COMS W4995 Project Report - Graph Coloring

Jacky Huang (jjh2231)

December 2025

1 Introduction

Graph Coloring is a classic NP-Complete problem involving assigning labels ("colors") to the vertices of a graph in a way such that no two adjacent vertices share the same color. Formally, given an undirected graph $G = (V, E)$, a coloring of G is an assignment c of (integer) labels to each vertex $v \in V$, subject to the constraint that for all edges $(v_1, v_2) \in E$, we have $c(v_1) \neq c(v_2)$. A k-coloring of a graph G is a coloring which uses at most k labels, and a graph is k-colorable if there exists a valid k-coloring of G. We will focus on the k-colorability problem for this project; given a graph $G = (V, E)$ and an integer k, is G k-colorable?

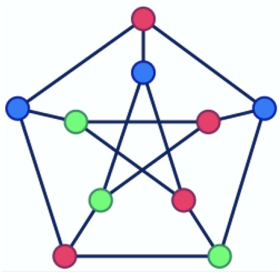

Figure 1: This is an example of a 3-coloring for the Petersen graph. Here, no two adjacent vertices share the same color. It turns out that 3 is also the smallest integer k such that the Petersen graph is k-colorable. The smallest such k is known as the *chromatic number* of a graph. Image obtained from [1]: `https://upload.wikimedia.org/wikipedia/commons/9/90/Petersen_graph_3-coloring.svg`

2 Algorithms

Note that this paper will focus only on exact algorithms, as the runtime of approximate algorithms for graph coloring are usually by default much faster than exponential time and hence is not a valid comparison of run time.

2.1 Brute Force

There is an obvious brute force algorithm which we implemented as a baseline; simply try out all k^n possible colorings for the n vertices in the graph, and see if any of them satisfy the constraints. Note that given a candidate coloring, it is trivial to check the legality of that coloring in polynomial time. This is very slow in practice for even moderate sizes of n; consequently, it was only able to solve only one of our test graphs due to our enforced timeout during testing.

2.2 Pruning

One simple optimization over the baseline algorithm can be done by backtracking with pruning. During the backtracking process, we can run legality checks on the partial colorings at each node; as soon as we detect a conflict with the partial coloring at that point in the execution, we can backtrack immediately instead of continuing to explore further down the branch.

2.3 Parallelized Pruning

A natural extension of this algorithm to the parallel case is by noticing that each branch of the backtracking can be evaluated independently of each other; as a basic example, when working with the first vertex in the graph we can examine each of the k possible colorings in parallel. The parallelism can easily be extended to the desired degree in the same manner by parallelizing further subtrees. In this project, we use `parList` `rdeepseq` as the parallelism construct of choice for handling branch parallelism in the search tree in order to force the direct evaluation of each branch and reduce thunks. To avoid over-parallelizing the recursive calls, we wanted to only up to a certain depth. We experimented with different levels of maxDepth but ultimately settled on parallelizing the branching up to a tree depth of 3, and running further subtrees in a sequential manner.

2.4 DSATUR-based backtracking

On the other hand, the core algorithm at play is still too slow in practice even if it is parallelized. To this end, we explored further optimizations of the basic algorithm. As an NP-Complete problem, there are no known polynomial-time (exact) algorithms for graph coloring. However, there are some heuristics-based approaches to the backtracking which is shown to be much faster in practice compared to the baseline algorithm, albeit they have the same exponential worst-case complexity.

One such example is the DSATUR-based backtracking algorithm. This algorithm is based on the DSATUR heuristic, which orders the vertices by the *degree of saturation*, where the degree of saturation for a vertex v is defined as the number of distinct colors already assigned to the neighbors of v. Instead of processing the vertices in a random order, the core idea of this optimization

is to explore the vertices in descending order of the DSATUR at each point in the backtracking search. Note that the DSATUR of the vertices needs to be recomputed at each stage of the backtracking algorithm; as this involves independent calculations across each of the vertices in the graph, this allows for parallelism using `parList rdeepseq`. This strategy can also be combined with the branch-level parallelism mentioned above.

