CSOR W4231 Analysis of Algorithms I - Spring 2021
Homework #2

Joseph High - `jph2185`

February 27, 2021

Problem 1

Construct a modified BFS algorithm on G starting from the node s such that the algorithm records the number of nodes in each layer. Search for a layer at level $k \in \{1, \ldots n/2\}$ with exactly one node. Return the node contained within this layer.

Pseudocode:

```
SEARCHANDDESTROY($G, s$)
    $BFS$ = BFS-MODIFIED($G, s$)
    for $u \in G$ do:
        if $BFS.dist[u] < n/2$ and $u.length(L[i]) == 1$ then:
            $v = u$
            break
        end if
    end for
    return $v$
end

BFS-MODIFIED($G = (V, E)$, $s$)                      // Only the modified portion shown
//Add the following above the while-loop in the original BFS pseudocode
    $n = size(V)$                        // Define number of nodes in $G$. Running time: $O(1)$
    $L = [\ ]$                           // Initialize array for each layer. Running time: $O(n)$
    $L[0] = s$

//Add the following to the while-loop & above the for-loop in the BFS pseudocode
    for $i = 1$ to $length(L)$ do:
        $L[i] = G.neighbor[u]$                          // Store all neighbors of $u$ in layer $L_i$
    end for
    return $L$
end
```

Running Time: The for-loop in the SearchAndDestroy function iterates at most $n/2$ times, so it has a running time of $O(n)$. The modification to the BFS algorithm requires $O(m)$ time., and the original BFS algorithm has a running time of $O(n+m)$. Therefore, the total running time is

$$O(n+m) + O(n) + O(m) = O(n+m)$$

Correctness:
There must exist at least one layer L_k for $k \in \{1, \ldots, n/2\}$ that excludes s and t which contains exactly one node. Indeed, s is in layer L_0 and, by the supposition, t is contained in layer L_δ for some $\delta = dist(s,t) > n/2$. Since the distance between s and t is strictly greater than $n/2$, then at least one layer must contain exactly one node. Indeed, if each of these layers contained $c > 1$ nodes, then the number of nodes in the union of these layers would exceed the number of nodes in G. That is, $c * (n/2) \geq n = |V|$, which is a contradiction.

Problem 2

If G contains a vertex from which all other vertices in G can be reached, then such a vertex must lie in a source SCC (see correctness argument for proof). Based on this, we can run the depth-first-search (DFS) algorithm on G from any vertex to find a source node, which must lie in a source SCC. In particular, run DFS and compute the finish times for each node. The node with the largest finish time is the source node. Next, run BFS from the node with the largest finish time, recording which nodes have been discovered/explored. If all nodes are explored in BFS, this implies that all nodes are reachable from this node. In this case, return 'True' (i.e., there exists a node in G from which all other nodes can be reached). Otherwise, return 'False' (i.e., such a node does not exist in G).

Pseudocode:

```
1:  Q2_Algorithm($G = (V, E)$)
2:     Run DFS($G$); compute $finish(u)$ for all $u \in V$.          // $O(n+m)$
3:     Compute $\max_{u \in V} finish(u)$, and                        // $O(n)$
           define $v^* = \max_{u \in V} finish(u)$
4:     Run BFS($G, v^*$); record all vertices explored in BFS        // $O(n+m)$
5:     Output all vertices discovered in BFS, and
           define $V_{BFS} = \{w \in V : w$ discovered in BFS($G, v^*$)$\}$
6:     if $size(V_{BFS})$ == $size(V)$ :
7:         return 'True'
8:     else:
9:         return 'False'
10:    end if
11: end
```

