CSCE 411-200, Fall 2016
Homework 1 Solutions

Problem 1: Suppose you are given an $n \times n$ boolean matrix, which is the adjacency matrix representation of an undirected graph with n vertices. Design a brute force / exhaustive search algorithm that determines whether or not the graph represented by the matrix is a star (i.e., there exists one vertex that is connected to all the other vertices but those are the only edges). What is the running time of your algorithm as a function of n?

Solution: Note that if k is the center of the star, then the adjacency matrix A consists of all zeroes, except for all ones in row k and all ones in column k (but with zero in $A[k, k]$). My brute force algorithm tries all possible choices for k (from 1 to n) and for each one, checks if A has the correct pattern of zeroes and ones. In C++-like pseudocode:

```
input:  boolean n x n matrix A
// function to check whether A is correct, assuming k is the center
bool check_center(int k) {
  for i = 1 to n {
    for j = 1 to n {
      // diagonal elements must be 0
      if ((i == j) and (A[i,j] != 0)) return false
      // row k and column k must hold 1's
      if (((i == k) or (j == k)) and (A[i,j] != 1)) return false
      // all other elements must be 0
      if (A[i,j] != 0) return false
    }
  }
}
// main: try all possible centers
for k = 1 to n {
  if (check_center(k) == true) return true
}
// if you get here, then none of the vertices work as the center
return false
```

Running time: Each call to `check_center` takes $\Theta(n^2)$ time, since there are two nested for-loops from 1 to n, and the innermost for-loop body takes constant time. Since `check_center` is called n times, the total running time is $\Theta(n^3)$.

Problem 2: Consider the following problem: given n positive integers, separate them into two groups such that adding all the numbers in one group gives the same result as adding all the numbers in the other group. For example, if the numbers are 1,2,3,4 then the two groups could be $\{1, 4\}$ and $\{2, 3\}$, which both sum to 5. For another example, if the numbers are 5,6,11, then the two groups coupld be $\{5, 6\}$ and $\{11\}$ (so the groups do not have to be of the same size). Describe an exhaustive search algorithm for this problem. What is the asymptotic worst-case running time of your algorithm? Justify your answer.

Solution: The idea of the algorithm is consider every possible subset X of the input set S, add up the numbers in X, add up the numers in $S - X$, and see if they are equal.

```
input:  S, set of n positive integers
let X1, X2, ..., Xp be a list of all the subsets of S
```

```
// no need to give pseudocode for algorithm to compute powerset of S
for i = 1 to p {
  let sum1 be the sum of all the elements in Xi
  let sum2 be the sum of all the elements in S - Xi
  if (sum1 == sum2) return Xi and S - Xi
}
// if you get here then no subset worked
return "not possible"
```

Correctness follows because this algorithm is just following the definition of the problem.

Running time: There are 2^n subsets of S, so the for-loop is executed 2^n times. Each iteration of the for-loop involves at most n additions and a small constant number of other operations. Thus the total running time is $\Theta(n \cdot 2^n)$.

Problem 3: Exercise 22.4-2 in [CLRS]. (Given a DAG $G = (V, E)$ and two vertices s and t in V, compute the number of simple paths from s to t in linear time.) Explain why your algorithm is correct and argue why your algorithm has $O(V + E)$ running time (this is what "linear time" means for graphs). *Hint:* Use topological sort.

Solution: The key idea is that the number of simple paths from s to t is equal to the sum, over all incoming neighbors u of t, of the number of simple paths from s to u. To do this calculation, we need the number of paths for all of t's incoming neighbors before we calculate the number for t, so we need the number for all the incoming neighbors of t's incoming neighbors before that, etc. That is, we need to calculate these numbers in an "outward" manner from s. The order for doing the calculation is what we get from a topological sort!

```
input:  DAG G = (V,E)  // let n = |V|
let v[1],v[2],...,v[n] be result of running topological sort algorithm on G
for i = 1 to |V| {
  // number of path from s to itself is 1
  if (v[i] == s) num_paths[i] = 1
  // for other nodes, number of paths is sum, over all incoming neighbors,
  // of their number of paths; if there are no incoming neighbors then 0
  else {
    num_paths[i] = 0
    for each v[j] such that (v[j],v[i]) is in E {
        num_paths[i] = num_paths[i] + num_paths[j]
    }
  }
}
return num_paths[t]
```

Correctness: Show by induction on i, the order in which vertices appear in the topological sort, that num_paths$[i]$ is correct.