3 Methodology

3.1 Data

The first step was to find a set of graphs to use as our benchmarks. This was tougher than expected; we needed to find graphs which were non-trivial to solve in order to show improvement with the parallel schemes, however it also couldn't be too difficult to solve due to the NP nature of the problem causing exponential increases in run time if the graph was slightly more complex. Thus, we decided to use a collection of graph instances found at `https://mat.tepper.cmu.edu/COLOR/instances.html#XXMYC` [2] which are actively used in benchmarking graph coloring algorithms, in addition to coming up with our own graphs. Due to time constraints, we only explored a small subset of the available graphs on the website.

In total, we utilize 5 graphs for our benchmarks:

- petersen.txt: The Petersen graph in Figure 1, with a chromatic number of 3. This was used initially for verifying correctness.
- myciel4.txt: A graph based on the Mycielski transformation which has a chromatic number of 5. The Mycielski transformation is a procedure which starts with a one-edge graph, and iteratively transforms the graph to produce a larger graph with a strictly bigger chromatic number, while keeping the invariant that the new graph does not contain any triangles (cliques of size 2). Obtained from [2] (original source from Michael Trick at CMU).
- queen5_5.txt, queen6_6.txt: A reduction of the n-queens problem into k-coloring. Each square on the n by n chessboard is represented by a vertex, and edges between vertices represent that the two corresponding squares are on the same row, column or diagonal. Then, a placement of n queens exists if and only if an n-coloring of this graph exists. They have chromatic numbers of 5 and 7 respectively. Obtained from [2] (original source from Stanford GraphBase).
- complete_10.txt: A complete graph of order 10. These are graphs where all vertices are connected to each other, and hence the chromatic number must be the number of vertices in the graph itself. In this sense, it is a hard graph for a k-coloring algorithm to solve.

The graphs are represented in an adjacency list format. Each row starts off with the 0-indexed ID of the node, followed by a colon and then a space-separated list of its neighbors. See Listing 1 for the representation of the Petersen graph. Note that the graphs from the website are in DIMACS format; as a result we needed to convert it into the adjacency list format expected in our code.

```
0: 1 4 5
1: 0 2 6
2: 1 3 7
3: 2 4 8
4: 3 0 9
5: 7 8 0
6: 8 9 1
7: 9 5 2
8: 5 6 3
9: 6 7 4
```

Listing 1: Adjacency list representation of the Petersen graph.

3.2 Experimentation

The benchmarks were run on a November 2024 Macbook Pro using the Apple Silicon M4 Max chip, with 14 cores and threads, along with 36GB of RAM. The binaries were compiled with `stack --resolver lts-22.33 ghc --make -threaded -rtsopts -Wall -O2 solver.hs -o solver`.

For ease of reproducibility, we wrote a simple benchmark script which loops through every configuration we tested and runs the code. We elected to test only 2 values of k for each graph; the chromatic number itself as well as (chromatic number - 1), in order to get a fair representation of the behavior of the algorithms in both the YES and NO cases to the k-coloring problem; we observed that the YES scenarios tend to run quickly regardless of the setup (excluding brute force), however the NO scenarios where the value of k is sufficiently high were the most interesting cases in observing speedup. Due to time constraints, we also enforced a 2 minute timeout for each experiment. For the parallel algorithms, we also ran the code for 1 to 8 cores in order to measure the speedup. Finally, we ran each configuration 3 times and took the median number in order to reduce variance in our results.

The results were aggregated into a csv file for analysis.