Running Time: Running DFS on $G = (V, E)$ with $|V| = n$ and $|E| = m$ requires a run-time of $O(n + m)$. Computing the maximum finish time in the second step requires $O(n)$ time. The call to BFS in the third step requires $O(n + m)$ time. Storing the all explored vertices from BFS requires $O(n)$ time. Therefore, the total running time is

$$O(n + m) + O(n) + O(n + m) + O(n) = O(n + m)$$

Correctness: We first show that running DFS to find the node with the max finish time is correct.
Claim: If G contains a vertex from which all other vertices in G can be reached, then such a vertex must lie in a source SCC.
Let SCC_{source} be a source SCC in the metagraph of G, and consider a node $s \in SCC_{source}$. Now, suppose to the contrary; that is, suppose $\exists\ v \neq s$ from which all other nodes in V are reachable, and that $v \notin SCC_{source}$. In this case, there must exist at least one edge entering a node that lies in SCC_{source} since there is a path from v to all nodes in V. However, this implies that v has a larger finish time than s and that SCC_{source} does not have the max finish time, a contradiction. Therefore, $v \in SCC_{source}$. Now, since s and v are in the same SCC, there must exist a path path from s to v and a path from v to s within SCC_{source}, by

definition of strongly connected components. Then since v can reach all other nodes in V, so can s.
(Correctness of running `BFS`(G, v^*)) By Theorem 22.5 in the course textbook[1], when BFS is run on undirected or directed graphs from a given source vertex $s \in V$, BFS discovers every vertex $v \in V$ that is reachable from the source s.

Alternative Approach:
Alternatively, we can run DFS from the node with the largest finish time, recording the number of DFS trees are in the output. If the second run of DFS from the node with the max finish time returns only one tree, this implies that all nodes are reachable from this node. That is, if DFS discovers all nodes in one exploration, then all nodes are reachable from the node with the largest finish time. In this case, return 'True' (i.e., there exists a node in G from which all other nodes can be reached). Otherwise, return 'False' (i.e., such a node does not exist in G).

Pseudocode for alternative approach:

```
1:  Q2_ALGORITHM($G = (V, E)$)
2:      Run DFS($G$); compute $finish(u)$ for all $u \in V$.        // $O(n + m)$
3:      Compute $\max_{u \in V} finish(u)$, and                      // $O(n)$
            define $v^* = \max_{u \in V} finish(u)$
4:      Run DFS($G, v^*$)                                            // $O(n + m)$
5:      if $\forall$ $v$ discovered in one tree/exploration then:
6:          return 'True'
7:      else:
8:          return 'False'
9:      end if
10:     return
11: end
```

[1]Cormen, T.H., Leiserson, C.E., Rivest, R.L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). The MIT Press.

Problem 3

Because G is a directed acyclic graph (DAG), then G has a topological ordering. Apply the topological sort algorithm on G to obtain the topological ordering of the vertices in G. Consider only the nodes between s and t, inclusive (i.e., $s, v_1, v_2, \dots, v_k, t$). Traverse the nodes in the list between nodes s and t, inclusive (i.e., $s, v_1, v_2, \dots, v_k, t$). Then recursively sum the number of paths from s to t. When the node recursion reaches t, the algorithm will add 1 to the count. If there is a node between s and t that has no outgoing edges, then it adds nothing to the count.

```
PathCounter($G = (V, E), s, t$)
    Run Topological-Sort($G$), and                                          // $O(n + m)$
        output the linked list of vertices between $s$ and $t$, inclusive.
    Define $s, v_1, \dots, v_{k-1}, t$ as the nodes output by Topological-Sort($G$),   // $O(n)$
        excluding all nodes to the left of $s$ and to the right of $t$.
    $count = 0$                                                              // $O(1)$
    if $s == t$ :                                                            // $O(1)$
        $count = 1$                                                          // $O(1)$
    else if $G.Adj[s] = \emptyset$ :                                         // $O(1)$
        $count = count$                                                      // $O(1)$
    else:
        for all $u \in G.Adj[s]$ do:                                         // $O(m)$
            $count = count$ + PathCounter($G, u, t$)
        end for
    end if
    return $count$
end
```

Running Time: Running topological sort on G requires a run-time of $O(n + m)$. The for-loop iterates $outdeg(u)$ times for each $u \in G.Adj[v]$. Then for all vertices between s and t, inclusive, the for-loop iterates $O(m)$ times. Therefore, the total running time is
$O(n + m) + O(n) + O(m) = O(n + m)$

