to select the fittest individuals. Then from this set of individuals a new population is created via the mutation and crossover operations. The search over a particular value of $k$ is terminated if a solution is found, or if the maximum number of iterations has been reached.

# Implementation Details

## Data Representation

- Graph
  The graph is represented using the *Data.IntMap* module. The module is an equivalent to a hashmap where the keys are integers. Each vertex of the graph is indexed by an integer, with enumeration starting from 0. Each key in the IntMap refers to a particular vertex and maps to a list of integers, which are the neighbors of the vertex.

- Individual and Population
  An individual is given by an IntMap where each key is a particular vertex. The keys map to integers, where the integer represents the color of that vertex. For a search over k colors, the colors are ennumerated by integers in the range $[0, k)$. The implementation uses IntMap instead of a list for this purpose, since when parallelizing the fitness function (see below), the colors of arbitrary vertices need to be obtained and an implementation using list would have a slower overall runtime. In the implementation, the populations are represented by a list of individuals.

## Operators and Fitness

- Fitness
  The fitness function maps an individual to an integer. In the implementation, the fitness of an individual is defined as the number of vertices for which there is a neighbor with the same color. In the algorithm, the fitness is minimized. A fitness of value 0 implies that the given individual is a solution.

- Mutation
  The mutation operator takes in an individual and creates a new one. The mutation is done by finding the first vertex in the individual with a neighbor of the same color and changing that individual's color to a random one.

- Crossover
  The crossover operator takes two individuals (parents) and returns a new one. The new individual is created by selecting a random vertex k. Then the vertices up to k have the colors found in the first parent and after k have the colors found in the second parent.

- Selection
  The selection function takes in a population and returns a new (smaller) population. The input population is broken into chunks. Then for each chunk the individuals are sorted by fitness and the top individuals in each chunk are selected. Then the chunks are merged together where they are sorted by fitness.

## Parallelism

Since the data is represented using IntMap and List, and on each iteration a new population is generated, the algorithm is very susceptible to parallelization. In this implementation I have parallelzied some of the operators, and the others are applied in parallel over the population/individual. For implementing parallelism, the implementation uses the Eval monad. Since most of the operations that can be parallelized are of the same complexity for each vertex/individual, no additional load balancing is added. Most of the parallelism is applied using a *parMap* type of strategy, as the structures the algorithm uses are of a sequential nature.

- Fitness
  The fitness function has a subroutine which takes in a vertex, looks up the neighbors and then checks if any of the neighbors have the same color. Since each of the vertices in an individual can be processed independently of each other, the subroutine is applied in parallalel for all the vertices on the individual.

- Mutation
  The mutation operation itself is not parallelized, as it needs to create a new individual. However, when mutating a population, the operator is applied in parallel to the individuals in the population.

- Crossover
  Similar to the Mutation operator, the crossover operator produces a new individual and itself cannot be parallelized. However, it is also being applied in parallel over pairs of individuals in the population.

- Selection
  The selection operator uses parallelism to process each chunk. Initially the population is broken into chunks. Then for each chunk, in parallel, the individuals are sorted by fitness and the top few are chosen. Then the chunks are merged and sorted (this is not done in parallel).

### Algorithm Outline

The algorithm starts the search from the number of vertices. The execution is done by the *searchRoutine* function. For each of the iterations, a subset of the population is chosen by the selection operator. Then some portion of it is mutated by applying the mutation operator in parallel, and some portion is crossed over. In particular, the crossover is done between adjacent individuals (i.e indices i, i+1). Then the mutated and crossedover individuals are concatenated into a new population. The search ends once a solution has been found or the number of iterartions exhausted. If a search is not successfully, the algorithm returns the last value that has been successful.

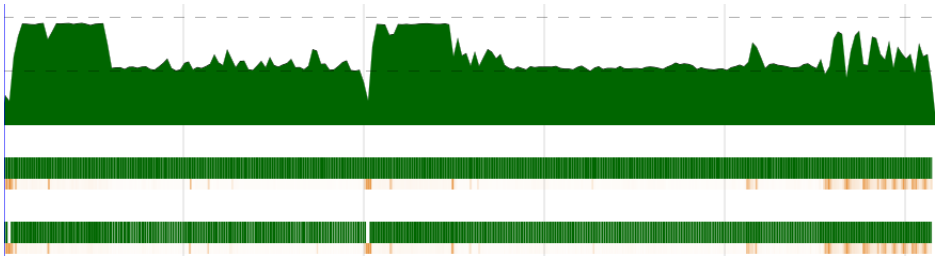
### Randomness

Since the algorithm relies on randomness, in the current implementation this is done with the *mkStdGen* with seed that is changed for every k (in particular, the number of colors k is used as a seed). When a new search over some k is started, a population of random individuals is generated using the value of k as a seed. The value is also used for picking the crossover index in the crossover operation (i.e for each k the value is fixed due to the properties of the language, however it changes from iteration to iteration). For the mutation operation, the current conflicting color plus the current value of k is used as a seed for the new color