3.3 Implementation

The graph and graph coloring objects were represented as Maps as shown in Listing 2. To accomodate the one-click-launch benchmark capabilities, it was necessary to expose all hyperparameter choices as command line arguments

to the Haskell program; these include the choice of algorithm, input graph file, value of k, and the number of cores for parallelism. Upon reading the command line arguments, the code then parses the graph file into the internal representation of a graph, and kicks off the relevant algorithm along with the specified configuration. Each algorithm has its own function, returning a `Maybe (GraphColoring)` object to denote whether a graph coloring was found or not.

```
type Graph = Map.Map Int [Int] -- adjacency list repr
type GraphColoring = Map.Map Int Int -- node to color mapping. 0
↪  is unassigned, 1 to k are the different colors
```

Listing 2: Haskell definitions of a graph and a graph coloring.

4 Results and Discussion

4.1 Sequential Algorithms

We firstly experimented briefly with the brute force algorithm on our datasets. As shown in Table 1, the efficiency of the algorithm was very poor, being able to solve only the basic Petersen graph under our 2 minute time constraint, while timing out for the other configurations.

Graph	K	N_threads	Output (is_k_colorable)	Time (s) - median of 3	Memory (K)
Petersen	2	1	No	0.34	804
Petersen	3	1	Yes	0.34	4001
Queen 5	4	1	N/A	N/A	N/A
Queen 5	5	1	N/A	N/A	N/A
Queen 6	6	1	N/A	N/A	N/A
Queen 6	7	1	N/A	N/A	N/A
Myciel 4	4	1	N/A	N/A	N/A
Myciel 4	5	1	N/A	N/A	N/A
Complete 10	9	1	N/A	N/A	N/A
Complete 10	10	1	N/A	N/A	N/A

Table 1: Results for brute force algorithm.

Moving on to our pruning optimization for backtracking was a lot more fruitful. As shown in Table 2, the pruning algorithm induced a 10x speedup over the standard brute force algorithm, and was able to solve the rest of the graphs as well in a reasonable time. Furthermore, the memory footprint was much smaller as well compared to the brute force.

On the other hand, the results for the sequential DSATUR algorithm as shown in Table 3 are a bit more surprising. While the times are identical for the easy Petersen and Queen 5 graphs, it looks like there is quite the variation for the NO instances for the other problems. Myciel 4 was significantly faster for

Graph	K	N_threads	Output (is_k_colorable)	Time (s) - median of 3	Memory (K)
Petersen	2	1	No	0.032	272
Petersen	3	1	Yes	0.032	335
Queen 5	4	1	No	0.033	705
Queen 5	5	1	Yes	0.032	1680
Queen 6	6	1	No	13.149	1684023
Queen 6	7	1	Yes	0.85	94809
Myciel 4	4	1	No	30.21	6435088
Myciel 4	5	1	Yes	0.031	894
Complete 10	9	1	No	9.48	3987455
Complete 10	10	1	Yes	0.036	588

Table 2: Results for sequential pruning algorithm.

DSATUR compared to pruning, while Queen 6 and Complete 10 were significantly slower for DSATUR than pruning. On the other hand, the YES instance of Queen 6 was faster by an order of magnitude for the DSATUR algorithm, but roughly in the same ballpark for the rest of the graphs. We hypothesize that the reason for the large discrepancy between Myciel 4 and Queen 6/Complete 10 is due to the characteristics of the graphs themselves. For example, Complete 10 is a graph of 10 vertices where each vertex is connected to every other vertex. This means that we should expect DSATUR to be *slower* than regular pruning; no matter what happens, the degree of saturation for each unassigned vertex is the exact same, meaning that the ordering in which we process the nodes is irrelevant. This actually makes DSATUR slower in practice than naive pruning due to the extra overhead associated with it. We suspect that a similar logic applies to the Queen 6 problem. On the other hand, the Myciel 4 is very favorable for running DSATUR. Again, we believe this is due to the characteristics of the graph allowing for DSATUR to be a useful heuristic. These examples show that the effectiveness of DSATUR depends heavily on the type of graph it is running on. The memory usage of the instances roughly followed the same trend as the time taken on each instance, which makes sense.

