OLLSCOIL NA hEIREANN, CORCAIGH
THE NATIONAL UNIVERSITY OF IRELAND, CORK

COLAISTE NA hOLLSCOILE, CORCAIGH
UNIVERSITY COLLEGE, CORK

Summer Examinations 2011

Fourth Science Computer Science

CS4407: Algorithms

Sample Final Exam

Dr Carron Shankland (extern)
Prof. J. Bowen (HoD)
Professor G. Provan

(**Instructions** –Answer 5 Questions.)

Time 1.5 Hours

CS4407: Algorithms
Sample Final

Please answer all questions; Total marks: 100
Points for each question are indicated by [xx]

1. **[20]** Consider the *BubbleSort* algorithm.
   (a) **[15]** Use the loop invariance approach to analyse this algorithm.
   (b) **[5]** Use this approach to specify the complexity of the algorithm.

```
procedure Bubblesort(A isoftype in/out Arr_Type)
  to_do, index isoftype Num
  to_do <- N - 1

  loop
    exitif(to_do = 0)
    index <- 1
    loop
      exitif(index > to_do)
      if(A[index] > A[index + 1]) then
        Swap(A[index], A[index + 1])
      endif
      index <- index + 1
    endloop
    to_do <- to_do - 1
  endloop
endprocedure // Bubblesort
```

Assume we have a list with n elements.

Loop Invariant: If A[i]< A[i-1] for any i, then reverse A[i] and A[i-1]

Pre-condition: show the loop invariant holds before the first iteration, when j=1. Here, the subarray consists of just A[n], which is (trivially) sorted.

Exit step: we exit when j=n, which occurs when the last element consists of the largest array element, and array consists of the elements originally in A[1…n] but in sorted order. Hence the entire array is now sorted.

Post-condition: when we exit with j=n+1, the entire array is now sorted.

Induction step: the body of the outer for loop works by moving A[i] to A[i-1], A[i-2], A[i-3] and so on by one position to the left until the proper position for A[i] is found. This is true for all i from 1 to n.

- Use this approach to specify the complexity of the algorithm.

In the worst case, with a reverse-sorted list to start with, the number of exchanges is

$$T_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = (n-1)n/2 \in O(n^2)$$

2. **[20]** Assume that Not All Equal 3SAT (a variant of 3SAT) is NP-Complete. Prove that Not All Equal 3SAT can be reduced to Set Splitting, thus proving that Set Splitting is NP-complete.

   Not All Equal 3SAT

   INSTANCE: Set U of variables, collection C of clauses over U such that each clause has 3 variables.

   QUESTION: Is there a truth assignment for U such that each clause in C has at least one true literal and at least one false literal?

   Set Splitting

   INSTANCE: Collection C of subsets of a finite set S.

   QUESTION: Is there a partition of S into two subsets S1 and S2 such that no subset in C is entirely contained in either S1 or S2?

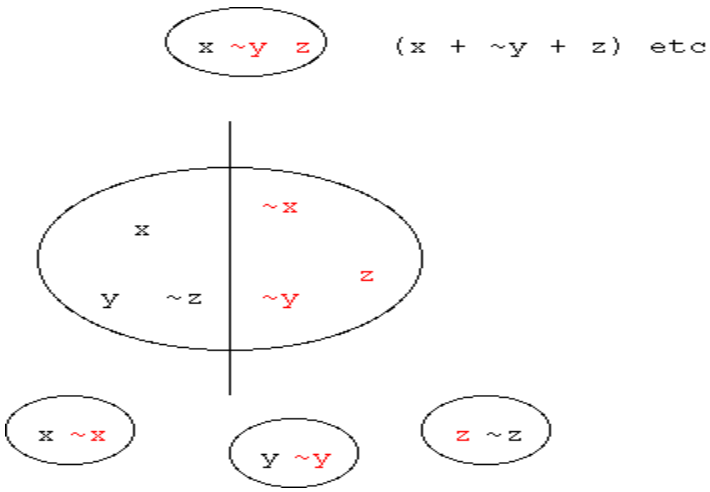
Construct the set S from variables U such that S contains all the variables in the NAE3SAT formula and their negations. Construct the collection C of subsets of S from the clauses as follows: For each variable-negation pair ((x, ~x), (y, ~y) etc) make a subset. Also make a subset for each clause (eg (x + ~y + z)).