## Results

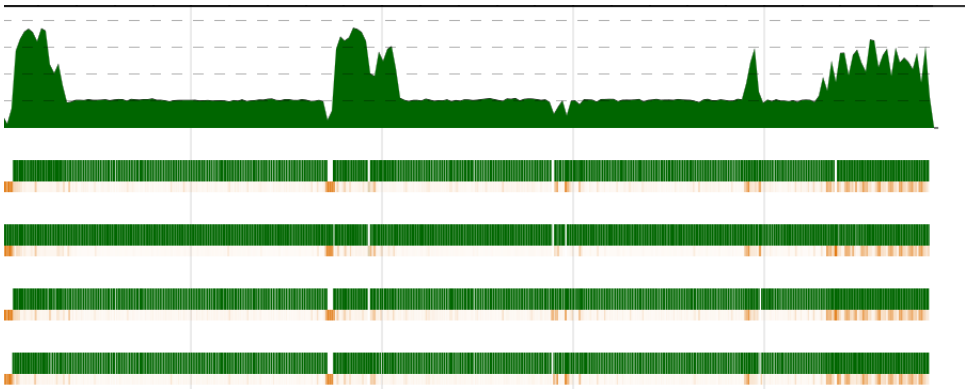
The algorithm was tested on standard Graph coloring tests provided by DIMACS. A link to all tests can be found in the README. The tests were done using the files myciel6.col and dsjc250.5.col. The algorithm was executed on a machine with Intel Core i7-5600U CPU, on Ubuntu 20.04.3 LTS. The graph dsjc250.5.col has 250 vertices and 31336 edges (and is one of the harder tests in DIMACS) and the graph myciel6.col has 95 vertices and 755 edges.

- myciel6.col with -N2



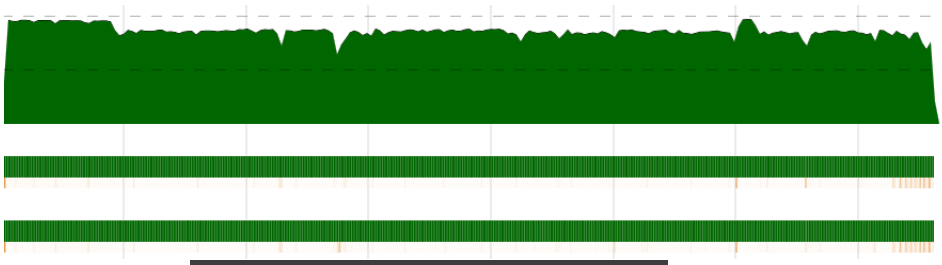
| | |
|---|---|
| Total time: | 2.58s |
| Mutator time: | 2.41s |
| GC time: | 0.17s |
| Productivity: | 93.3% of mutator vs total |

- myciel6.col with -N4



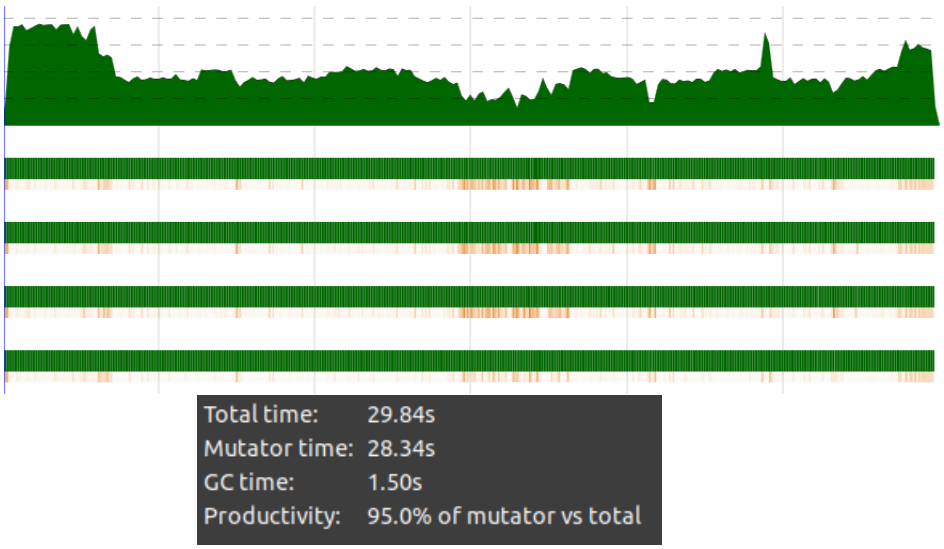
| | |
|---|---|
| Total time: | 2.44s |
| Mutator time: | 2.24s |
| GC time: | 0.20s |
| Productivity: | 91.8% of mutator vs total |