Graph	K	N_threads	Output (is_k_colorable)	Time (s) - median of 3	Memory (K)
Petersen	2	1	No	0.033	388
Petersen	3	1	Yes	0.032	373
Queen 5	4	1	No	0.032	4641
Queen 5	5	1	Yes	0.032	2492
Queen 6	6	1	No	25.758	56010962
Queen 6	7	1	Yes	0.065	51591
Myciel 4	4	1	No	0.547	1099596
Myciel 4	5	1	Yes	0.033	1266
Complete 10	9	1	No	21.969	22596606
Complete 10	10	1	Yes	0.032	786

Table 3: Results for sequential DSATUR algorithm.

Specifically looking at the memory statistics for the Complete 10 problem with $k = 9$, as shown in Listings 3 and 4, the DSATUR algorithm required far

more (around 5x) bytes in the heap. This makes sense as there are additional data structures and operations required for the DSATUR algorithm. The ratio of bytes copied during GC to the total number of bytes allocated in the heap is also roughly equal between the two algorithms.

```
Pruning:

4,083,154,712 bytes allocated in the heap
       1,301,672 bytes copied during GC
          85,968 bytes maximum residency (2 sample(s))
          30,040 bytes maximum slop
               6 MiB total memory in use (0 MiB lost due to
                ↪  fragmentation)

                                     Tot time (elapsed)  Avg
                                      ↪  pause  Max pause
  Gen  0       973 colls,     0 par    0.003s   0.004s
   ↪  0.0000s    0.0005s
  Gen  1         2 colls,     0 par    0.000s   0.000s
   ↪  0.0001s    0.0001s

  TASKS: 4 (1 bound, 3 peak workers (3 total), using -N1)

  SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.002s  (  0.002s elapsed)
  MUT     time    9.013s  (  9.008s elapsed)
  GC      time    0.003s  (  0.004s elapsed)
  EXIT    time    0.001s  (  0.005s elapsed)
  Total   time    9.019s  (  9.019s elapsed)

  Alloc rate    453,029,229 bytes per MUT second
```

Listing 3: Garbage collection statistics for sequential pruning on the complete 10 graph with $k = 9$ (NO instance).

4.2 Parallel Algorithms

Due to the embarrassingly parallel nature of the recursive subproblems, we observed significant improvements in the instances where k is not colorable. As a sanity check, we firstly verified that the times for the one-thread instance of these parallel algorithms roughly matched with the times shown in Tables 2 and 3, before running the instances with higher parallelism.

Figure 2 shows the speedup from increasing the number of threads for the parallel pruning algorithm. The left graph plots the instances where the graph

```
23,138,924,592 bytes allocated in the heap
       6,847,632 bytes copied during GC
          76,208 bytes maximum residency (2 sample(s))
          30,040 bytes maximum slop
               6 MiB total memory in use (0 MiB lost due to
                ↪  fragmentation)

                                     Tot time (elapsed)  Avg
                                      ↪  pause  Max pause
  Gen  0      5545 colls,     0 par    0.014s   0.021s
   ↪  0.0000s    0.0006s
  Gen  1         2 colls,     0 par    0.000s   0.000s
   ↪  0.0001s    0.0001s

  TASKS: 4 (1 bound, 3 peak workers (3 total), using -N1)

  SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.002s  (  0.002s elapsed)
  MUT     time   21.930s  ( 21.915s elapsed)
  GC      time    0.015s  (  0.021s elapsed)
  EXIT    time    0.000s  (  0.010s elapsed)
  Total   time   21.947s  ( 21.948s elapsed)

  Alloc rate    1,055,127,097 bytes per MUT second
```

Listing 4: Garbage collection statistics for sequential DSATUR on the complete 10 graph with $k = 9$ (NO instance).

is not k-colorable, while the right graph plots the instances where the graph is k-colorable. It can be seen immediately that the behaviour of the speedup is completely different. In the case where the algorithm is trying to find a k-coloring which can't exist, the speed up is very close to ideal; this is because the recursive subproblems it is trying to solve are easily parallelizable, and as a solution doesn't actually exist, the algorithm is forced to explore many paths before it is able to conclude that the graph is not k-colorable. This attribute of the problem naturally increases the runtime of the algorithm, and hence allows for larger improvements through parallelism over the sequential algorithm. On the other hand, when the problem instance is k-colorable, the algorithm simply needs to find one example of a k-coloring and it is able to conclude that the graph is k-colorable; as shown in Table 2, this is done quite quickly and hence there is much less room for speedup with scaling up the number of cores.