Clearly, this construction can be done in polynomial time.

The set S can be split in the required way if and only if the formula has a truth-value assignation that makes it true.

$\Rightarrow$ We prove that if NAE3SAT has a satisfying assignment, then there exists a Set Splitting assignment. If NAE3SAT has a satisfying assignment, then each clause must have at least one variable true and one variable false, and all literals must have a truth assignment; this means that every set representing a variable-negation pair ((x, ~x), (y, ~y) etc) must be split, and every set representing a clause must also be split, since each clause must have at least one variable true and one variable false.

$\Leftarrow$ We prove that if there exists a Set Splitting assignment, then NAE3SAT has a satisfying assignment. If the collection C of subsets of S can be split, then one part of the split set represents "true", the other "false". Each subset must contain at least one "true" variable and one "false" variable, which is exactly what the NAE3SAT problem requires.



3. **[20]** Consider the Travelling Salesman Problem on a complete undirected graph G with a length L(i,j)≥0 for each edge (i,j). Suppose the lengths satisfy

$$L(i,j) \leq L(i,k) + 2\,L(k,j) \text{ for all i, j, k.}$$

(a) **[10]** Provide an approximation algorithm for G.

We can just apply the approximation algorithm shown in class, which consists of (a) generating a minimum spanning tree T for G, (b) computing a pre-order walk on T, and (c) applying the triangle inequality to create a Hamilton cycle H from the pre-order walk, noting that the short-cuts introduced will not increase the length of the pre-order walk.

(b) **[10]** What is the approximation ratio?

First prove (by induction) that for each path between i and j, the length of (i,j) is at most twice the length of the path.

We use the input that $L(i,j) \leq L(i,k) + 2\,L(k,j)$ for all i, j, k, such that $i \neq j \neq k$.

This will give us the fact that, for any walk W, the cost of that walk $c(W) \leq 2\,c(H^*)$, where H* is the optimal Hamilton cycle.

We use the following notation: T stands for a minimum spanning tree, and H for a Hamilton cycle.

With this result, we can then use the result shown in class, that we have a 2-approximation for the optimal Hamilton cycle, using the "standard" triangle inequality $L(i,j) \leq L(i,k) + L(k,j)$ for all i, j, k.

We can apply the 2-approximation algorithm shown in class to get an approximation ratio of 4, given the new version of the triangle inequality.

