

**MAT206**

Graph Theory and its Applications

**Image-Based Iterative and State Saving Graph Coloring**

**Group Members:-**

DEEPANJAN MUKHERJEE 14BCE1039

HARSHAL KAVATHIA 14BCE1029

Graph Coloring

In graph theory, graph coloring is a special case of graph labeling; it is an assignment of labels traditionally called "colors" to elements of a graph subject to certain constraints. In its simplest form, it is a way of coloring the vertices of a graph such that no two adjacent vertices share the same color; this is called a vertex coloring.

The convention of using colors originates from coloring the countries of a map, where each face is literally colored. This was generalized to coloring the faces of a graph embedded in the plane. By planar duality it became coloring the vertices, and in this form it generalizes to all graphs. In mathematical and computer representations, it is typical to use the first few positive or nonnegative integers as the "colors". In general, one can use any finite set as the "color set". The nature of the coloring problem depends on the number of colors but not on what they are.

The graph coloring problem deals with assigning each vertex of the graph with a color such that the adjacent vertices have distinct colors. Many applications have been shown to be reducible to the graph coloring problem. Some of these include 1) register allocation in the back-end of a complier, and 2) scheduling of multiple tasks to resources under various constraints. Graph coloring is computationally hard. It is NP-complete to determine if a given graph admits k-coloring (except for k=1 or k=2).

State space search

State space search is a process used in the field of computer science, including artificial intelligence (AI), in which successive configurations or *states* of an instance are considered, with the goal of finding a *goal state* with a desired property. Problems are often modelled as a state space, a set of *states* that a problem can be in. The set of states forms a graph where two states are connected if there is an *operation* that can be performed to transform the first state into the second.

State space search is a technique where successive configurations (a.k.a states) are explored until the state with desired property is found. The different states form a tree where child nodes are produced from the parent node by changing some part of the parent state. In typical problems, exploring the entire state space graph is impractical due to execution time and memory constraints. Application of heuristics has been shown to be effective in pruning the search space.

Difference from normal Graph Coloring

Large-scale systems with distributed memory are a natural fit to solving the computationally complex graph coloring problem. There are two main parallelization strategies, namely domain decomposition and state space exploration. In domain decomposition, the graph is divided into sub-graphs, and each of these is assigned to a processor. The sub-graph is colored locally at each processor. The algorithm operates in time-steps, where at the end of each time-step, the processors exchange coloring information with each other to resolve the conflicts for the vertices on the sub-graph boundaries. State space exploration takes a different approach to dividing work between processors. A state in the state space tree is a partially colored input graph, and child states are produced from parent states by coloring one of the uncolored vertices. A solution is found if one of the leaves if a legitimately colored input graph. Each processor is responsible for exploration of a part of the state space tree.

Heuristics for State-Space Search

Pre-Coloring

The first of these heuristics is a pre-processing technique, namely the pre-coloring of the graph. We find the 3-vertex cliques in the graph, and assign them the lowest available color ensuring that a vertex does not get any of the colors assigned to its neighbours. However, we found that this technique is best applied to sparse graphs only. As dense graphs tend to have many, many 3-vertex cliques, this pre-processing technique often results in an unoptimized sequential coloring of a large part of the input graph. This may result in un-colourability of the graph even though it may be colourable with the given number of colors by using an optimized approach.

Vertex Removal

By noting that a vertex with degree less than the number of available colors must always have a valid possible coloring, we can remove such a vertex from the graph (pushing it onto a local stack), color the rest of the graph, and re-include the removed vertex, assigning it the lowest available color. It is important to note that this removal can be performed recursively. That is, the removal of a vertex from the graph may lower the degree of its neighbouring vertices to the point that this heuristic could then be applied to them.

Next Vertex Selection

An intuitive and simple heuristic guides our process for selecting the next vertex to be colored. By picking the vertex with the least available number of potential colors, we can move more efficiently through the state-space. Two related optimizations include impossibility-testing and forced-move. In the former, if after a vertex is colored, one of its neighbours has no available colors left, then the state for that coloring is not generated since it can never lead to a valid coloring of the entire graph. The latter dictates that after a vertex is colored, if the number of possible colors at any neighbour is reduced to 1, then the neighbour is colored in the same step. This process is repeated recursively since the coloring of a neighbour might lead to forced-move for the neighbour of the neighbour.