The behavior for the parallel DSATUR algorithm is very similar, albeit with slightly worse speedup as shown in Figure 3. This can be explained by the nature of the algorithm; DSATUR (with pruning) is more complex than naive

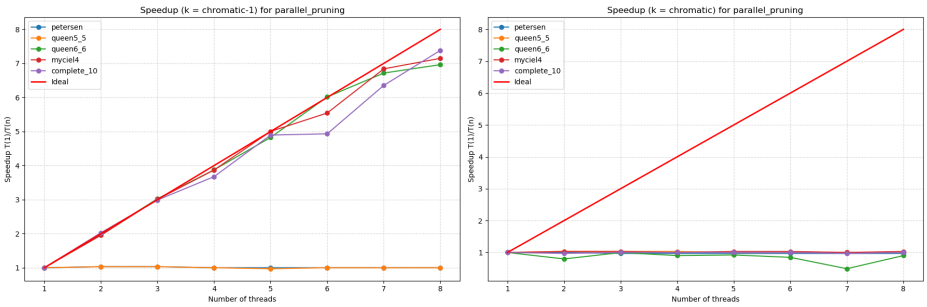

Figure 2: Speedup graphs for parallel pruning.

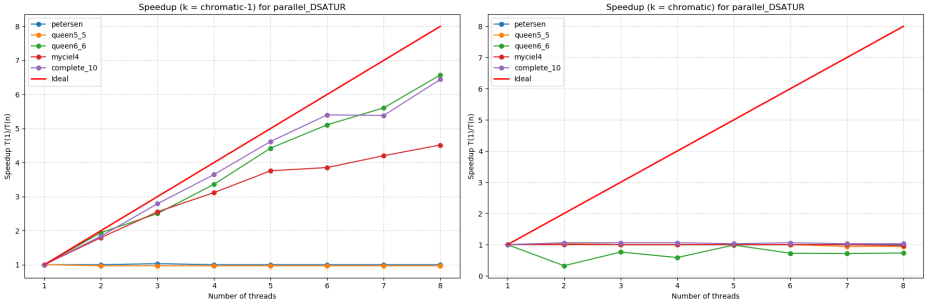

Figure 3: Speedup graphs for parallel DSATUR.

pruning and has more overhead costs in the form of auxiliary functions required. Some of these functions are harder to parallelize compared to naive pruning, which explains why the speedup (while still decent) is less than naive parallel pruning. On the other hand, the case where k = chromatic number follows the same behavior as in Figure 2; as before, the reason can be explained by Table 3. Interestingly, Amdahl's law suggests that the parallelizability of both algorithms are roughly equivalent, with p being in the range of 0.12 to 0.14 for both algorithms.

Furthermore, the speedup for both parallel pruning and DSATUR (k = chromatic - 1 case) look like they have yet to plateau; it is reasonable to assume that further increasing the number of threads will yield even greater improvements in the speedup. As one would expect from the speedup ratios, Figures 4 and 5 show that the Threadscope timeline is healthy.

