Final Exam CS 600

Instruction: *Answer the following questions in this document or a word or PDF document and submit it in Canvas according to the Final Exam Procedure.*

*First Problem has 12 points, the other 8, each has 11 points.*

1. (12 Points) Consider n space stations that communicate using laser between each pair of stations. Naturally, the energy needed to connect two stations depends on the distance between them. Therefore, each pair of stations has a different known energy requirement. We want to connect all these stations together using the minimum amount of energy possible. Describe an algorithm for constructing such a communication network.

**Solution:**

This problem can be easily solved using All pair shortest path as it requires you to find the minimum distance between the nodes of a graph.

**Algorithm ModShortPath(G,Source):**

**Input:** A graph G which contains n space stations which have edges that are connected through lasers and the weight d which is the energy required (also distance)

**Output:** A path where minimum amount energy is consumed i.e smallest distance

dist[source] := 0 // Distance from source to source is set to 0

for each vertex v in Graph: // Initializations

if v ≠ source

dist[v] := infinity // Unknown distance function from source to each node set to infinity

add v to Q // All nodes initially in Q

while Q is not empty: // The main loop

v := vertex in Q with min dist[v] // In the first run-through, this vertex is the source node

remove v from Q

for each neighbor u of v: // where neighbor u has not yet been removed from Q.

alt := dist[v] + length(v, u)

if alt < dist[u]: // A shorter path to u has been found

dist[u] := alt

return dist[]

The run time of the algorithm is O(V) where V are the no of nodes (n space stations) in this case.

1. (11 Points) Suppose you are given a network, G, of routers (vertices) and links (edges), with a source, s, and sink, t, together with bandwidth constraints on each edge, which indicate the maximum speed MB/s that the communication link represented by that edge can support. As in a Maximum Flow Algorithm, your goal is to produce a maximum flow from s to t, respecting the bandwidth constraints on the edges. Suppose now, however, that you also have a bandwidth constraint on each router in the network, which specifies the maximum amount of information MB/s that can pass through that router. Describe an efficient algorithm for finding a maximum flow in the network, G, that satisfies the bandwidth capacity constraints on the edges as well as the vertices. What is the running time of your algorithm?

**Solution:** This problem can be solved by using Ford Fulkerson algorithm.

The main concept behind *Ford-Fulkerson algorithm* is to incrementally increase the value of a flow in stages, where at each stage some amount of flow is pushed along an augmenting path from the source to the sink.

1. Initially, the flow of each edge is equal to 0.
2. At each stage, an augmenting path *π* is computed and an amount of flow equal to the residual capacity of *π* is pushed along *π*, as in the proof of
3. The algorithm terminates when the current flow *f* does not admit an

augmenting path.

**Algorithm MaxFlowFF(N):**

**Input:** Flow Network N = (G, c, s, t)

**Output:** A maximum flow *f* for N

**for** each edge *e ∈ N* **do**

*f*(*e*) *←* 0

*stop ←* **false**

**repeat**

traverse *G* starting at *s* to find an augmenting path for *f*

if an augmenting path *π* exists then

// Compute the residual capacity Δ*f* (*π*) of *π*

Δ *←* +*∞*

for each edge *e ∈ π* do

if Δ*f* (*e*) *<* Δ then

Δ *←* Δ*f* (*e*)

for each edge *e ∈ π* do // pushΔ = Δ*f* (*π*) units along *π*

if *e* is a forward edge then

*f*(*e*) *← f*(*e*) + Δ

else

*f*(*e*) *← f*(*e*) *−* Δ // *e* is a backward edge

else

*stop ←* true

*return f //flow*

Augmentation is possible can be easily done in O(E) time.

The run time of this algorithm would be O(E| *f* \*|) where *f\** is the maximum flow found by the algorithm.

If there exists all of the flows of integers, then the while loop is run most of the times where it’s the maximum flow. This happens because the flow increases if worst is considered 1 by each iteration

1. (11 Points) We define SUBGRAPH-ISOMPRPHISM as the problem that takes a graph G, and another graph H, and determine if H is isomorphic to a subgraph of G. That is the problem is to determine whether there is one-to-one mapping, f, of the vertices in H to a subset of the vertices in G such that if (u, w) is an edge in H, then (f(u), f(w)) is an edge in G. Show that SUBGRAPH-ISOMPRHISM is NP-complete.