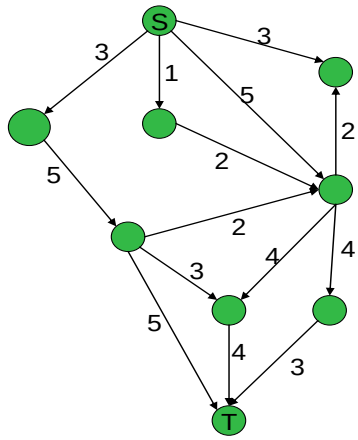
$$c(T) \le c(H^*)$$

$$c(W) = 2c(T) \le 2c(H^*)$$

$$c(H) \le 2c(W)$$

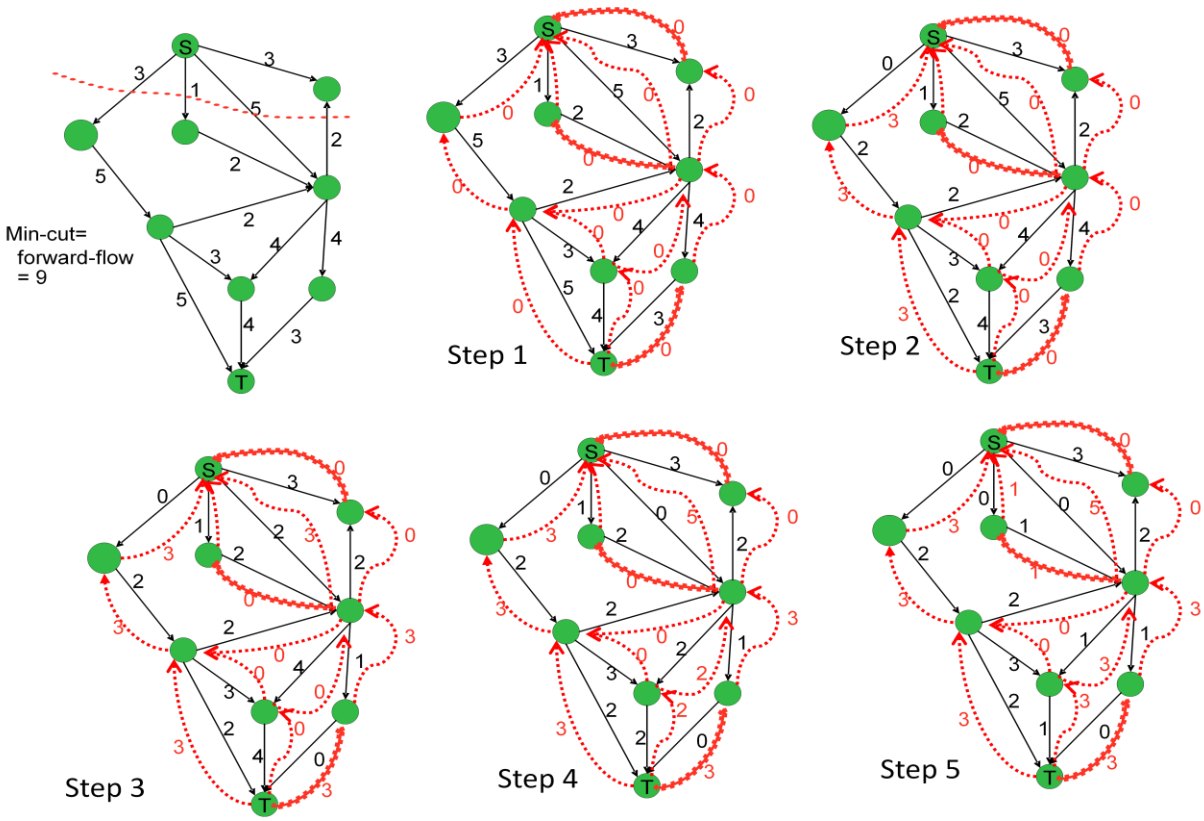
$$\Rightarrow c(H) \le 4c(H^*)$$

4.**[20]** Consider a graph G(V,E), with source node S and sink node T.



For the instance of a flow network shown below, compute the maximum flow. Give the actual flow as well as its value. Justify your answer.

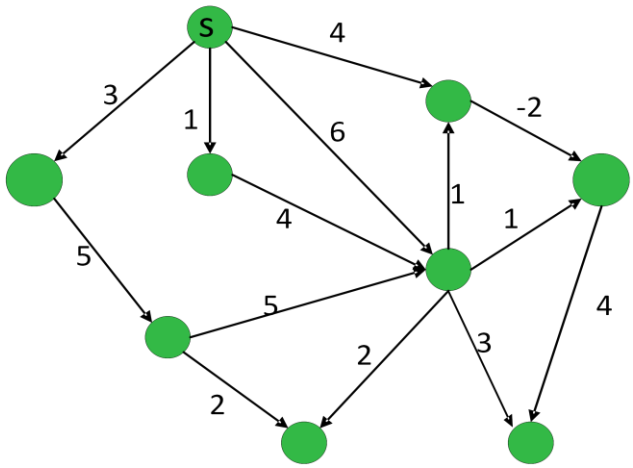
No further augmenting flow when total flow is 9.



5. **[20]** Given the Directed graph G shown below,
   (a) **[10]** Describe an algorithm that can be used to test the graph below for cycles

Using a Depth-First Search algorithm will detect cycles. You just report a cycle if you encounter a node that has already been searched. (The code for this is in your lecture notes.)

   (a) **[10]** Can a greedy algorithm be used to compute min-cost paths from a node s to all other nodes? If not, show why. If yes, show the algorithm you can use.



We can use a greedy algorithm if there are no negative-weight cycles.
One greedy solution is to use Dijkstra's algorithm.

- N: set of nodes for which shortest path already found
- Initialization: (*Start with source node s)*
  - $N = \{s\}$, $D_s = 0$, "s is distance zero from itself"
  - $D_j = C_{sj}$ for all $j \neq s$, distances of directly-connected neighbors
- Step A: (*Find next closest node i*)
  - Find $i \notin N$ such that
  - D_i = min Dj for $j \notin N$
  - Add i to N
  - If N contains all the nodes, stop
- Step B: (*update minimum costs)*
  - For each node $j \notin N$
  - $D_j = \min (D_j, D_i + C_{ij})$
  - Go to Step A