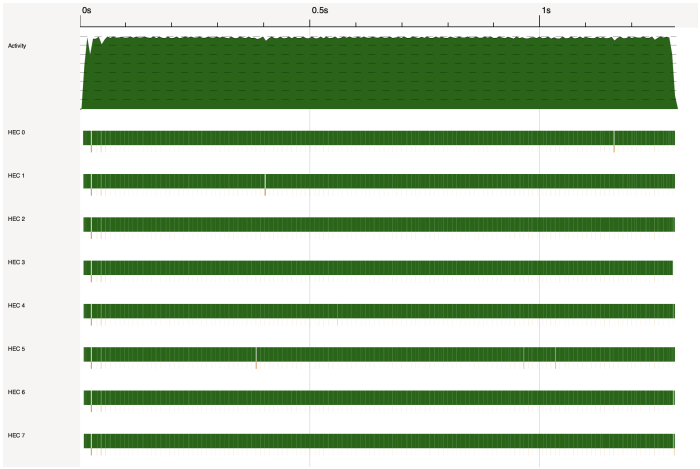

Figure 4: Threadscope graph for parallel pruning for the Complete 10 graph with $k = 9$.

5 Limitations and Future Work

One concerning issue is that a lot of the sparks are not being converted properly, with most of them being GCed as in Figure 6. It would be an interesting exercise to try and figure out where this inefficiency is coming from and try to fix that. Another work item would be to parallelize the checkColoring function for both parallel pruning and DSATUR. It would also be good to extend the number of threads in the above experiments and see how good the speedup is when the number of threads increases, as well as test on a more complete/harder suite of graphs. Finally, there are some easy improvements from a sequential perspective to be made, such as using IntSets and IntMaps for more efficiency.

Figure 5: Threadscope graph for parallel DSATUR for the Complete 10 graph with $k = 9$.

HEC	Total	Converted	Overflowed	Dud	GCed	Fizzled
Total	8893609	440	0	0	8711293	181876
HEC 0	1105797	124	0	0	1053420	36352
HEC 1	1108315	116	0	0	1066789	42735
HEC 2	1117371	81	0	0	1083579	40790
HEC 3	1116433	6	0	0	1110176	9464
HEC 4	1108386	25	0	0	1100461	9393
HEC 5	1108983	11	0	0	1095859	12233
HEC 6	1111912	57	0	0	1088497	25822
HEC 7	1116412	20	0	0	1112512	5087

Figure 6: Threadscope sparks view for parallel DSATUR for the Complete 10 graph with $k = 9$.

References

[1] Wikipedia, Graph coloring, accessed December 2025, https://en.wikipedia.org/wiki/Graph_coloring

[2] Michael Trick, Graph Coloring Instances, accessed December 2025, https://mat.tepper.cmu.edu/COLOR/instances.html#XXMYC