Correctness: Induct on the number of paths from s to t.
Base Case (No paths from s to t): The algorithm starts with a count of zero. Since there are no paths from s to t, the algorithm returns a count of 0, the desired output.
Induction Hypothesis: Suppose there are a total of k paths from s to t. Further suppose that the algorithm correctly computes the number of paths to be k.
Induction Step: Suppose there are a total of $k + 1$ distinct $s - t$ paths. Let p be one of the $k + 1$ paths such that p contains nodes and edges that overlap with nodes and edges of other distinct paths. Such a path must exist. Now consider all $s - t$ paths but p; there are k of them. From the induction hypothesis, the algorithm will enumerate the k paths correctly. Now, because p overlaps with other paths already computed by the algorithm and part of the recurision, then this path will be counted as well. A count of $k + 1$ paths is returned.

Problem 4

Let (b_1, b_2) represent the state of the two bottles, where b_1 and b_2 are the number of liters contained in the first and second bottle, respectively, in the given state. We are given an initial state of (x, y). Run BFS from the initial state node (x, y) and generate the BFS tree using each of the admissible operations of **FILLUP(i)**, **EMPTY(i)**, and **POUR(i, j)**. At each subsequent leaf node in the BFS tree, apply the operations again. Repeat until one of the bottles has exactly A liters. That is, repeat until the state is either (A, b_2) or (b_1, A), for any b_1 and b_2.

Pseudocode provided on the next page. It's too long to fit on this page.

Running Time: Running time depends on the number of outgoing edges from each node in the BFS tree (not including leaf nodes), and the number of outgoing edges at each node is conditional on the current state of the bottles. In the worst case, the maximum number of outgoing edges at any one node is 6 since the there are 6 possible actions in every state. Then the maximum possible number states generated at each node is 6. Thus, the number of possible states explored in BFS is $O(6^d)$, where d is the depth of the BFS tree. Therefore, the running time of BFS is $O(6^d)$.

Correctness: Since BFS computes the shortest path, this algorithm computes the length of the shortest path to one of the desired states. That is, it computes the smallest number of steps required for at least one of the two bottles to contain exactly A liters of water.

Note again: Pseudocode provided on next page.

```
BottleFillSearch($X, Y, x, y, A$)
    $V_{\mathcal{S}} = $ StateSpace($X, Y, x, y, A$)
    $E = \{(V_{\mathcal{S}}[k-1], V_{\mathcal{S}}[k]) : \ \forall\ k\}$
    do
        Run BFS($G = (V_{\mathcal{S}}, E), (x, y)$)
    while $(A, y)$ and $(x, A)$ not discovered
    return $length$(BFS.$shortestpath$)
end
State space used to generate nodes in BFS
StateSpace($X, Y, x, y, A$)
    Let $B_1$ and $B_2$ denote bottles 1 and 2, respectively.
    states $= [(x, y)]$                          // Initialize state space array with initial state
    if $(x < A)$ and $(y \neq A)$ :
        FILLUP($B_1$)
        states $+= (X, y)$                       // Augment to states
    end if
    if $(x \neq A)$ and $(y < A)$ :
        FILLUP($B_2$)
        states $+= (x, Y)$                       // Augment to states
    end if
    if $x > 0$ :
        EMPTY($B_1$)
        states $+= (0, y)$                       // Augment to states
    end if
    if $y > 0$ :
        EMPTY($B_2$)
        states $+= (x, 0)$                       // Augment to states
    end if
    if $(x > 0)$ and $(y < Y)$ and $(x + y \geq Y)$
        POUR($B_1, B_2$)
        states $+= (x - \min\{x, Y - y\},\ y + \min\{x, Y - y\})$      // Augment to states
    end if
    if $(x < X)$ and $(y > 0)$ and $(x + y \geq X)$
        POUR($B_2, B_1$)
        states $+= (x + \min\{y, X - x\},\ y - \min\{y, X - x\})$      // Augment to states
    end if
    return states
end
```

