

# CSCE 411-200: Honors Design and Analysis of Algorithms

## Assignment Cover Page

### Fall 2016

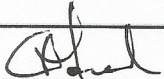
<b>Name:</b>	Carsten Hood
<b>Email:</b>	carstenhood@tamu.edu
<b>Assignment:</b>	Homework 1
<b>Grade (filled in by grader):</b>	

Please list below all sources (people, books, webpages, etc) consulted regarding this assignment (use the back if necessary):

CSCE 411 Students	Other People	Printed Material	Web Material (give URL)	Other Sources
1.	1.	1. wikipedia.org/wiki/Adjacency_matrix	1.	1.
2.	2.	2. wikipedia.org/wiki/Partition_problem	2.	2.
3.	3.	3. wikipedia.org/wiki/Depth-first_search	3.	3.
4.	4. wikipedia.org/wiki/Strongly_connected_component	4.	4.	4.
5.	5.	5.	5.	5.

Recall that TAMU Student Rules define academic misconduct to include acquiring answers from any unauthorized source, working with another person when not specifically permitted, observing the work of other students during any exam, providing answers when not specifically authorized to do so, informing any person of the contents of an exam prior to the exam, and failing to credit sources used. *Disciplinary actions range from grade penalty to expulsion.*

"On my honor, as an Aggie, I have neither given nor received unauthorized aid on this academic work. In particular, I certify that I have listed above all the sources that I consulted regarding this assignment, and that I have not received or given any assistance that is contrary to the letter or the spirit of the collaboration guidelines for this assignment."

<b>Signature:</b>	
<b>Date:</b>	9/12/2016

Carsten Hood  
UIN: 922009787  
CSCE 411-200 Fall 2016  
Homework 1  
Monday, Sep 12

---

**(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  $n$ ?**

Problem statement:

We're looking for a symmetric matrix in which all elements are 0s except for one row and column containing only 1s, except for their intersection, which should also be 0.

Algorithm summary:

Scan each row in the matrix using a double loop through  $1 \leq i \leq n$  and  $1 \leq j \leq n$ . Elements  $A_{ij}$  and  $A_{ji}$  should always be equal. Count the number of '1's in each row. One row should have  $n - 1$  '1's. Every other row should have exactly one '1'.

Algorithm pseudocode:

```
let boolean full_row_found be false
1 for i from 1 to n
2   let int ones_in_row be 0
3   for int j from 1 TO n
4     if  $A_{ij} \neq A_{ji}$  then return false      // must be symmetric
5     if  $A_{ij} = 1$  then if  $i = j$  return false  // shouldn't be an edge from a vertex to
itself
6     else if  $A_{ij} = 1$  then increment ones_in_row by 1  // count 1s
7     if full_row_found = false and ones_in_row == n-1 then set full_row_found to true
8     else if ones_in_row != 1 then return false  // not enough 1s in the row
9 return full_row_found
```

Running time:

As a brute force search process, the algorithm processes each element in an  $n \times n$  matrix, so the complexity is  $O(n \times n) \Rightarrow O(n^2)$ .

(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 could 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.

Algorithm summary:

Calculate the sum  $S$  of all  $n$  integers. If  $S$  is odd, no solution exists. If  $S$  is even, calculate  $S/2$ . Then iteratively consider the possible subsets of integers, calculate the sum of the elements of each, and compare this subset sum to  $S/2$ . If the subset's sum equals  $S/2$ , return the subset and its complement subset as the output. If all possible subsets have been exhausted without any solution then none exists.

Algorithm pseudocode:

```

input: set SET, size N
output: pair (set SUBSET, set COMPLEMENT), or NULL
int SUM := sum_elements( SET )
if SUM*0.5 % 2 != 0 then return NULL
int HALF = SUM / 2 // determine desired subset sum total
declare sets SUBSET, COMPLEMENT, SUBSETS1, SUBSETS2, EMPTYSET
SUBSETS1.add ( EMPTYSET )
for J from 1 to N // calculate subsets until correct one is found
    set SUBSETS2 = SUBSETS1
    for each SUBSET in SUBSETS2 // build subsets
        SUBSET.add( SET[J] )
    for each SUBSET in SUBSETS2 // add subsets to subsets container
        SUBSETS1.add( SUBSET )
        if sum_elements ( SUBSET ) = HALF // check if subset satisfies requirement
            break out of double for loop
    SUBSET = NULL // clear subset
if SUBSET = NULL then return NULL // return if no correct subset found
// determine complement of correct subset
COMPLEMENT = SET
int K = 1, int J = 1
for int J from 1 to COMPLEMENT.size
    if COMPLEMENT[J] = SUBSET[K] then
        COMPLEMENT.remove_at_index(J)
        K += 1
    else J += 1
return pair (SUBSET, COMPLEMENT)

```

Running time:

$O(2^n)$ . In the worst case it takes  $O(2^n)$  time to recursively consider all of the subsets of  $n$  integers, all other operations being constant. This is because a set of  $n$  elements has  $2^n$  subsets, and potentially each is processed by the double for-loop. Determining the complement of the discovered subset (at the end) takes only linear time  $O(n)$  because like elements in both sets retain their relative orders; hence, this operation's complexity can be disregarded.