6 Appendix

The full Haskell code can be found below.

```haskell
import System.Environment (getArgs, getProgName)
import System.Exit (die)
import qualified Data.Map as Map
import qualified Data.Set as S
import Data.Maybe (catMaybes, listToMaybe)
import Control.Parallel.Strategies (using, parList, rdeepseq)


type Graph = Map.Map Int [Int] -- adjacency list repr
type GraphColoring = Map.Map Int Int -- node to color mapping. 0 is unassigned, 1 to k are the different colors

main :: IO ()
main = do
  args <- getArgs
  case args of
    [fileName, kStr, algo] -> runSolver fileName kStr algo
    _ -> getProgName >>= (\progName -> die ("Usage: " ++ progName ++ " <filename>" ++ " k" ++ " algo"))

-- general utils

loadGraph :: String -> Graph
loadGraph =
    Map.fromList . map processNode . lines
  where
    processNode :: String -> (Int, [Int])
    processNode nodeLine =
      let (nodeIDStr, neighborsStr) = break (== ':') nodeLine
          neighborsWithoutColon = drop 1 neighborsStr
          neighborList = words neighborsWithoutColon
          children = map read neighborList
          node = read nodeIDStr
      in (node, children)
```

```haskell

checkColoring :: Graph -> GraphColoring -> Int -> Bool -- for each node, assert that the coloring for each of the neighbors is not the same
checkColoring graph coloring k =
    Map.foldrWithKey checkNode True coloring
    where
        checkNode :: Int -> Int -> Bool -> Bool
        checkNode currNode color acc = acc && (color >= 0 && color <= k) && (checkNeighbors currNode color)
          where
            checkNeighbors :: Int -> Int -> Bool
            checkNeighbors node col = (col == 0) || not (any (\n -> Map.findWithDefault 0 n coloring == col) (Map.findWithDefault [] node graph))

returnMaybeColoring :: Graph -> GraphColoring -> Int -> Maybe (GraphColoring)
returnMaybeColoring graph coloring k
    | checkColoring graph coloring k = Just coloring
    | otherwise = Nothing

initColoring :: Int -> GraphColoring
initColoring n = (Map.fromList [(i, 0) | i <- [0..(n - 1)]])

-- BRUTE FORCE

runBruteForce :: Graph -> Int -> GraphColoring -> Int -> Int -> Maybe (GraphColoring)
runBruteForce graph k col v c
    | v == Map.size graph = returnMaybeColoring graph col k
    | otherwise = listToMaybe (catMaybes [ runBruteForce graph k (Map.insert v c col) (v + 1) c_ | c_ <- [1..k] ])

bruteForceAlgorithm :: Graph -> Int -> Int -> Maybe (GraphColoring)
bruteForceAlgorithm graph k n = listToMaybe (catMaybes [ runBruteForce graph k (initColoring n) 0 c | c <- [1..k] ])

-- Pruning

runPruning :: Graph -> Int -> GraphColoring -> Int -> Int -> Maybe (GraphColoring)
runPruning graph k col v c
    | v == Map.size graph = returnMaybeColoring graph col k
    | checkColoring graph col k == False = Nothing
```

```haskell
    | otherwise = listToMaybe (catMaybes [ runPruning graph k
    ↪  (Map.insert v c col) (v + 1) c_ | c_ <- [1..k] ])

pruningAlgorithm :: Graph -> Int -> Int -> Maybe
↪  (GraphColoring)
pruningAlgorithm graph k n = listToMaybe (catMaybes [
↪  runPruning graph k (initColoring n) 0 c | c <- [1..k] ])


-- DSATUR

unassignedNodes :: GraphColoring -> [Int]
unassignedNodes coloring = Map.foldrWithKey (\k v acc -> if v
↪  == 0 then k : acc else acc) [] coloring

getSatur :: Graph -> GraphColoring -> Int -> Int
getSatur graph coloring node =
    let neighbours = Map.findWithDefault [] node graph
        neighbourNonZeroLabels = (S.size . S.fromList . filter
        ↪  (/= 0)) [Map.findWithDefault 0 n coloring | n <-
        ↪  neighbours]
    in neighbourNonZeroLabels

saturs :: Graph -> GraphColoring -> Map.Map Int Int
saturs graph coloring =
    let unassigned = unassignedNodes coloring
        allSaturs = Map.fromList ([(v, getSatur graph coloring
        ↪  v) | v <- unassigned])
    in allSaturs

maxDSATUR :: Graph -> GraphColoring -> Int
maxDSATUR graph coloring =
    let allSaturs = saturs graph coloring
        nextNode = fst $ Map.foldrWithKey (\k v (currK, currV)
        ↪  -> if v > currV then (k, v) else (currK, currV))
        ↪  (-1, -1) allSaturs
    in nextNode

runDSATUR :: Graph -> Int -> GraphColoring -> Int -> Int ->
↪  Maybe (GraphColoring)
runDSATUR graph k col v c
    | v == -1 = returnMaybeColoring graph col k
    | checkColoring graph col k == False = Nothing
    | otherwise = listToMaybe (catMaybes [ runDSATUR graph k
    ↪  nextColoring (maxDSATUR graph nextColoring) c_ | c_ <-
    ↪  [1..k] ])
```

```haskell
    where nextColoring = (Map.insert v c col)

dSATURAlgorithm :: Graph -> Int -> Int -> Maybe (GraphColoring)
dSATURAlgorithm graph k n = listToMaybe (catMaybes [ runDSATUR graph k (initColoring n) 0 c | c <- [1..k] ]) -- at the start it is fine to start with 0 as all nodes have 0 saturation

-- Parallel DSATUR

parallelSaturs :: Graph -> GraphColoring -> Map.Map Int Int
parallelSaturs graph coloring =
    let unassigned = unassignedNodes coloring
        allSaturs = Map.fromList ([(v, getSatur graph coloring v) | v <- unassigned] `using` parList rdeepseq)
    in allSaturs


parallelMaxDSATUR :: Graph -> GraphColoring -> Int
parallelMaxDSATUR graph coloring =
    let allSaturs = parallelSaturs graph coloring
        nextNode = fst $ Map.foldrWithKey (\k v (currK, currV) -> if v > currV then (k, v) else (currK, currV)) (-1, -1) allSaturs
    in nextNode

runParallelDSATUR :: Graph -> Int -> GraphColoring -> Int -> Int -> Int -> Maybe (GraphColoring)
runParallelDSATUR graph k col v c d
    | v == -1 = returnMaybeColoring graph col k
    | checkColoring graph col k == False = Nothing
    | d < 4 = listToMaybe (catMaybes ([ runParallelDSATUR graph k nextColoring (parallelMaxDSATUR graph nextColoring) c_ (d + 1) | c_ <- [1..k] ] `using` parList rdeepseq))
    | otherwise = listToMaybe (catMaybes [ runParallelDSATUR graph k nextColoring (parallelMaxDSATUR graph nextColoring) c_ (d + 1) | c_ <- [1..k] ])
    where nextColoring = (Map.insert v c col)

parallelDSATURAlgorithm :: Graph -> Int -> Int -> Maybe (GraphColoring)
parallelDSATURAlgorithm graph k n = listToMaybe (catMaybes ([ runParallelDSATUR graph k (initColoring n) 0 c 1 | c <- [1..k] ] `using` parList rdeepseq)) -- at the start it is fine to start with 0 as all nodes have 0 saturation