Detection of Independent Sub graphs

This heuristic is to detect independent sub graphs, an, O(V + E) operation, in the input graph and then to run the coloring algorithm on each sub graph separately. By splitting the graph into two or more disconnected pieces, we often obtain large speedups as the problem is exponential in nature and evaluating multiple smaller sub graphs results in extremely fewer possible states as compared to evaluating a single, larger graph. This heuristic also leads to a changing of the structure of the state-space tree in that it transforms it into an AND-OR tree. In the single graph approach, either the left OR right sub tree (or at least one of many) has to find a solution to the coloring problem for an overall solution to be found, and failure in one sub tree does not necessitate the failure of the entire search. However, in the case of an AND-OR tree, when splitting the problem into multiple, independent sub graphs it becomes necessary for a solution to exist for all of the sub graphs (the left AND right sub trees have to yield solutions), and failure for any one sub graph implies failure for the entire search. Additionally, higher priority is given to that sub tree of an AND node which has less uncolored vertices, with the hope that it would finish faster.

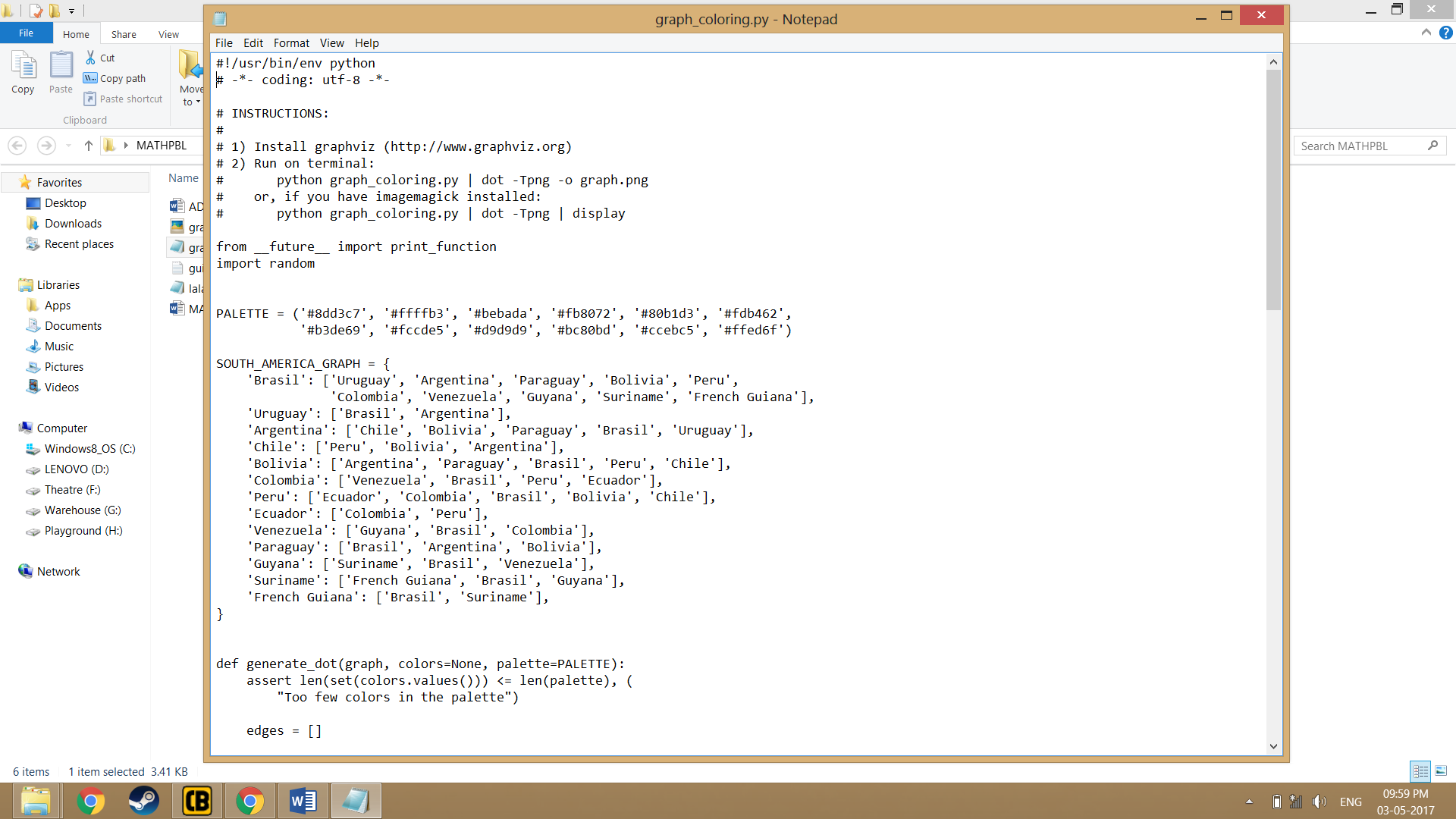
Basic Explanation of Working:

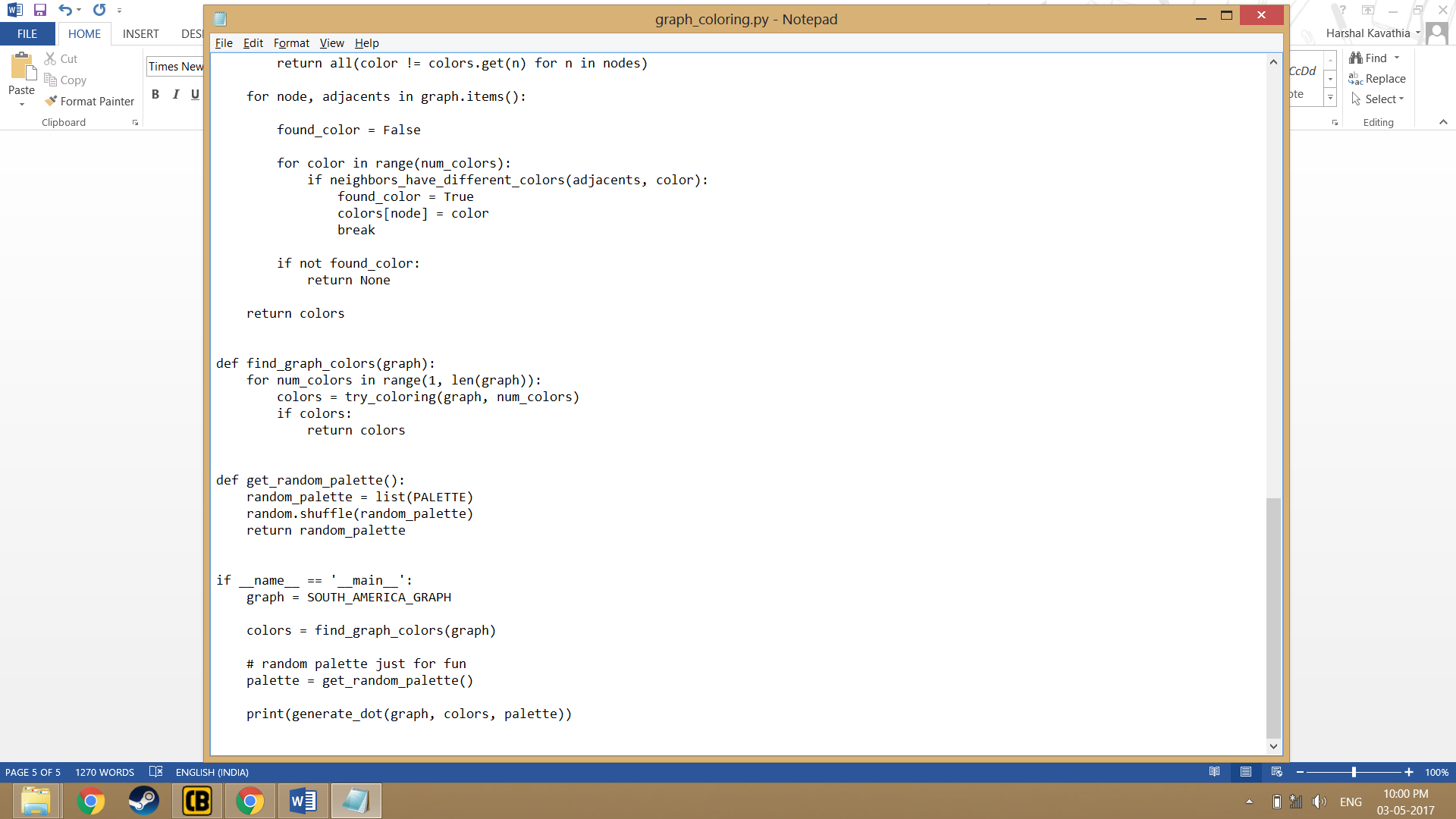
The initial step of reading the graph data is the most important. In this step, we perform vertex coloring on the graph given and create a PNG image on the basis of a greedy calculation of one possible graph coloring state.