- dsjc250.5.col with -N2



| | |
|---|---|
| Total time: | 38.07s |
| Mutator time: | 36.89s |
| GC time: | 1.19s |
| Productivity: | 96.9% of mutator vs total |

- dsjc250.5.col with -N2




From the results it can be seen that adding more threads speeds up the result. For the smaller graph this is not as significant. For the larger graph the speedup is much more significant.

## Future Improvements

- Search Strategy
  Theoretically the current search strategy works, since given an optimal coloring of k colors, a coloring of k+1 colors can be generated by coloring a vertex with that color. However, in this case this is is not optimal because the algorithm does not always find a solution. Thus due to the current search strategy, it tends to severely overestimate the value by stopping at the first value of k for which it did not find a solution. Alternative strategies were also considered. For example one of the candidate strategies in consideration was to spawn a search for every k in the range. However this was not practical due to two the need for an adequate load balancing, since the complexity/duration of each search could increase with changes in the value of k, and because this caused too many sparks to get generated and the process to get killed (even for populations of sizes less than 1000).

- Random Number Generation
  The random number generation can be improved. To do this, the stdGen in the *System.Random* module can be used. However due to the properties of the language this would require the entirety of the code to be wrapped within a monad.

- Hyperparameters
  The hyperparameters (i.e population size, sizes of mutation and crossover, etc.) can be further optimized. In the current implementation a population size of 1000 is used. However population sizes up to 8000 were used. The value 1000 was chosen since this scales better for large graphs, whereas higher values cause the process to get killed.

## Sources

- The algorithm was loosely based on
  Abbasian, Reza, and Malek Mouhoub. "An efficient hierarchical parallel genetic algorithm for graph coloring problem." Proceedings of the 13th annual conference on Genetic and evolutionary computation. ACM, 2011.
- Lecture Notes, Video Lectures
- Hackage for Data.IntMap, Data.List, System.Random
- VS Code syntax suggestions

