Programming Assignment 3

CMPE 250, Data Structures and Algorithms, Fall 2015

Instructor: A. T. Cemgil
TA's: Çağatay Yıldız, Atakan Arıkan, Barış Kaya

Due: December 11, 2015, 23:55 Sharp

Four Color Theorem

In mathematics, four color theorem states that no more than four colors are required to color the regions of a map so that no two adjacent regions have the same color. Map coloring problem can be studied on graphs if each vertex represents a region and an undirected arc between two vertices implies that regions corresponding to these vertices are adjacent. If one can assign colors to the vertices in such a way that no two vertices connected with a single arc are of the same color, then this set of assignment solves the problem of map coloring. For an illustrative example, see the example slides from Russell and Norvig's book *Artificial Intelligence: A Modern Approach* at the end of this document.

Goal

Graph coloring problem is an NP-hard problem. But there are good algorithms to the solution. One of them is backtracking search (which finds a solution if there exists one but runs in exponential time) and the other one is Welsh-Powell algorithm (which uses a heuristic and does not guarantee to find the optimal coloring). You can see the ideas of these at the end of the document.
In this project, you will implement *an* algorithm to color vertices. You may choose one of those noted above or you can implement your own algorithm.

Input/Output File Format

Input to your program is simply the graph. The structure of the input file:

1. First line contains the number of vertices in the graph. Say it is N.
2. The adjacency matrix, made up of N rows and N columns.

To color the whole map, you are allowed to use only 4 colors: `red, green, blue, orange`. So, if the input graph is **not** 4-colorable, generate an output file with a single word: `ups`. Otherwise, output file contains the color assignments to vertices, starting from 0,1,2,... This file should have N lines if the number of vertices is N.

Sample Input File	Sample Output File
6	red
0 1 1 0 1 0	blue
1 0 0 0 1 0	blue
1 0 0 1 1 0	red
0 0 1 0 1 0	green
1 1 1 1 0 0	red
0 0 0 0 0 0	

Table 1: Sample Input and Output files

Implementation Details

1. As always, your program will be compiled with `make` command. Therefore, if you add new files, you have to update `Makefile` accordingly so that your code can be auto-compiled. Note that it is fine to code in a single file in this project (Because I am assuming you already know what `.h` and `.cpp` files are and how to use those)

2. I will execute your program with `./project3 inputFile outputFile` command. So, use the command line arguments to your main function accordingly.

3. I have given two example graphs and their colorings. Note that coloring is not fixed. So, the output of your program may be different than that of mine, no problem. My suggestion is that write an additional function that reads a graph and its coloring and checks whether the coloring is possible or not.

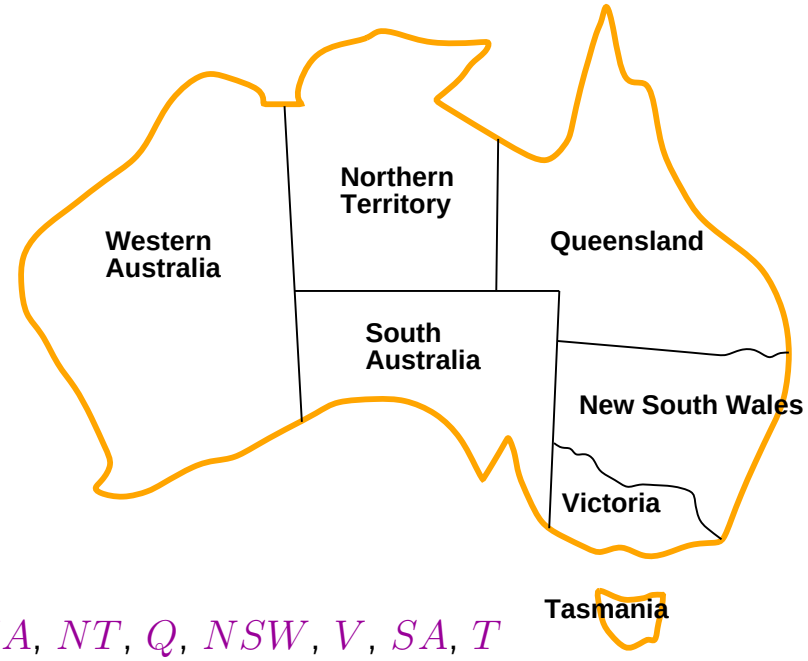
4. You are allowed to use only 4 colors: `red, green, blue, orange`.

