DMA 11g

Carl Dybdahl, Patrick Hartvigsen, Emil Chr. Søderblom

January 9, 2017

1.

We have been asked to write pseudocode that implements depth-first search using a stack instead of recursion. This is our implementation:

dfs.txt

```
1  DFS(G)
2          for each vertex u in G.V
3                  u.Color = white
4                  u.Pi = Nil
5          clear (S)
6          for each vertex u in G.V
7                  push (S,(u, Nil))
8                  while not empty(S) do
9                          u, v = pop (S)
10                         DFS-visit(u, v)
11
12 DFS-visit(u, p)
13         if u.Color = white
14                 u.Color = grey
15                 u.Pi = p
16                 for each v in adj(u)
17                         push(S,(v, u))
18                 u.Color = black
```

2.

By using a stack, our algorithm avoids recursion. Each element of the stack consists of a pair, `(u, v)`, where `u` is a visited node and `v` is its potential parent. Otherwise, our algorithm is somewhat similar to that in CLRS.

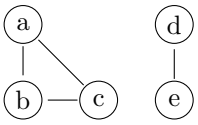
We initially color every vertex white and set its parent to nil.

The algorithm searches through each node of the graph, pushing them and a nil parent to the stack. In each iteration of this search, it processes the stack until it is empty.

An iteration of processing the stack consists of popping a pair `(u, v)`, and, if the vertex `u` is white, visiting it. The visit consists of coloring it grey, adding all its neighbours `w` to the stack, paired together with `u` as `(w, u)`, and then finally coloring it black.

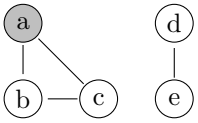
3.

We have chosen the following graph for the example:



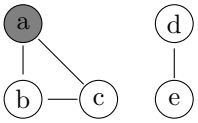
The algorithm loops through each vertex of the graph, checking whether they're white. The first such element is (a), which it adds to the stack.

The stack now contains `(a, nil)`. This means that it is not empty, and so the algorithm enters the while-loop, where this element is popped off the stack. (a) is colored grey and its neighbours are visited, the first being (b), which is added to the stack paired with (a):

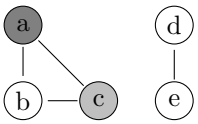


Now the stack contains `(b, a)`. However, there are still more neigbours of (a) to visit. (c) is also visited and added to the stack:

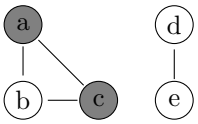
The stack now contains `(b, a)` and `(c, a)`, with `(c, a)` on top. Now that each neighbour has been visited, (a) is colored black and the next iteration of the while-loop begins.



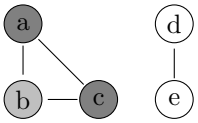
(c) is popped off the stack, so that it now contains `(b, a)`. (c) is colored grey, and its parent is set to (a). (c)'s neighbours (a) and (b) are added to the stack.



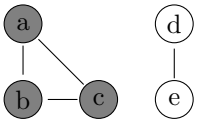
The stack now contains `(b, a)`, `(a, c)` and `(b, c)`. (c) is colored black, since it has been visited.



The top of the stack, `(b, c)`, is popped, and since (b) is still white its parent is set to (c) and it is colored gray. Each of its neighbours is added to the stack.



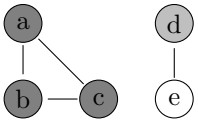
The stack now contains `(b, a)`, `(a, c)`, `(a, b)` and `(c, b)`. The visiting procedure of (b) ends with (b) being colored black.



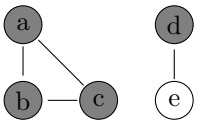
`(c, b)` is popped from the stack, but since (c) isn't white it is not visited. The same happens to the rest of the stack, and the processing loop ends as the stack becomes empty.

The outer loop keeps searching and pushes `(b, nil)` to the stack, however since again (b) isn't white, it is not visited and the stack becomes empty. The same happens for (c).

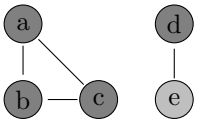
Finally, the outer loop finds (d) and pushes `(d, nil)` to the stack. It gets popped and colored gray, and its neighbour (e) is added to the stack.



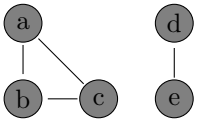
After the neighbour has been pushed, (d) is colored black.



(e) is popped from the stack, colored gray, and its parent is set to (d). Its neighbour (d) is pushed.



Next, (e) is colored black.



Since (d) isn't white, it doesn't get visited, and the algorithm terminates.