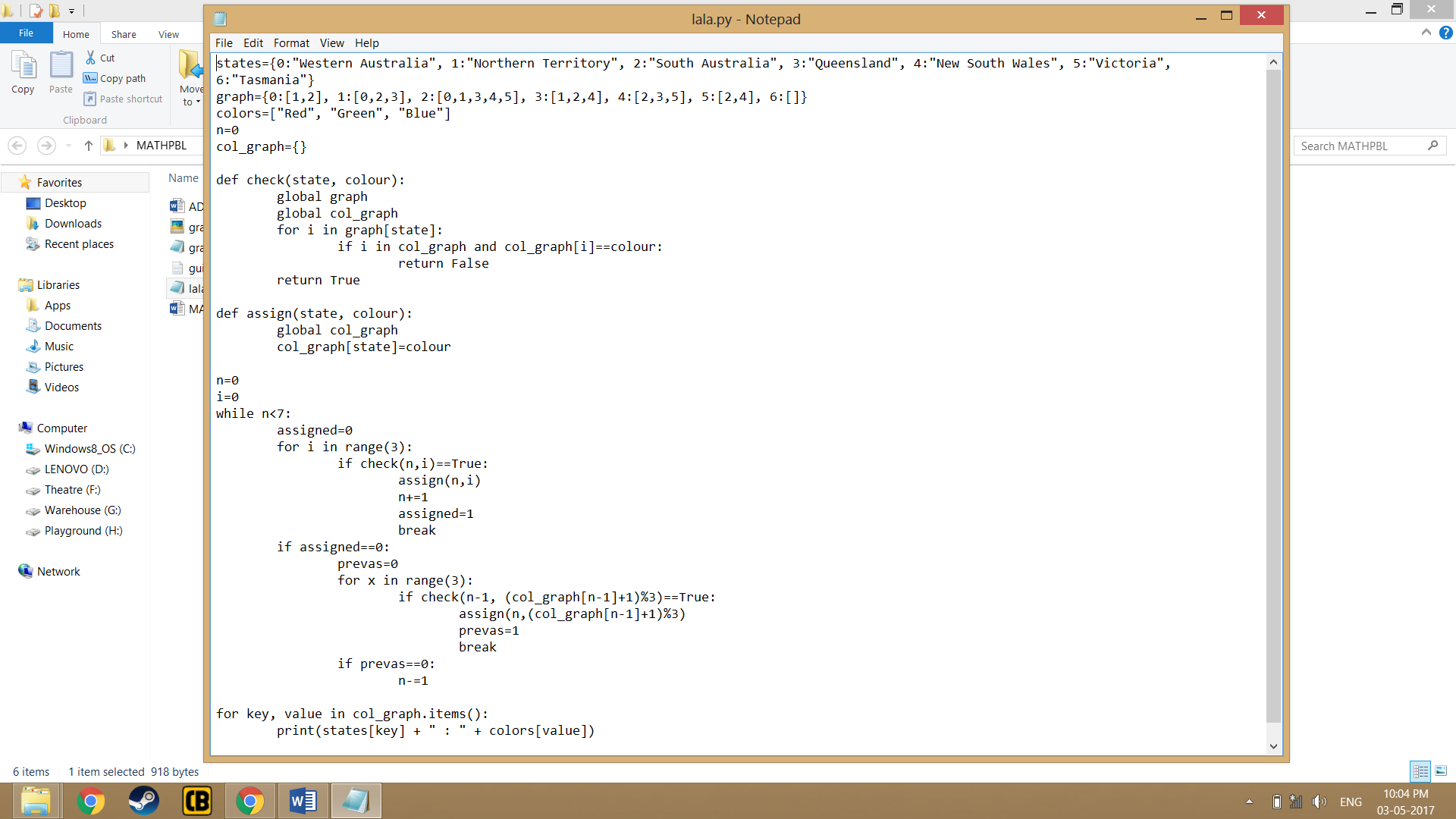
After that we perform state creation. Each step analyses the previous result via Dynamic Programming and decides upon a possible incremental improvement to the same.

This incremental improvement is then applied to the PNG image and that image is replaced by a new PNG image which is utilized for the next iteration.

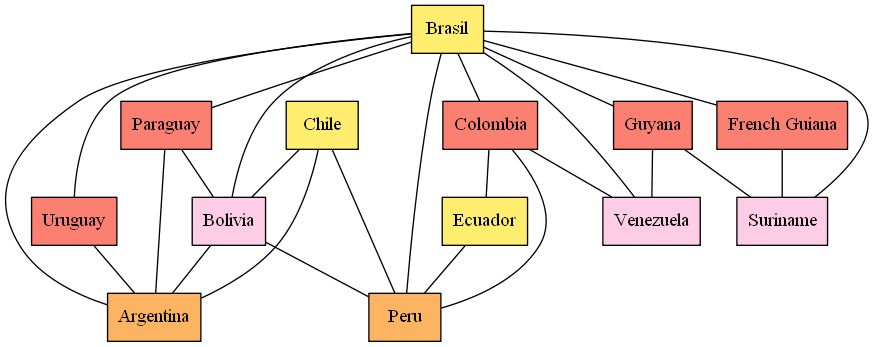
Code Snippets







Output



References

* Graph Coloring Using State Space Search by University of Illinois, Urbana-Champaign.
* Stuart J. Russell and Peter Norvig. Artificial Intelligence: A Modern Approach.
* Python Library 2.31.
* M. Kubale, History of graph coloring.
* Wikipedia.