Name(last, first): SURAJ VATHSA.

ID (rightmost 4 digits): 2512

U C L A Computer Science Department

CS 180 Algorithms & Complexity

Final Exam **Total Time: 3 hours** **December 10, 2018**

*** **Write all algorithms in bullet form (as done in the past)** ***

You need to prove EVERY answer that you provide.

There are a total of 8 pages including this page.

Name(last, first): VATHSA, SURAJ

1. (20 points: each part has 10 points)

a. Consider a S-T network N. **Prove** that if **f** is a maxflow in the network N then there is a cut C with its capacity equal to **f**.

- We are given that f is a maxflow.
- If f is a max. flow then there are no more augmenting s-t paths in the residual graph G_f.

Proof:

Assume there was an s-t path in the residual graph G_f. This would mean that we could increase the flow from s to t by atleast 1 (since all flows are integer values). This would give us a flow $f' > f$ which would be a contradiction to the fact that f is a maxflow.

- Consider all the saturated edges in G_f (i.e. correspond to a forward edge with capacity 0). Since there are no s-t paths, we are guaranteed that the saturated edges isolate vertex s from t.
- This implies that the saturated edges form a cut.
- We know that for any cut, Flow into the cut has to equal the flow out of the cut.
- Given that these edges are saturated, the max flow we can push from s to t is the sum of capacities of the cut.

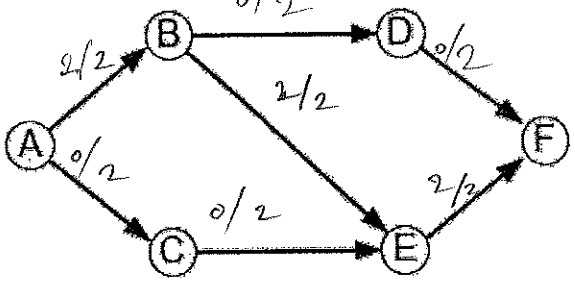
$$\therefore \quad f = c(S,T)$$

We can make this statement since $v(f) \leq c(S,T)$ for any flow.

Name(last, first): VATHSA, SURAJ

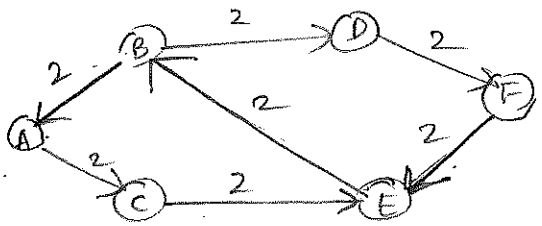
b. Consider the following network with source **A** and sink **F**. If the Ford-Fulkerson max flow algorithm initially finds the path **A,B,E,F** in the network below and sends **2 units** of flow on it, **show the residual network** (also known as the augmented network) and all subsequent steps of Ford-Fulkerson algorithm on this network (all capacities are equal to 2).

flow ↖ 0/2 ↗ capacity



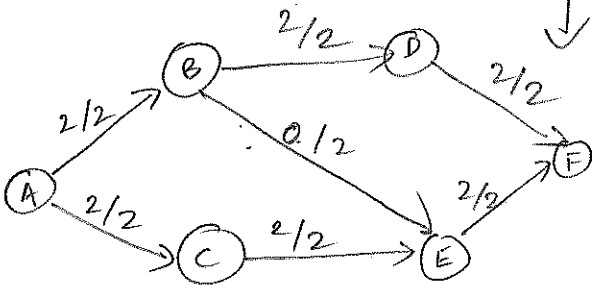
G_1

residual network

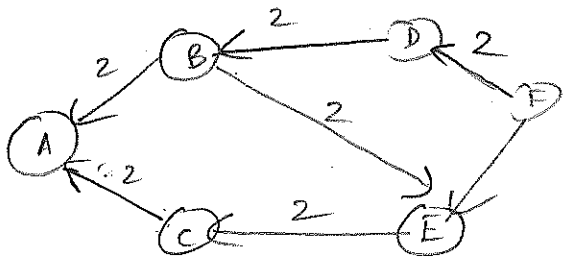


G_{1f}

Choose path A C E B D F in G_{1f}



G_1



G_{1f}

We stop because there are no more s-t augmenting paths in G_{1f}.