**Solution:** SUBGRAPH-ISOMORPHISM is actually a comparison problem wherein there exists two graphs say G and H and decide if G is subgraph of H, which basically means each and every node of G exists as subgraph in H which is isomorphic.

* This problem can be thought of a way which are further generalized as either CLIQUE problem or HAMILTONIAN-CYCLE problem which are NP-complete as stated in the textbook.
* We can approach this by simply reducing G using CLIQUE problem which gives us an instance of (G, k) of CLIQUE with n vertices in G.
* Furthermore, we try to create a subgraph isomorphism (G, Q) where Q is actually a complete graph with m vertices and m is minimum of either k or n+1. This type of reduction of can be easily done with polynomial time.
* If the condition k > n is true, then (G, k) does not have any instances of clique and thus we were unable to produce any instance of subgraph isomorphism because we can’t have a graph H in G which contains just n vertices.
* Else, it is evidently clear that G contains a clique of size k if and only if G has a subgraph that is isomorphic to H. As this means same thing.
* Thus, we can conclude that SUBGRAPH-ISOMORPHISM is **NP-complete.**

1. (11 Points) Suppose you are processing many operations in a consumer-producer process, such as a buffer for a large media stream. Describe an external-memory data structure to implement a queue so that the total number of disk transfers needed to process a sequence of n ***enqueue*** and ***dequeue*** operations is O(n/B).

**Solution:** We can easily accomplish this by the use of Linked lists which will allow the insert and remove operations in O(1) run time transfers each.

1. Consider each node present in the linked list is a block of data of size B.
2. A Queue stores the data in a sequential manner from head to tail.
3. When the queue is null, its head and tail points would point to the same index which is 0.

*Queue enables us to perform to important operations namely :*

*• Enqueue*

*• Dequeue*

1. The Enqueue operation is performed by inserting the data at the tail and incrementing the index of the tail by one every time*.*
2. In the same case, a queue can delete an element on the first come first serve (FIFO) basis. Dequeue operation is done by taking the head index and adding incrementing it too.
3. We can perform the insert operations same way we do in the linked list.

To perform an insertion, we access the last block of the list. If the block is empty, we insert new element in the block and transfer it back to the disk once it is done.

1. If the blocks full, we allocate a new block space then insert the newest element in the block.
2. In the similar way, we can remove element from the above list if we assume we know which block holds an item to be removed.
3. The run time of these operations in the Linked List are O (1).

So, the run time for the memory transfer of n operations would be **O(n/B) operations.**

1. (11 Points) Design an O(n2) time algorithm for testing whether a polygon with n vertices is simple. Assume that the polygon is given by the list of vertices.

**Solution:** a **simple polygon** is a flat shape consisting of straight, non-intersecting line segments or "sides" that are joined pair-wise to form a closed path.

**Algorithm: SimplePolygonTest(L)**:

**Input**: An ArrayList L which contains all the vertices of polygon  
**Output**: Whether polygon is simple or not

New\_ArrayList 🡸 new List of pair of end points of line  
**for** x1 in L **do**:  
 **for** x2 in L **do**:  
 **if** (x1, x2) || (x2, x1) in New\_ArrayList **do**:  
 //proceed  
 **else** **do**:  
 **if** x1 is adjacent to x2 :  
 New\_ArrayList.insert((x1, x2))  
**for** (x1, x2) in New\_ArrayList **do**:

// x1 and x2 are points of a line  
 **for** (y1, y2) in New\_ArrayList **do**:

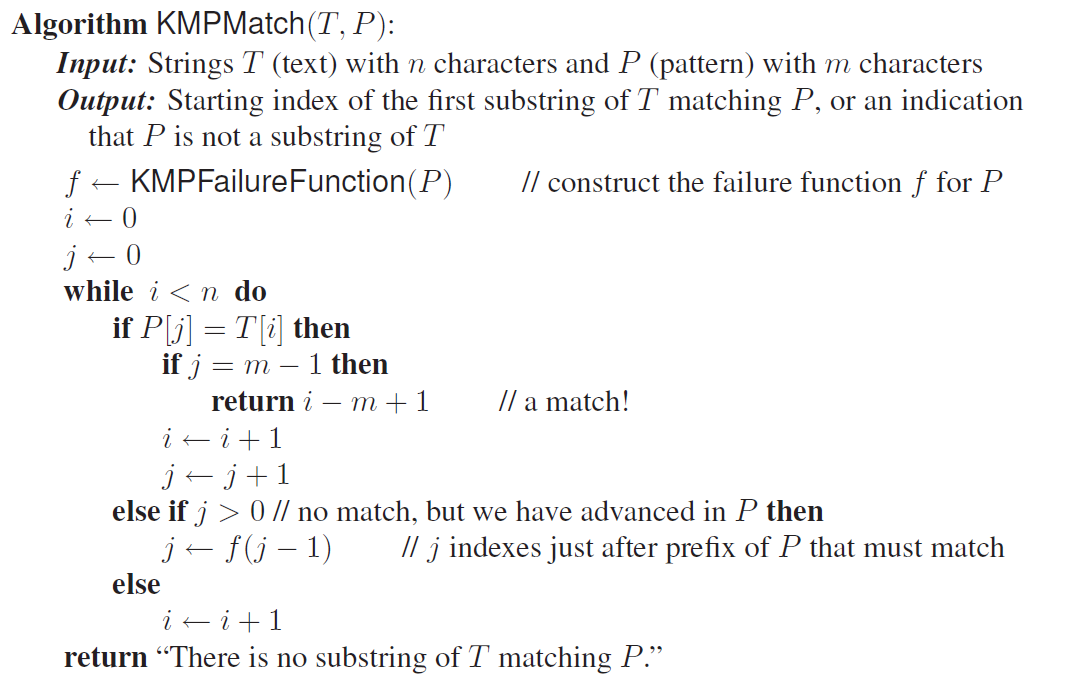
// y1 and y2 are points of a line

**If** ((x1, y1, x2) = CCW && (x1, y1, y2) = CW && (x2, y2, x1) = CW && (x2, y2, y1) = CCW)) **||** ((x1, y1, x2) = COLL && (x1, y1, y2) = CW and (x2, y2, x1) = CW and (x2, y2, y1) = CCW)) **do**  
**return** False

**return** True

1. (11 Points) Show how to modify the KMP string pattern matching algorithm so as to find every occurrence of a pattern string P that appears as a substring in T, while still running in O(n + m) time. (Be sure to catch even those matches that over- lap.)

**Solution:** The following algorithm given below is the KMPMatch algorithm taken from the textbook.



We can easily modify the while loop doing the following:

***while*** *i < n* ***do:***

***if*** *P[j] = T[i]* ***then***

*i 🡸 i + 1*

*j 🡸 j + 1*

***if*** *j =m* ***then***

***Display*** *(Pattern has been found at index,String(i-j)) //Pattern Found*

*j 🡸 f(j-1)*

***else if*** *i < n && P[j] != T[i]* ***then*** *//mismatch occurs after j i.e they do not match f[0..f[j-1]] characters, they will match anyway i.e overlap*

***if*** *j != 0* ***do***

*j 🡸 f[j-1]*

*else:*

*i 🡸 i+ 1*

We can clearly see that the modified KMP loops n times so the run time would be O(n) and we are also aware the KMPFailure Function runs at O(m) time where m are the number of characters.

Thus, the run time of the algorithm would be O (n+m).

1. (11 Points) Consider a network of routers that are connected via a ring of fiber-optics to form a single, simple cycle. A message between two routers a and b routers can be transmitted by routing it clockwise or counter-clockwise around the ring. Given a set, M, of messages to transmit, each specified by a pair (a, b), we want to route all messages in M so as to minimize the maximum load on any link in the ring (that joins two adjacent routers). Such an optimization problem is useful, since such a solution minimizes the bandwidth needed to transmit all the messages in M. Describe a 2-approximation algorithm for solving this problem.

Solution: Let us make an assumption that there are “n” number of messages to be unloaded and their respective weights are w1,w2,w3,…. ,wn. Then let N\* be the optimal number of routers that are required. We know that each of the single routers would not be able to carry more than U units of load. Thus, we get the following formula.

i ≤ KN\* where N\* ≥ 1/K I 🡺①



Let us assume that “N” be the number of routers that the greedy algorithm finds. We prove that it is within a factor two of the minimum possible number, for any set of weights and any value of K.

Let us denote Ij as the set of messages that router “j” loads and let Wj be the total weight of the messages that are present in Ij, so the equation would be

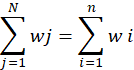
Wj := .