5. You may safely assume that there is no arc from a vertex to itself.

Project Guidelines

- **Warning:** All source codes are checked automatically for similarity with other submissions and also the submissions from previous years. Make sure you write and submit your own code.
- Your program will be graded based the correctness of your output and the clarity of the source code. Correctness of your output will be tested automatically so make sure you stick with the format described above.
- There are several issues that makes a code piece 'quality'. In our case, you are expected to use C++ as powerful, steady and flexible as possible. Use mechanisms that affects these issues positively.
- Make sure you document your code with necessary inline comments, and use meaningful variable names. Do not over-comment, or make your variable names unnecessarily long.
- Try to write as efficient (both in terms of space and time) as possible. Informally speaking, try to make sure that your program completes in meaningful amount of time.

Example: Map-Coloring



Variables WA, NT, Q, NSW, V, SA, T

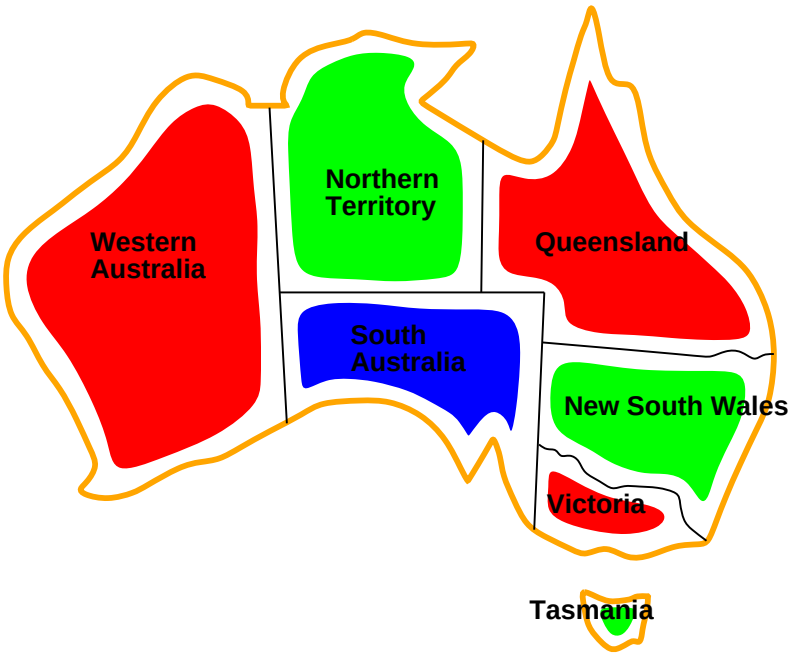
Domains $D_i = \{red, green, blue\}$

Constraints: adjacent regions must have different colors

e.g., $WA \neq NT$ (if the language allows this), or

$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \ldots\}$

Example: Map-Coloring contd.

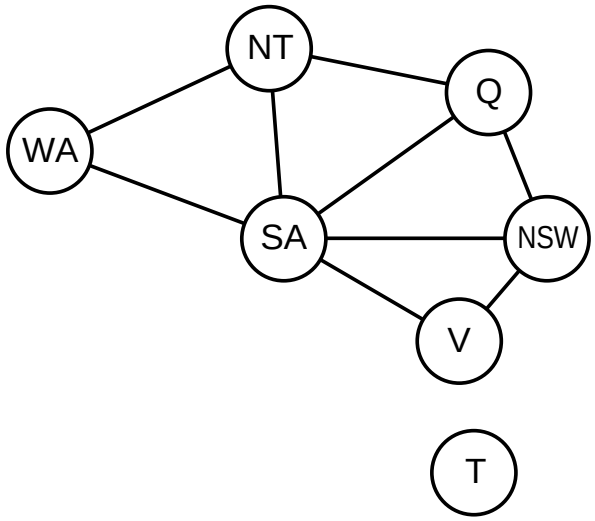


Solutions are assignments satisfying all constraints, e.g.,
$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green\}$

Constraint graph

Binary CSP: each constraint relates at most two variables

Constraint graph: nodes are variables, arcs show constraints



General-purpose CSP algorithms use the graph structure
to speed up search. E.g., Tasmania is an independent subproblem!