∴ Max flow = 2 + 2 = 4

Name(last, first): VATHSA, SURAJ

2. (15 points) a. Given a n x n matrix where all numbers are distinct, design an **efficient algorithm** that finds the maximum length path (starting from any cell) such that all cells along the path are in increasing order with a difference of 1.

b. Analyze the time complexity of your algorithm

We can move in 4 directions from a given cell (i, j), i.e., we can move to (i+1, j) or (i, j+1) or (i-1, j) or (i, j-1) with the condition that the adjacent cells have a difference of 1.

```
Input: mat[][] = {{1, 2, 9}
                  {5, 3, 8}
                  {4, 6, 7}}
Output: 4
The longest path is 6-7-8-9.
```

- If we are given a matrix M with distinct numbers, for a given element, a number of value 1 greater than current number can be in only 1 of the four positions M[i+1][j], M[i-1][j], M[i, j+1] or M[i][j-1]
- We could build up our solution by calculating the longest path that obeys our invariant by summing up the longest paths that obey our invariant from adjacent positions (which can be at most 2).
- Algorithm:

```
longest-path(M):
    Initialize an n x n matrix C that keeps track of the
    length of the longest path so far.
    C[0][0] = 1
    for j = 1, ... n:
        if |M[0][j-1] - M[0][j]| equals 1:
            C[0][j] = C[0][j-1] + 1
        else
            C[0][j] = 1
    for i = 1, ... n:
        if |M[i][0] - M[i-1][0]| equals 1:
            C[i][0] = C[i-1][0] + 1
        else
```

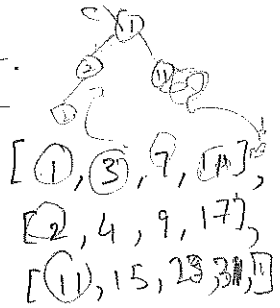
```
C[i][0] = 1
for i = 1, ..., n-1:
    for j = 1, ..., n-1:
        if C[i][j] = 1
            if |M[i][j] - M[i-1][j]| equals 1:
                C[i][j] += C[i-1][j]
            if |M[i][j] - M[i][j-1]| equals 1:
                C[i][j] += C[i][j-1]
max_path = 1
for i = 0, ..., n-1:
    for j = 0, ..., n-1:
        max_path = max(max_path, C[i][j])
return max_path.
```

Time complexity: Given that we iterate through the matrix using a double for loop twice, the time complexity is given by $O(n^2) + O(n^2) = O(n^2)$.

Proof:

1) Base case: When we have only one number in the matrix $M[0][0] \Rightarrow C[0][0] = 1$ so its correct.
2) Assume we have the path length for the longest path for all elements upto some $A[i][j]$
3) For $A[i][j]$, there can be only 2 out of 4 num that can either precede it or succeed it so we only consider $A[i-1][j]$ and $A[j][j-1]$ a check if $A[i][j]$ obeys the invariant (in which case we update the longest path).

Name(last, first): VATHSA, SURAJ



3. (20 points: Each part has 10 points)

a. Consider **d** sorted arrays of integers each containing n1, n2, .. nd numbers. The numbers ni's can be very different. The total number of all elements is **n** (sum of all ni's). Design an **O(n log d)** algorithm that merges all arrays into one sorted list. You may wish to use a data structure that we have discussed in class.

b. Prove a lower bound on sorting **n** numbers in the decision tree model (using comparison exchange).

a.