Problem 5

Proof. ($\Longrightarrow$) Suppose there exists a strongly connected component containing both x and $\neg x$ in G_ϕ. Then, by definition of SCCs, there is a path from x to $\neg x$ and a path from $\neg x$ to x. If we assign $x = 1$ (i.e., True), then $\neg x = 0$ (i.e., False). Now consider the path from x to $\neg x$: $x \Longrightarrow v_1 \Longrightarrow \cdots v_k \Longrightarrow \neg x$. At least one implication on this path must have a truth value of 0 (or False) since $x = 1$ and $\neg x = 0$. That is, there must be one implication such that $1 \Longrightarrow 0$ which has a value of 0. If instead x is given a value of False, then the truth values of all implications on this path are True. However, this implies that the path from $\neg x$ to x contains an implication with a truth value of 0 (or False). Therefore, there must exist at least one clause in the SCC containing both x and $\neg x$ with a truth value of $0 \implies \phi = 0$. Hence, ϕ is not satisfiable.

($\Longleftarrow$) Suppose G_ϕ does not contain any SCCs containing both x and $\neg x$. It suffices to find an appropriate satisfying truth assignment for ϕ. To accomplish this, we must find an appropriate way to assign truth values to each of the literals in G_ϕ such that the conjunction of all implications (edges in the graph) have a truth value of 1. For any x and y, the implication $x \Longrightarrow y$ has a truth value of 1 whenever $y = 1$, for any true value of x. Recall from the supposition that all literals and their negations in G_ϕ are not reachable from one another. Using these facts, assign a 1 to all literals v such that there is a path from x to v; do this for all literals in G_ϕ. Since literals cannot be reached by their negation in G_ϕ, there will be no implications in G_ϕ of the form $1 \Longrightarrow 0$. This results in a satisfying truth assignment for ϕ. Alternatively, we can use DFS to find all sink SCCs in G_ϕ and assign a truth value of 1 to all literals in the sink SCCs. □

Algorithm/Pseudocode for `2SAT`
Using the if and only if statement proved above, we can determine whether G_ϕ is satisfiable by computing the strongly-connected components in G_ϕ and subsequently evaluating whether the literals and their negation are in the same SCC. This can be done by running `SCC`(G_ϕ), which runs DFS twice.

Pseudocode provided on the next page. It's too long to fit on this page.

Running Time: The call to `SCC`(G_ϕ) requires $O(n + m)$ time. The first for-loop on line 6 iterates at most n times, so the running time is $O(n)$. The for-loop on line 16 iterates at most n times; in this case, the nested for-loop on line 17 will only run once since if the for-loop on line 16 runs n times it implies that there are n SCCs and thus each *SCC* contains only one node. Therefore, the running time of lines 15 to line 20 is $O(n)$. By a similar argument, the running time of lines 21 to line 25 is $O(n)$. Lines 26 and 28 require $O(1)$ time. The total running time is then $O(n + m)$. Indeed,

$$O(n + m) + O(n) + O(n) + O(n) + c = O(n + m)$$

Correctness: Correctness in determine satisfiability was shown above. Using sink SCC to compute truth assignments was also argued in the proof above.
Note again: Pseudocode is provided on next page

```
2SAT-Solver($G_\phi$)
    Run SCC($G_\phi$) to compute SCCs
    Let $SCC$ = output of SCC($G_\phi$)
    $satisfiability$ = “ ”                                  // Initialize empty string
    $truthAssignment$ = [ ]
    for each $SCC$ do:
        if ($x \in SCC$) and ($\neg x \in SCC$) then:
            $satisfiability$ = ‘UNSATISFIABLE’
        else:
            $satisfiability$ = ‘SATISFIABLE’
        end if
    end for
    return $satisfiability$

    if $satisfiability$ == ‘SATISFIABLE’ then:
        for each $SCC.sink \in G_\phi$ do:
            for each $v \in SCC.sink$ do:
                $v = 1$
            end for
        end for
        for each $SCC.source \in G_\phi$ do:
            for each $w \in SCC.source$ do:
                $w = 0$
            end for
        end for
        Let $truthAssignment$ = the truth values of all nodes computed above.
    end if
    return $truthAssignment$
end
```