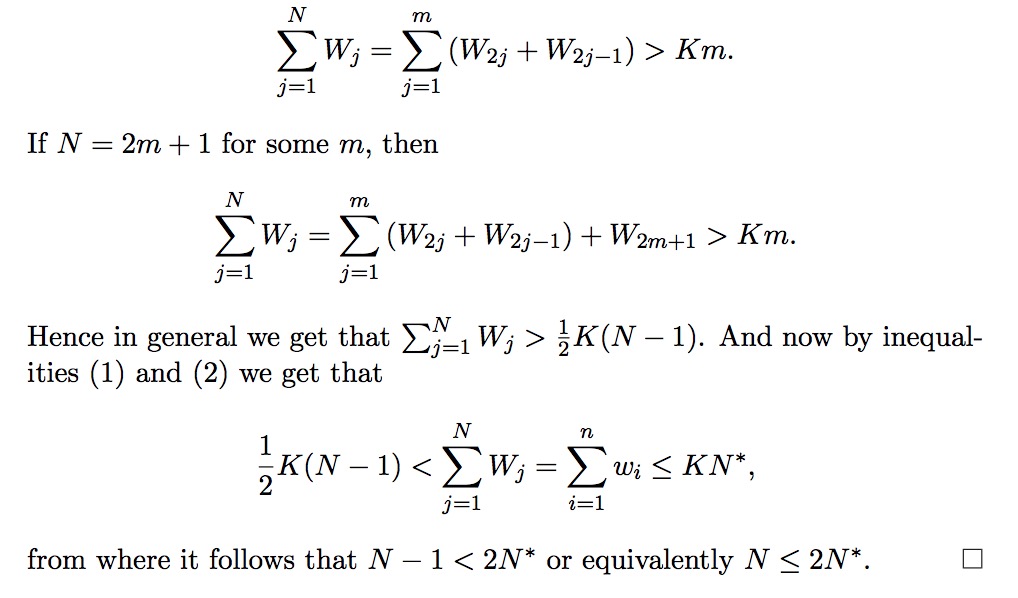
From the equation we can conclude that the algorithm would be true any j > 1.

Wj + Wj-1 > K

We also have the following equation



Since this is a 2-Approximation problem, consider N = 2m, Then, we get



Thus, we have proved that the given algorithm has an approximation ratio for at most 2 for the problem of minimizing the weight of the heavily loaded router.

1. (11 Points) Suppose a set S contains n two-dimensional points whose coordinates are all integers in the range [0, N]. What is the worst-case depth of a quadtree defined on S? Justify your answer.

**Solution:** A quadtree T can be defined by recursively performing a split at each child v of r if necessary.

Each and every child v of r gets a square region Ri which is associated with it we perform a checking if there exists region Ri for v which has more than one point of S.

If they contain, then we perform a split at v, which subdivides Ri in 4 equal-sized squares and we keep on repeating the subdivision process at v.

We can find the worst for the depth by following the steps below:

* Starting with the root r of T, & we are comparing region R for r to Z. If A and R, does not intersect at all, then we are finished. There are no points that exists in the subtree that is rooted at r that fall inside Z.
* Or else, if Z has R, then we can simply loop the external-node descendants of r.
* These have only two cases. If instead R and Z intersect, but Z does not completely have R, then we shall recursively perform this search on each and every child v of r.

The run time for quadtree T for S is **O(n)** for building each and every level of T.

So, the worst-case run would depend on the depth of its quadtree, Suppose k is the depth of the tree then the run time would be O(kn).

1. (11 Points). Consider a directed graph G that has n vertices, which is being updated by adding new edges to G. Now consider the transitive closure of G that needs to be updated anytime a new edge is added to G. Assume that the transitive closure is represented by an adjacency matrix. Show that the transitive closure can be updated in O(n2) when a new edge is added to G.

**Solution:** Consider A be a adjacency matrix which has dimensions [n x n] which represents the transitive closure, in such a way that the matrix X[i][j] == NULL if there exists no path between i to j and 1 if there exists is a path.

X[i][j] = 1

if i == j

else

NULL.

When consider a new edge - *(u, v)* which is added to the graph B, X would be updated using the following

Transitive-Closure-Update (u, v)

for i, j = 1 to n

If X[i][u] = 1 and X[v][j] = 1 do

X[i][j] = 1

Thus, adding an edge *(u, v)* would imply that there is a need to create a new path from every vertex v that could reach to u and from every vertex that could already be reached by v.

Performing such parsing would effectively mean that we could pass the entire adjacency matrix X to set X[u][v] to 1 instead of null, this would require nested loops which would effectively mean a runtime of **O(nˆ2)** since i, j run from 1 to n.