Graded Assignment 4 – DSA

Claudio Maggioni

May 27, 2019

Contents

Listings

1 Exercise 1

```
FUNCTION BEST-PATH(G=(V,E), v, w):
  P[V(G)[0]] = NIL

  for each vertex u ∈ V(G):
    prev_start[u] = NIL
    prev_end[u] = NIL

  prev_start[v] = START (non-NIL)
  prev_end[w] = END (non-NIL)

  HELP-SETUP(G, P, Adj[V(G)[0]], V(G)[0])

  s = v
  e = w

  while prev_end[s] is NIL and prev_start[e] is NIL:
    if P[s] is not NIL:
      prev_start[P[s]] = s
      s = P[s]
    if P[e] is not NIL:
      prev_end[P[e]] = e
      e = P[e]

  if prev_end[s] is not NIL:
    n = s
  else:
    n = e

  while s is not v:
    s = prev_start[s]
    prev_end[s] = P[s]

  s = v
  while s != w:
    print(s)
    s = prev_end[s]
  if v != w:
    print(w)

FUNCTION HELP-SETUP(G=(V,E), P, S, v):
  for each vertex u ∈ S:
    P[u] = v
```

```
43      HELP-SETUP(G, P, Adj[u] \ {v}, u)
```

Listing 1: Solution for exercise 1

The $O(n)$ setup happens between line 2 and line 14. This is mainly needed to initialize some help arrays and define an arbitrary root (and consequent parent relation) on the tree.
The rest of the algorithm walks the tree from the start to the root and from the end to the root concurrently, keeping track of the path taken and stopping when an edge was traversed by both walks. Then, the path memory to the start is reversed and inserted in the path memory for the end in order to obtain a mapping to the next node in the path from v to w. This mapping is then printed. The complexity of this step is $O(dist(v, w))$, since the number of traversed edges is at most two times the distance from v to w, and the reversing operation at the end requires at most $dist(v, w)$ steps, as the printing operation.

2 Exercise 2

```
FUNCTION CONNECTED-COMPONENTS(G=(V,E)):
  for each vertex u ∈ V(G):
    color[u] = WHITE

  c = 0

  for each vertex s ∈ V(G):
    if color[u] ≠ WHITE:
      continue

    color[s] = GRAY
    Q = ∅
    c = c + 1
    ENQUEUE(Q, s)

    while Q ≠ ∅:
      u = DEQUEUE(Q)
      for each v ∈ Adj[u]:
        if color[v] == WHITE:
          color[v] = GRAY
          ENQUEUE(Q, v)
        color[u] = BLACK

  return c
```

Listing 2: Solution for exercise 2

The algorithm is simply a modified color-only version of BFS with an extra iteration: using every vertex in the graph as a starting node. If the node was already visited, the color makes this iteration over all vertexes skip to the next vertex. The complexity of this algorithm is $O(|V| + |E|)$ like BFS, since the iterative application to BFS over every connected component will cover every edge and node of the graph exactly once, and the extra check for nodes being which is just another $O(|V|)$ cost, which can be ignored.

3 Exercise 3

Assume data is provided in Graph-like form $G = (V, E)$ where V is the set of butterflies, E is the set of edges, and a relation $o : E \to$ TRUE, FALSE where FALSE means "same" and TRUE means "different" to determine the the type of observation. Ambiguous observations are not included in $E(G)$ in the first place, so o does not have to be defined for this case.

```
FUNCTION OBSERVATION-HOLDS(G=(V,E), o):
  for each vertex u ∈ V(G):
    color[u] = WHITE
    species[u] = NIL

  for each vertex u ∈ V(G):
    if color[u] ≠ WHITE:
      continue
    species[u] = FALSE
    if not DFS-CHECK-OBSERVATION(species, o, v):
      return FALSE

  return TRUE

FUNCTION DFS-CHECK-OBSERVATION(species, o, v):
  color[u] = GREY

  for each vertex v ∈ Adj[u]:
    if color[v] == WHITE:
      species[v] = species[u] XOR o(edge (u, v))
      if not DFS-CHECK-OBSERVATION(species, o, v):
        return FALSE
    else:
```

```
    if species[u] $\neq$ species[v] XOR o(edge (v, u)) $\vee$
       species[v] $\neq$ species[u] XOR o(edge (u, v)):
       return FALSE

  color[u] = BLACK
  return TRUE
```