Basis: Since there is nothing different in the argument for $i = 1$ and all the other values of i, we can have a vacuous basis case by considering $i = 0$.

Induction: We will use strong induction. Assume that num_paths$[j]$ is correct for all $j < i$, and show that num_paths$[i]$ is correct. If $v_i = s$, then num_paths$[i]$ is set to 1, which is the correct value. Otherwise, num_paths$[i]$ is set to the sum, over all j such that $(v[j], v[i])$ is in E. By the correctness of topological

sort, we have already computed `num_paths`$[j]$ when `num_paths`$[i]$ is computed, i.e., $j < i$. By the (strong) inductive hypothesis. `num_paths`$[j]$ holds the correct number of paths from s to $v[j]$. Thus the calculation of `num_paths`$[i]$ is correct.

Running time: Topological sort takes $\Theta(V + E)$ time. The rest of the algorithm consists of a doubly-nested for-loop. The outer for-loop is executed V times. The amount of work done inside the outer for-loop, other than the inner for-loop, is constant per iteration. The total time spent in the inner for-loop, added up over *all* iterations of the outer for-loop is $\Theta(E)$, that is, each edge is considered once. Thus the total time is $\Theta(V + E)$.

Problem 4: Exercise 22.5-3 in [CLRS]. (What happens if the strongly-connected components considers the vertices in *increasing* order of finishing time when doing the second DFS?) Either prove that the modified algorithm is correct or give a counter-example.

Solution: The algorithm is no longer correct. As a counter-example, consider the graph that was used as the running example in the lecture slides from class. The increasing order of finishing times from the first DFS is c, d, b, e, a, h, g, f. When doing DFS on the transpose graph, we get the following execution (shows nesting of the recursive calls):

```
[ c [ b [ a [ e ] [ f [ h [ g ] ] ] ] ] ] [ d ]
```

This consists of just two DFS trees, one rooted at c and other rooted at d. The tree rooted at c contains three of the SCC's of the original graph, instead of just one.

Problem 5: Exercise 22.5-7 in [CLRS]. (Give an efficient algorithm to determine if a directed graph $G = (V, E)$ is "semi-connected", meaning that for every pair of vertices u and v, either there is a path from u to v or there is a path from v to u.) Prove the algorithm is correct and analyze its running time.

Solution: First note that all the nodes in a strongly connected component of G are reachable from each other, so we can just consider the component graph of G, G^{SCC}. Recall that G^{SCC} is always a DAG.

Let's find a characterization of DAGs that tells us whether or not they are semi-connected. Note that if there are two sinks (vertices with no outgoing edges) then neither can reach the other, and the same is true of there are two sources (vertices with no incoming edges). This leads us to think that we need there to be a path that goes through all the vertices (it's fine to have other edges as well).

Claim: A DAG $H = (V', E')$ is semi-connected if and only if there is a path through all the vertices. Let $m = |V'|$.

Proof of claim: $\Rightarrow$: Suppose H is semi-connected. Let $v_1, v_2, ..., v_m$ be a topological sort of V'. Consider each v_i, v_{i+1} in the list. By the definition of topological sort, (v_{i+1}, v_i) cannot be in H. Since H is semi-connected, there must be a path from v_i to v_{i+1}. So there must be a path from v_1 to v_2 to v_3 to ... to v_m, which means—since H is a DAG—there must be an edge from v_1 to v_2 to v_3 to ... to v_m.

$\Leftarrow$: Suppose there is a path through all the vertices, say $v_1, v_2, ..., v_m$. Then for any two vertices, v_i and v_j, there is a path from the one with the smaller subscript to the one with the larger subscript, so H is semi-connected. (End of proof of claim.)

Putting the pieces together, we get:

```
input: directed graph G = (V,E)
Run SCC algorithm on G to create H = (V',E'), the component graph of G
// the vertices of H are c[1], c[2], ..., c[k]; each one is an SCC of G
Run topological sort on H, resulting in c[1],c[2],...,c[k]
for i = 1 to k-1 {
  if (c[i],c[i+1]) not an edge of H then return false
}
```

```
// if get here, then c[1],c[2],...,c[k] is a path through H
return true
```

Correctness follows from the explanation given before the pseudocode.

Running time: SCC algorithm takes $\Theta(V + E)$ time. Topological sort algorithm takes $\Theta(V' + E')$ time where $H = (V', E')$; note that $V' = O(V)$ and $E' = O(E)$, so topological sort takes $O(V + E)$. Checking for the edges in the for-loop at the end takes $O(E') = O(E)$ time. So the total time is $O(V + E)$.