Matthew Parke
Professor Sitaraman
CmpSci 311: Algorithms
12/1/15

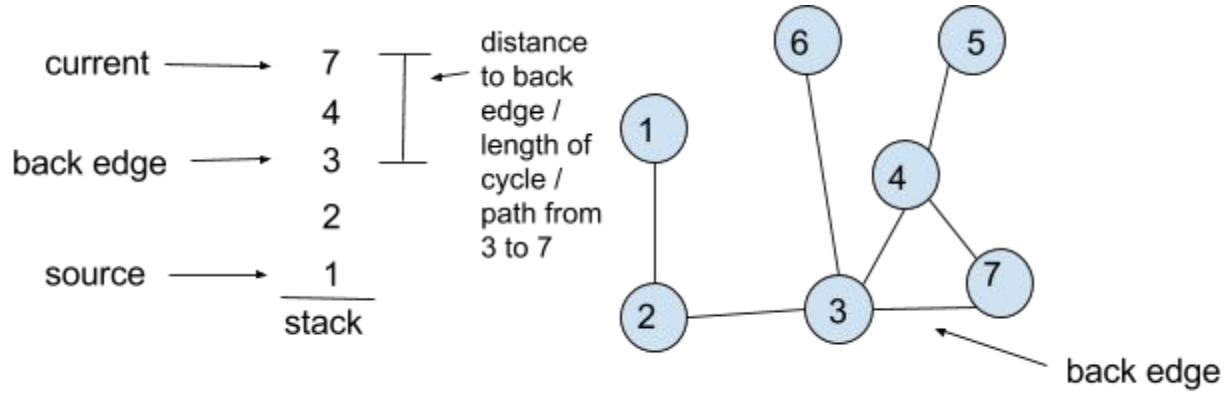
Description:

Creating the graph: To create the graph the program is passed a file through the args array. The number of vertices is read from the top and an empty graph is created based on that number. Integer by integer, pairs of vertices are read in and have their edge lists updated to include their pair. If the vertex does not exist in the graph, it is created, updated with the proper edge list, and then placed in the graph. When there is nothing left in the file to read the graph will be complete.

Two Colorable Algorithm: Once the graph is populated, the method isTwoColorable() is run on the graph. The method looks for an odd length cycle and returns the cycle if one is found, otherwise it returns the two-colored version of the graph. The method starts with a foreach loop that goes over the total list of vertices held in a graph object. It then checks each vertex to see if it is colored white, a white coloring indicates that the vertex has never been visited by the algorithm. It is important to check if all vertices have been visited because if the graph has multiple disjoint parts then a Depth first search (DFS) of just one node will not reach all parts of the graph. Once a white vertex is chosen, then DFS is performed using the white vertex as the source vertex. My implementation of DFS uses white, grey, and black as visitation statuses for vertices; white standing for unvisited, grey for in the process of visitation, and black for fully visited. A stack is kept of all grey vertices, meaning that they are all in the process of being visited. The action taken as each vertex is based on the visitation-color of the vertex's children and the color of the parent (red or green). If there is a grey child, that is not the parent, then a back edge is found and the stack is examined to determine the length of the cycle (correctness description will say why). If the length is odd then the cycle from the current vertex to the back edge is returned as a graph and printed to a file. If there is no odd cycle detected then the program will examine the next white vertex, color it opposite of its parent (red or green) and color its visitation-color grey, place it on the stack, and travel to that vertex. If there is no cycle and no children to go to then the node is visitation-colored black and popped off the stack, the new head of the stack (the parent) is now the current node. On return of the method either a fully two-colored graph will be returned or a trace of the cycle that invalidates the graph.

Writing out to file: Simple PrintWriter object writes vertex by vertex to "graphoutput.txt". If the vertices list is null in the graph object then the graph represents a cycle and the PrintWriter writes the vertices in the cycle array to the file, else, the graph is two colored and it writes all vertices in the vertices array to the file.

Correctness:
My algorithm produces the correct result because I make sure to check all vertices for being part of an odd cycle by checking if each vertex has a back edge to a non parent. If a vertex as a back edge to a nonparent vertex then a cycle is found, however, even cycles are accepted. I can accurately determine the length of the cycle by seeing how many vertices I travel through using depth first search (DFS) to get to the current vertex from the vertex connected by the back edge. In DFS the stack hold the search path from the source to the current vertex, so if we look at two vertices on the stack then everything in between them is a valid path to get from one vertex to the other. We can use this to check the length of the cycle. Take this picture for example:



This structuring makes it easy to produce the correct cycle because the path of the cycle is all the vertices between the two vertices connected by the back edge. My algorithm produces the correct colored graph when no cycle is detected because as each vertex is examined it is colored opposite of its parent vertex. When the algorithm has examined all vertices then all vertices should have an opposite color to their parent, making the graph a valid two color graph.

Runtime:
I used the unix shell command ‘time’ to time my program on the edlab server, the results are as follows.
Smallgraph: real 0m 0.139s
user 0m 0.124s
sys 0m 0.024s

Largegraph1: real: 0m 10.618s
user: 0m 11.605s
sys: 0m 0.332s

Largegraph2: real: 0m 4.719s
user: 0m 5.896s
sys: 0m 0.304s

Runtime Analysis:
Psuedocode for algorithm:

```
foreach vertex v in V                          ---------- Θ(|V|) at very worst, but then edges
    if V is white                                             wouldn't exist
        push v onto stack/color grey
        while stack is not empty               ---------- Θ(|V|) at very worst → all one graph
            v = peek
            v.color is opposite of parent
            check if cycle                     --------- |
                output cycle                             |---------- Θ(|E|) if else statements
            check if white child exists        --------- |
                push child onto stack
            else
                pop
                paint black
```

The runtime for my program is a little complex because of the outer foreach loop of the algorithm. At the very worst no vertices are connected and we run DFS on each vertex, but examine no edges. This would make runtime to be Θ(|V|) because the DFS would be constant time, and the changing vertex color to grey would be constant. If we have an entirely connected graph, the forloop will only execute one DFS, which runs in Θ(|V| + |E|) time and each iteration after that would be constant, making the run time Θ(|V| + |E| + c), if with c standing for the time for the remaining forloop execution. If the number of disjoint sections to the graph is the value c then the runtime would be Θ(c(|V/c| + |E/c|)), moreover, Θ(c|V/c|+c|E/c|) which is Θ(|V|+|E|).