- Consider the d sorted arrays.
- First, we initialize a min heap consisting of d nodes such that each node is the first element in the d sorted array.
- We are guaranteed that the minimum element in the entire array (largest) lies in this heap because all the array are sorted.
- We also store the information regarding which array the element comes from ($a_1, \ldots a_d$) in the node of the minheap. We also initialize pointers to the first element of each array
- We extract the minimum value and add it to our final result. We advance the pointer by 1 in the array that this element comes from.
- We then add this element to our minheap and re-heapify.
- We keep doing this until we run out of one of the lists.
- When we run out of the list, we add ∞ (or max_int) into our heap with the node associated to the array we just exhausted.
- Adding ∞ will not disturb our final sorted order because every value will have to be $\leq \infty$.
- We keep track of the number of infinities we added to our minheap and keep doing the above procedure when we reach d ∞s in our heap (i.e. we run out of elements in all our d arrays).

Time Complexity:

The time complexity can be calculated as follows:

- We have d sorted lists.
- This gives us a min heap is of size d.
- Each time we add an element to our final sorted list we spend logd time reheapifying.
- There are n elements.
- ∴ Time Complexity = O(nlogd).

↳ Justification:

- Given d sorted lists the minimum element has to be one of the first elements from the list:
  - Assuming it is not.
  - List containing the min element is not sorted.
  - Contradiction to our invariant of d sorted lists.

b. Given n numbers

- In the comparison-exchange model the only two operations we can perform are comparisons and exchanges.
- In the decision tree model, we take two numbers and ask which one is greater, once we determine this we will know the relative ordering of these two numbers.
- In the end there are n! possible orderings of n number out of which one of them is the right ordering.
- These orderings constitute the leaves of the decision tree.
- If we have a max of l levels in the graph then

Name(last, first): VATHSA, SURAJ

4. (15 points)

Consider an array a1,...,an of n integers, that is hidden from us. We have access to this array through an procedure knapsack(.,.). For a set S ⊆ {1,...,n} and an integer k, knapsack(S,k) will output "yes" if there is a subset T ⊆ S such that the numbers indexed in T add up to k, and it will output "no" otherwise.

Design an algorithm that calls knapsack only O(n) times and outputs a set S ⊆{1,...,n} such that the numbers indexed in S add up to k, if such a set exists. You can use ONLY the knapsack function (e.g., you cannot sort the numbers or do any other operations on them).

For example, suppose a1 = 2, a2 = 4, a3 = 3, a4 = 1, and k = 7. Then, knapsack({1,2,3,4},7) returns "yes" and knapsack({1,3,4},7) returns "no". In this case your algorithm can output either of the sets {1,2,4} or {2,3}. Note that for example {1,2,4} are indices of the numbers, that is, a1, a2, and a4.

Algorithm:

Result = {} (empty set)

target_set(S, k):

if knapsack(S, k) == No:

return Result

else:

pick an arbitary element v_i in S and remove it and

if knapsack(S − {v_i}, k) == "yes":

discard v_i:

target_set(S − {v_i}, k)

else:

add v_i to our result:

target_set(S − {v_i}, k − v_i)

Time Complexity: The given algorithm will discard one element in each recursion. Eg we make at most 3 calls to knapsack in each depth of our recursion and there can

be at most n depths. ∴ Time complexity is O(N). Time Complexity (Knapsack) = . Hmake O(N) calls to knapsack.

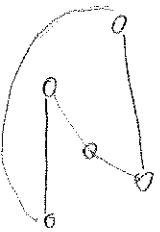
Justification:

- Claim: If knapsack (S, k) is NO then the result we have accumulated so far is a subset.
  - The result that we have accumulated so far are only the numbers that lie in the intersection of all the subsets that add up to k.
  - Once we run out of such numbers we will have a result knapsack (S', k') is NO.

Name(last, first): VATHSA, SURAJ

5. (15points) (UNDIRECTED)

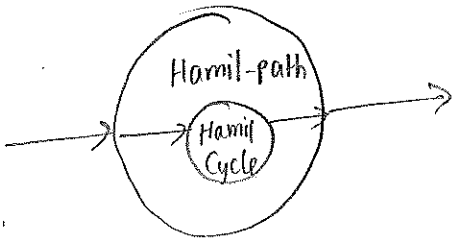
A **Hamiltonian cycle** in a graph with **n** vertices is a cycle of length n, i.e., it is a cycle that visits all vertices exactly once and returns back to the starting point. A **Hamiltonian path** in a graph with n vertices is a path of length n−1, i.e., it is a path that visits all vertices of the graph exactly once.



