Homework 8

Due by 4:30pm on November 4, 2020

Viewings and Readings

[Links to the slides, recordings, and quiz are on the course web site.]
Review the week 8 lecture slides.
View the week 8 lecture recordings and the discussion session recording.
Complete the week 8 quiz.
Read chapter 5 and sections 6.1-3 of the textbook (section 6.4 is optional).

Problems

Please write precise and concise answers. Except where explicitly indicated, your algorithm descriptions should use either clear, concise, and precise plain English or clear, concise, and precise pseudo-code that uses a style similar to the pseudo-code in your textbook. Submit your solutions to problems **1, 2(a), and 3(a)** via D2L as a Word or PDF file or as scans/photos of legible handwritten notes. Submit your solutions to problems **2(b)** and **3(b)** via Kattis.

1. Problem 10(a)-(c) page 210. This problem is meant to illuminate the workings of Depth-First Search.

One of the earliest published descriptions of whatever-first search as a generic class of algorithms was by Edsger Dijkstra, Leslie Lamport, Alain Martin, Carel Scholten, and Elisabeth Steffens in 1975, as part of the design of an automatic garbage collector. Instead of maintaining marked and unmarked vertices, their algorithm maintains a color for each vertex, which is either white, gray, or black. As usual, in the following algorithm, we imagine a fixed underlying graph G.

```
ThreeColorSearch(s):
  color all nodes white
  color s gray
  while at least one vertex is gray
    ThreeColorStep( )


ThreeColorStep( ):
  v <- any gray vertex
  if v has no white neighbors
    color v black
  else
    w <- any white neighbor of v
    parent(w) <- v
    color w gray
```

(a) Prove that ThreeColorSearch maintains the following invariant at all times: No black vertex is a neighbor of a white vertex. [Hint: This should be easy.]

```
Vertexes are only colored white at the start of the problem.
Vertexes are only colored black if they have no white neighbors

This means a brand new black vertex has no white neighbors
And since we never color a vertex white after initialization there is never an opportunity for a white to become that black vertexes neighbor
```

(b) Prove that after ThreeColorSearch(s) terminates, all vertices reachable from s are black, all vertices not reachable from s are white, and that the parent edges v□parent(v) define a rooted spanning tree of the component containing s. [Hint: Intuitively, black nodes are â€œmarkedâ€□ and gray nodes are â€œin the bagâ€□. Unlike our formulation of WhateverFirstSearch, however, the three-color algorithm is not required to process all edges out of a node at the same time.]

```
The algorithm will get a white neighbor of a gray vertiex and make it gray
So the algorithm will reach all vertixes reachable by the initial vertex since neighbors
Since it grabs any gray vertex we can assume theres a scenario where if theres a vertex with a white neighor, we select that vertex and make the neig
This will make all reachable vertexes of s gray, then if we keep running the algorithm, all those gray vertexes have no white neighbors so they all b

All vertexes not reachable never become gray and thus never become black, they stay white

This makes a spanning tree since each node will only have one parent
This is because the same time a vetex gets a parent when turning from white to gray which can only happen once
```

(c) Prove that the following variant of ThreeColorSearch, which maintains the set of gray vertices in a standard stack, is equivalent to depth-first search. [Hint: The order of the last two lines of ThreeColorStackStep matters!]

```
ThreeColorSearch(s):
  color all nodes white
  color s gray
  push s onto the stack
  while at least one vertex is gray
    ThreeColorStep( )


ThreeColorStep( ):
  pop v from the stack
  v <- any gray vertex
  if v has no white neighbors
    color v black
  else
    w <- any white neighbor of v
    parent(w) <- v
    color w gray
    push v onto the stack
    push w onto the stack
```

```
Since the push actions first push the parent then the child this is DFS
The "pop v from stack" action will always pop the vertex that just became gray, which is the most recent child vertex.

It will keep pushing and popping the child until the child has no white neighbors
Then after that reaches the end of the tree it will recurse back up and start popping the parents of the child and getting other children of that par

We can see it will go as deep down the tree as it can before coming back up to try other nodes of the tree
```

2. Week 8 problem *runningmom* on Kattis.

(a) Explain how a DFS traversal of the graph modeling the problem and computing the `_.pre` and `_.post` values at every vertex can be used to solve this problem.

```
In my solution I kept track of whether the connection flight was on the stack or not.

If the traversal of the graph brought us back to an already visited destination, that doesnt always mean there's a loop.  There could be two disjoi

We need to check whether the destination has been visted before AND if that destination is currently on the stack.
If its still on the stack, that means we havent recursed back to that destination. We are still traversing with that destination as an ancestor whicl
```

(b) Use your insight from **(a)** to implement a solution using your preferred language and submit your implementation via Kattis.

3. Week 8 problem *torn2pieces* on Kattis.

(a) Explain how a DFS traversal of the graph modeling the problem and computing the DFS traversal tree rooted at the starting tree can be used to solve this problem.

```
Traverising the graph in this problem works because we can recurse through all potential routes to the destination.  We don't need to revisit statior
Traversing lets us compute whether the destination is reachable and then all we need is some extra logic to store the path taken and we get the solut
```

(b) Use your insight from (b) to implement a solution using your preferred language and submit your implementation via Kattis.