```

```haskell

-- Parallel Pruning

runParallelPruning :: Graph -> Int -> GraphColoring -> Int -> Int -> Int -> Maybe (GraphColoring)
runParallelPruning graph k col v c d
    | v == Map.size graph = returnMaybeColoring graph col k
    | checkColoring graph col k == False = Nothing
    | d < 4 = listToMaybe (catMaybes ([ runParallelPruning graph k (Map.insert v c col) (v + 1) c_ (d + 1) | c_ <- [1..k] ] `using` parList rdeepseq))
    | otherwise = listToMaybe (catMaybes [ runParallelPruning graph k (Map.insert v c col) (v + 1) c_ (d + 1) | c_ <- [1..k] ])

parallelPruningAlgorithm :: Graph -> Int -> Int -> Maybe (GraphColoring)
parallelPruningAlgorithm graph k n = listToMaybe (catMaybes ([ runParallelPruning graph k (initColoring n) 0 c 1 | c <- [1..k] ] `using` parList rdeepseq))



-- runner

runAlgorithm :: Graph -> Int -> String -> Int -> Maybe (GraphColoring)
runAlgorithm graph k algo n
    | algo == "parallel_DSATUR" = parallelDSATURAlgorithm graph k n
    | algo == "parallel_pruning" = parallelPruningAlgorithm graph k n
    | algo == "DSATUR" = dSATURAlgorithm graph k n
    | algo == "pruning" = pruningAlgorithm graph k n
    | algo == "brute_force" = bruteForceAlgorithm graph k n
    | otherwise = error ("Unknown algorithm: " ++ algo)


runSolver :: String -> String -> String -> IO ()
runSolver fileName kStr algo = do
    inputData <- readFile fileName
    let graph = loadGraph inputData
        k = read kStr :: Int
        n = Map.size graph
        coloring = runAlgorithm graph k algo n
    --print graph
```

```
168     print k
169     print coloring
170
171     --putStrLn inputData
```