Backtracking search

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

VERTEX COLORING: WELSH-POWELL ALGORITHM

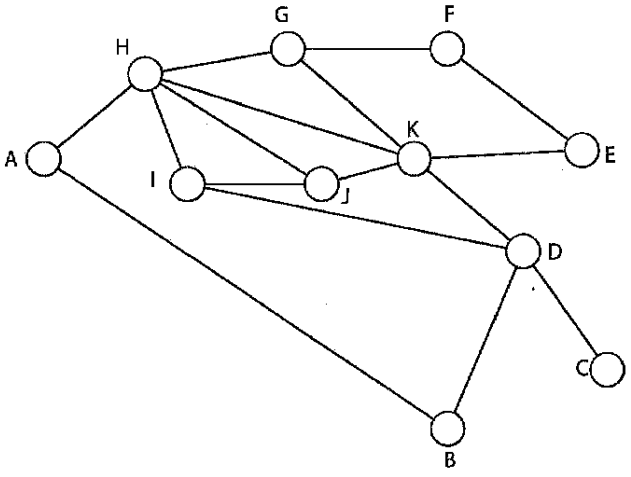
One algorithm that gives a good solution to a vertex-coloring problem is the *Welsh-Powell algorithm*. It may not always give the best solution, but it will usually perform better than just coloring the vertices without a plan will.

The Welsh-Powell algorithm consists of the following steps:

1. Find the valence for each vertex.
2. List the vertices in order of descending valence (you can break ties any way you wish).
3. Color the first vertex in the list (the vertex with the highest valence) with color 1.
4. Go down the list and color every vertex not connected to the colored vertices above the same color. Then cross out all colored vertices in the list.
5. Repeat the process on the uncolored vertices with a new color – always working in descending order of valence until all the vertices have been colored.

The idea is that, by starting with the vertices that have the highest valences, you will be able to take care of the vertices with the largest number of conflicts as early as possible.

EXAMPLE:



Vertex	Valence
A	2
B	2
C	1
D	4
E	2
F	2
G	3
H	5
I	3
J	3
K	5

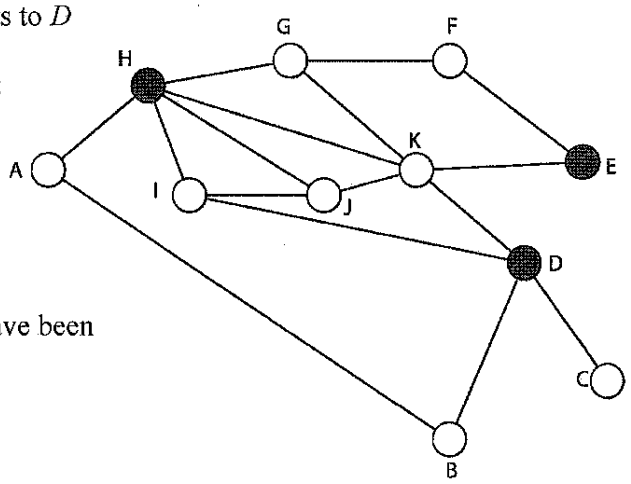
Arrange the list of vertices in descending order of valence. If there's a tie, you can randomly choose how to break it. (To get the ordering below, we used alphabetical order to break ties.) The new order will be:

H, K, D, G, I, J, A, B, E, F, C

Now we color the vertices, in the order listed above. Here's the thought process for the first color:

H color red
K don't color red since it connects to *H*
D color red
G don't color red since it connects to *H*
I don't color red since it connects to *H*
J don't color red since it connects to *H*
A don't color red since it connects to *H*
B don't color red since it connects to *D*
E color red
F don't color red since it connects to *E*
C don't color red since it connects to *D*

Now the graph should look like this:



If we now ignore the vertices that have been already colored, we're left with:

K, G, I, J, A, B, F, C

We can repeat the process now with a second color (blue):

K color blue
G don't color blue since it connects with *K*
I color blue
J don't color blue since it connects with *I*
A color blue
B don't color blue since it connects with *A*
F color blue
C color blue

Again, cross out the colored vertices, leaving you with *G, J, B*, and start at the top of the new list with a new color (yellow):

G color yellow
J color yellow
B color yellow