## Code Listing

- Graphs (some lines broken for visibility)

```
   module Graphs where

import qualified Data.IntMap.Strict as IntMap
import Control.Parallel.Strategies (runEval, rpar, Eval)
import Control.Monad (foldM)
import Data.Maybe (isJust)
import System.Random  (randomRs, mkStdGen)
import System.IO(openFile, hGetContents, IOMode( ReadMode ))
import Data.List (sortBy)

type Vertex = Int
type Color = Int
type Graph = IntMap.IntMap [Vertex]

type Individual = IntMap.IntMap Color
type Population = [Individual]

seed :: Int
seed = 1337

populationSize :: Int
populationSize = 1000

iterations :: Int
iterations = 10

numOfChunks :: Int
numOfChunks = 5

numToSelect :: Int
numToSelect = 100

numToMutate :: Int
numToMutate = 200

numToCrossover :: Int
numToCrossover = 800

succ1 :: Int -> Color -> Color
succ1 = getRandomColor
```

```
getRandomColor :: Int -> Int -> Int
getRandomColor k col = head  $ take 1  $ randomRs (0, k-1) (mkStdGen (k+col))

-- map a function over a list with Eval
mapParMap :: (a -> b) -> [a] -> Eval [b]
mapParMap _ [] = return []
mapParMap f (a:as) = do
    b <- rpar (f a)
    bs <- mapParMap f as
    return (b:bs)

-- Fitness function
fitness :: Graph -> Individual -> Maybe Int
fitness graph individual = do
    let counts = runEval (mapParMap (checkVertex graph individual)
        (IntMap.toList individual))
    foldM (fmap . (+)) 0 counts

-- checks if vertex has neighbors of same color
checkVertex :: Graph -> Individual -> (Vertex, Color) -> Maybe Int
checkVertex graph individual (vertex, color) = do
    neighbors <- IntMap.lookup vertex graph
    colors <- mapM (`IntMap.lookup` individual) neighbors
    if color `notElem` colors then return 0 else return 1

-- mutation operator
mutate :: Graph -> Int -> Individual -> Individual
mutate graph colors individual = IntMap.fromList(f (IntMap.toList individual))
    where
        f :: [(Int, Color)] -> [(Int, Color)]
        f [] = []
        f ((k,v):xs) = if checkVertex graph individual (k,v) == Just 0
            then (k,v) : f xs
            else (k, succ1 colors v) : f xs

-- crossover operator
crossover :: Int -> Individual -> Individual -> Individual
crossover col ind1 ind2 = IntMap.fromList (f l1 l2 k)
    where
        l1 = IntMap.toList ind1
        l2 = IntMap.toList ind2
        k = getRandomCrossover col (length l1)
        f :: [(Int, Color)] -> [(Int, Color)] -> Int -> [(Int, Color)]
        f [] [] _ = []
        f [] lst2 _ = lst2
        f lst1 [] _ = lst1
```

```
        f (x:xs) (_:ys) depth = if depth /= 0 then x : f xs ys (depth-1) else ys


-- split population to chunks
splitToChunks :: Int -> [a] -> [[a]]
splitToChunks n l = f n l where
    (k,_) = quotRem (length l) n
    f :: Int -> [a] -> [[a]]
    f _ [] = []
    f nch lst = if nch == 0 then [lst] else chunk : f (nch-1) tl
        where
            (chunk, tl) = splitAt k lst

-- sort a chunk
sortChunk :: Graph -> [Individual] -> [Individual]
sortChunk graph = sortBy (\x y -> compare (fitness graph x) (fitness graph y))

-- select K from population
selectK :: Graph -> Int -> Int -> Population -> Population
selectK graph nchunks k population = sortChunk graph $ concatMap (take k)
    chunksSorted where
    chunks = splitToChunks nchunks population
    chunksSorted = runEval $ mapParMap (sortChunk graph) chunks

-- used to read row for dimacs
readRow :: String -> [(Int, [Int])]
readRow input = case head row of
    ['e'] -> [(a,[b]), (b,[a])]
    _ -> []
    where
        row = words input
        a' = read ((head . tail . tail) row) :: Int
        b' = read((head . tail) row) :: Int
        -- since vertices are 1-indexed
        a = a'-1
        b = b'-1


readGraphDIMACS :: String -> IO Graph
readGraphDIMACS filename = do
    h <- openFile filename ReadMode
    contents <- hGetContents h
    let edgeList = concatMap readRow (lines contents)
    let graph = IntMap.fromListWith (++) edgeList
    return graph
```

```
getRandomNums :: Int -> Int -> [Int]
getRandomNums k size = take size  $ randomRs (0, k) (mkStdGen k)

getRandomCrossover :: Int -> Int -> Int
getRandomCrossover k individualSize = head  $ take 1  $
    randomRs (0, individualSize-1)  (mkStdGen k)

-- creates population
initializePopulation :: Int -> Int -> Population
initializePopulation individualSize colors = population
    where
        numbers = getRandomNums colors (individualSize*populationSize)
        individualChunks = splitToChunks populationSize numbers
        population = map (IntMap.fromList . zip [0..(individualSize-1)])
            individualChunks

-- overall search
searchK :: Int -> Graph -> Maybe Int
searchK k graph = do
    let individualSize = IntMap.size graph
    if k == 0 then
        return individualSize
    else do

        let population = initializePopulation individualSize k
        res <- searchRoutine graph population k iterations
        if res then do searchK (k-1) graph
        else return k

-- search for particular k
searchRoutine :: Graph -> Population -> Int -> Int -> Maybe Bool
searchRoutine graph population k iter = do
    if iter == 0 then return False else do
        let newPopulation = selectK graph numOfChunks numToSelect population
        let fits = filter isJust (map (fitness graph) newPopulation)
        let solution = Just 0 `elem` fits
        if solution then return True
        else do
            let mutated = runEval $ mapParMap (mutate graph k)
                (take numToMutate newPopulation)
            let crossedover =  runEval $ mapParMap (uncurry (crossover k))
                (take numToCrossover [(x,y) | x <- cycle newPopulation,
                    y <- tail (cycle newPopulation)])
            let nextPopulation = crossedover ++ mutated
            searchRoutine graph nextPopulation k (iter-1)
```

- Main

```
module Main where
import Graphs
import System.Environment(getArgs)
import qualified Data.IntMap.Strict as IntMap
import Control.DeepSeq (force)

main :: IO()
main = do
    args <- getArgs
    case args of
        [filename] -> do
            graph <- force readGraphDIMACS filename
            let initialK = IntMap.size graph
            print (searchK initialK graph)
        _ -> error "Bad Input"
```