Listing 3: Solution for exercise 3

The code given traverses G using a modified DFS by first assigning an arbitrary species (`FALSE`) to the first vertex encountered, then by computing the species of the subsequent nodes encountered using the observation mapping $o : E \rightarrow$ TRUE, FALSE. A XOR is used to assign the opposite species (flip the species[...] bit) to the newly discovered node if o(CURRENT NODE, NEW NODE) is "different", and to assign the same species if the observation is "same" [1].
Paths between already visited vertexes (non-white vertexes) are checked in order to find inconsistencies. Both edge traversal directions are checked in order to handle cases where observation are directed (i.e. $o(edge(v, u)) \neq o(edge(v, u))$). If an inconsistency is found, we return `FALSE`, otherwise we return true.
Note that if the observation graph G is composed by more than one connected component assigning an arbitrary species to the first vertex encountered in each connected component does not compromise the solutions, since we are not asked to find the correct species assignment but we are just asked to find inconsistencies.

4 Exercise 4

4.1 Point 1

We assume the minimum spanning tree T is represented as an adjacency-mapped graph. *weight* is the weight mapping for every edge in the minimal spanning tree. The other parameters must be given as described in the assignment.

```
FUNCTION IS-MST-MINIMAL(T=(V,E), weight, v, w, c):
  P[w] = NIL
  DEFINE-PARENT(T, P, Adj[w], w)
  s = v
  while s $\neq$ w:
    edge_w = weight(edge (s, P[s]))
    if edge_w > c:
      return FALSE
```

[1] TRUE, FALSE mean "different" and "same" when they are result of $o(\ldots)$. When assigning to `species[...]`, they represent an arbitrary assignment of the two species of butterfly.

```
    s = P[s]

  return TRUE

FUNCTION DEFINE-PARENT(G=(V,E), P, S, v):
  for each vertex u ∈ S:
    P[u] = v
    DEFINE-PARENT(G, P, Adj[u] \ {v}, u)
```

Listing 4: Solution for exercise 4 point 1

The algorithm works by walking the entire tree with `DEFINE-PARENT` in order to define a parent relation considering w as the root. Then, this relation is used to define the path between v and w, and the weights of every edge in the path from v to w are memorized in `edge_w` and compared and checked against the weight of (v, w). If we find an edge in the path with weight bigger than (v, w) we can then replace that edge with (v, w), so we return `FLASE` *(sic)* [2] since we the old MST is not minimal anymore. The complexity of this is $O(|V|)$ since the total number of edges in a tree is linearly dependent to the number of vertices (i.e.: $|E_T| = |V| - 1$).

4.2 Point 2

```
FUNCTION MAKE-MST-MINIMAL(T=(V,E), weight, v, w, c):
  P[w] = NIL
  DEFINE-PARENT(T, P, Adj[w], w)
  s = P[s]
  s_max = s
  max = weight(edge (s, P[s]))
  while s ≠ w:
    edge_w = weight(edge (s, P[s]))
    if edge_w > max:
      max = edge_w
      s_max = s
    s = P[s]

  if max < c:
    return
  else:
    remove edge (max_s, P[max_s]) from T // O(|V|) cost operation
    add (v, w) to T // constant cost
```

[2] https://medium.com/@DanielC7/dbf8773df767

Listing 5: Solution for exercise 4 point 2

For what said before, this algorithm updates T to a valid MST and runs in $O(|V_T|)$ which is always $< O(|E|)$. In order to find the new MST, we need to find remove the edge with highest weight in the path from v to w and add (v, w) to the MST.