(3) Exercise 22.4-2 in [CLRS]. 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.

22.4-2: "Give a linear-time algorithm that takes as input a directed acyclic graph  $G = (V, E)$  and two vertices  $s$  and  $t$ , and returns the number of simple paths from  $s$  to  $t$  in  $G$ . For example, the directed acyclic graph of Figure 22.8 contains exactly four simple paths from vertex  $p$  to vertex  $r$ :  $pov$ ,  $poryv$ ,  $posryv$ , and  $psryv$ . (Your algorithm needs only to count the simple paths, not list them.)"

Algorithm summary, running time, and correctness:

Associate a counter value of 1 to vertex  $t$  and counter values of 0 to all other vertices. Then perform a depth-first search rooted at vertex  $s$ . If a non- $t$  vertex is reached, set its counter value to the sum of all adjacent vertices' counter values. If the vertex is  $s$  then return its counter value as the solution. If vertex  $t$  is reached, cease operating on that branch. As a variation of a depth-first search, this takes  $O(V+E)$  time. All other operations (e.g., marking the vertices) take constant time. This algorithm is correct because the counter value of every vertex always represents the number of its simple paths to vertex  $t$  (starting with  $t$  itself, whose counter value is 1 since it is its own path to itself).

Algorithm pseudocode:

**input:** graph  $G$ , vertices  $Vs$ , edges  $Es$ , vertex  $S$ , vertex  $T$

**output:** int (sum of simple paths from  $S$  to  $T$  in  $G$ )

**for** each vertex  $V$  in  $Vs$

**set**  $V$ .counter to 0

**set**  $T$ .counter to 1

**let** STACK be a stack for vertices

STACK.push( $S$ )

**while** STACK isn't empty      // stack-based depth-first search

**let**  $V = \text{STACK.pop}()$

**if**  $V = T$  then              // stop operating on this branch

**continue**

**for** each vertex  $W$  in  $G$ .outgoing-edges( $V$ )

**if**  $W$  is not marked discovered

$V$ .counter +=  $W$ .counter      // set to sum of adjacent vertices' counters

            mark  $W$  as discovered

            STACK.push( $W$ )

**if**  $V = S$  then

**return**  $V$ .counter              // DFS is finished

}

**return** 0;



(4) Exercise 22.5-3 in [CLRS]. Either prove that the modified algorithm is correct or give a counter-example.

22.5-3: “Professor Bacon claims that the algorithm for strongly connected components would be simpler if it used the original (instead of the transpose) graph in the second depth-first search and scanned the vertices in order of increasing finishing times. Does this simpler algorithm always produce correct results?”

No. As a counterexample, consider the graph G:

(B)  $\longleftrightarrow$  (A)  $\longrightarrow$  (C)

By inspection, G should have two SCCs: (A, B) and (C).

Now run the modified SCC algorithm:

Step 1 - first DFS ordering produces: B, C, A.

Step 2 - second DFS yields only (B,A,C) as the SCC, which is incorrect.

---

(5) Exercise 22.5-7 in [CLRS].

22.5-7: “A directed graph  $G = (V, E)$  is “semiconnected” if, for all pairs of vertices  $u, v \in V$ , we have  $u \rightarrow v$  or  $v \rightarrow u$ . Give an efficient algorithm to determine whether or not G is “semiconnected”. Prove that your algorithm is correct, and analyze its running time.”

Algorithm summary:

Find all SCCs in the graph and consolidate them into single vertices, yielding an acyclic component graph. Perform a topological sort on the component. Check if edges exist between all adjacent pairs in the resulting topologically sorted list. If so, the component graph and original graph are “semiconnected”.

Algorithm pseudo-code:

```
let G1 = G.component-graph()
let LIST = G1.topological-sort()
for int J from 1 to LIST.count-1
    if G1 does not have an edge from LIST[J] to LIST[J+1]
        return false
return true    // no contradictions were found
```

Algorithm running time:

The SCC algorithm and topological sort both take  $O(V+E)$ . Iterating through the list of vertices and checking for directed edges is also  $O(V+E)$ , provided the graph implementation is structured so that it takes linear time to check for edges between given pairs of vertices.

Algorithm correctness:

The algorithm checks whether the component graph is linear, which is a valid approach because a graph is only “semiconnected” if it comprises a linear chain of SCCs. This is because SCCs are semiconnected by definition, since paths exist between all vertices within SCCs; so the deciding factor in determining semiconnectivity is the arrangement of the SCCs, or component graph. If the component graph is not arranged in a linear chain, then because it is necessarily acyclic, there will exist two components which lack a connective path, and hence the original graph cannot be semiconnected.