Hamil-cycle problem is defined as follows: Given a graph G = (V,E), does it have a Hamiltonian cycle? Hamil-path problem is defined as follows: Given a graph G = (V,E), does it have a Hamiltonian path?

Prove that Hamil-path is polynomial-time transformable to Hamil-cyle.

That is Hamil-path ≤P Hamil-cycle.



Input to Hamil-path: Graph $G_1 = (V, E)$

Input to Hamil-path: Graph $G_1 = (V, E)$

The inputs to Hamil-path is polynomial time transferrable to hamil-cycle.

Assuming that we are given a black box that can solve hamil-cycle. For an arbitary graph G_1, let's say we determine if the G_1 contains a hamiltonian cycle. Gtv

Now, we know that G_1 has to contain a hamiltonian path.

Proof:

Let $\{v_1, \ldots, v_n, v_1\}$ be a hamiltonian cycle. Since the hamiltonian cycle visits all vertices exactly once, we know that the only vertex that repeats itself is v_1. We can simply delete the edge

from V_n to V_1 in order to obtain the simple path $\{V_1, \ldots, V_n\}$ which is a path of length $n-1$.

∴ If a graph G has a Hamiltonian cycle, then we can for sure know it has a hamiltonian path.

Output transformation:

The transformation of output does not require any additional time.

∴ Hamil-path is polynomial time transferrable to hamil-cycle.

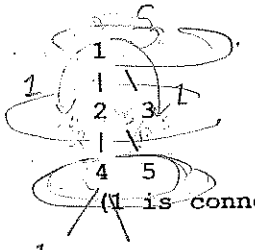
Name(last, first): VATHSA, SURAJ

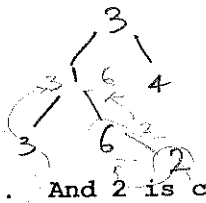
6. (15 points)

You are given a tree T where every node **i** has weight **wi** ≥ 0.

a. Design a polynomial time algorithm to find the weight of the largest weight independent set in T: among all independent sets one with maximum sum of the weights (an **independent set** is a subset of vertices where there are no edges between any of them).

For example, suppose in the following picture w1 = 3, w2 = 1, w3 = 4, w4 = 3, w5 = 6. The maximum independent set has nodes 3,4,5 with weight 4 + 3 + 6 = 13.





(1 is connected to 2 and 3. And 2 is connected to 4 and 5)

b. Analyze the time complexity of your algorithm.

- No two vertices in the independent set can share a parent-child relationship.
- For each node in the tree T, we can find the max sum of the weights from the left child and subtree and from the right subtree.
- We also keep track if the current node was added to the sum or not.

Algorithm:

```
max_weight(root):
    if root == None:
        return 0 and No (because NULL is not really a child).

    else:
        left = max_weight(root.left)
        right = max_weight(root.right)
        if left child and right child were added:
            return max(left + right, max_weight + root.value)
                YES if we end up adding root, NO otherwise
        else
            if only left child was added:
```

return max (right + left , left - leftchildvalue + right + root.value)

else if only right child was added YES if we endup adding root, NO otherwise

return max (right + left , right - rightchild.value + left + root.value),

YES if we end up adding root, NO otherwise.

else :

return (root + left + right , YES)

↳ Justification:

- Claim: No parent–child can exist in our max weighted sum.
  - In any tree T there exists an edge between the parent and the child
  - If they are part of the max weighted sum, then they are ~~not~~ isolated independent.
  - But this is not possible as they share an edge.

6. Time Complexity:

Given that we visit each vertex exactly once, we can solve the problem in O(N) where N is number of vertices.