CSORW4231_001_2022_1 - ANALYSIS OF ALGORITHMS I

hw5

Benjamin Jenney

TOTAL POINTS

126 / 130

QUESTION 1

1 Problem 1 **28 / 30**

1 Is F-F O($$nm$$)?

QUESTION 2

2 Problem 2 **23 / 25**

QUESTION 3

3 Problem 3 **25 / 25**

QUESTION 4

4 Problem 4 **20 / 20**

QUESTION 5

5 Problem 5 **25 / 25**

QUESTION 6

6 Extra credit **5 / 5**

Analysis of Algorithms
HW 5

Benjamin Jenney, bej2117

May 2022

Problem 1

The situation pretty much maps over to a network flow with multiple sources and sinks and since we are only trying to find out if we can send all n people or not, this reduces into a decision problem.

Naturally if every injured person can be put up into a hospital then we can construct a flow in which each person can be put in a hospital. And if we can produce a maximum flow value equal to n then each injured person can be taken to a hospital.

Psuedocode: $A(G = (V, E), \mathit{capacities}, T)$

Input: an empty adjacency list representation of a graph $G = (V, E)$ of size $n + k$, and a set of times T where $T(u, v)$ corresponds to the time it would take person u to get to hospital v.
Output: The max flow on the graph G, the residual graph G_f, $True$ or $False$, depending on whether there exists a feasible flow.

```
1. Let $N$ be a representation of $\{u_1, u_2, ..., u_n\}$ vertices corresponding to each of the $n$ injured people, and $H$ be a representation of $v_1, v_2, ..., v_k$ vertices corresponding to k hospitals. Let $V = (N, H, s, t)$ where $s, t$ are a super source and super sink respectively. This can be done in $O(n)$ time if $n > k$ else $O(k)$.

2. for each $u \in V.N$, create an edge $(s, u)$: $G[s].append(u)$ with $c(s, u) = 1$. $O(n)$.

3. for each $v \in V.H$, create a $(v, t)$ edge: $G[v].append(t)$ edge with $c(v, t) = ceil(n/k)$. $O(k)$.

4. for each $u \in V.N$ and for each $v \in V.H$ if $T(u, v) \leq 30$ create a $(u, v)$ edge: $G[u].append(v)$ with $c(u, v) = 1$. $O(|V||E|)$

5. Run Ford-Fulkerson on $G$ returning the flow $f$. $O(|V||E|)$.
```

6. $if(|f| == n)$ add the edges between $V.N$ and $V.H$ to a list, R, each edge in this list represents a relation of each person to the hospital they can be sent to. $O(|E|)$.

7. return the list R, and the boolean value $if(|f| = n)$.

Correctness

Correcntess of Ford-Fulkerson (FF) has been proven in class, and slides. Thus if there is a feasible flow FF returns a max flow on G.

Claim: n injured people can be sent to k hospitals iff f is a feasible flow.

($\Rightarrow$) *Given that n injured people can all be put into k hospitals we have a feasible flow.* Since n people have ended up in k hospitals we will, by construction, end up with an s-t path for each person, in which each person u ended up in a hospital v. Since we have for each person that made it into a hospital a corresponding sum of capacities into $t = k * (n/k) = n$ and for each person that made it into a hospital a corresponding sum of capacities out of s = n, we have not exceeded capacity constraints, which is to say no hospital is over capacity and $n = |f|$ if we flood s.

($\Leftarrow$) *Given a feasible flow all n injured people can be put into k hospitals.* Given that f is a feasible flow, then by construction, we see that if we flood s the total flow out of s equals n, and the total flow into t = k*(n/k) = n. And since there exists a (u, v) edge for each person u and each hospital v with capacity one, then this flow corresponds to n people being sent to k hospitals. Therefore given a feasible flow, (since $c(s, v) = 1, (u, v) = 1$, and $(v, t) = ceil(n/k)$ for all u,v all edges are integral) we have it that all n injured people can be put into k hospitals, and $|f| = n$ if we flood s.

Furthemore, if it is a feasible flow then then R will contain person to hospital pairs for those who can be sent to a hospital.

Time

Clearly st[illegible] the time taken by 1,2,6,7 are dominated by steps 4 and 5. Step 5 adds a $\Theta(|V||E|)$ term in the worst case to the run time since we have to check each person against each hospital in order to establish which hospitals that person can make it to in the allotted time. And step 5 adds a $\Theta(|V||E|)$ term in the worst case. Taken all together the algorithm runs in $\Theta(|V||E|)$ time in the worst case.

1 Problem 1 **28 / 30**

1 Is F-F O($$nm$$)?

Problem 2

Input:
MAX-FLOW($G = (V, E), capacities$)
MIN-COST-FLOW($G' = (V', E'), capacities, cost$)

MAX-FLOW $\leq_p$ MIN-COST-FLOW

Transformation

G is a network with a feasible flow, we will construct a new graph G' that is an instance of MIN-COST-FLOW with a feasible flow.

1. $V' = V$
2. set *cost* to zero for all edges in G
3. Let $E' = E \cup (t, s)$ (add an edge from t to s) with infinite capacity and $cost(t, s) = -1$

We assume that G is a network with a feasible flow and integer capacities (by way of Integrality theorem). By construction G' has only integer costs. Besides the addition of a t to s edge with cost of -1. G' is identical to G.

If we feed G' to MAX-FLOW we produce a maximum flow for G. By construction G' maintains all the same s-t paths present in the original graph G. Therefore MAX-FLOW on G' will only augment on a subgraph of G' that is equivalent to G thus returning an equivalent max flow for G. Conversely passing G' to MIN-COST-FLOW will calculate the Max flow of G since it will augment on a subgraph of G' equivalent to G on a min cost path (by construction there is only one min cost path). Notice the min cost flow for G' equals $\Sigma_{e \in G'} f(e) * cost(e) = -|f|$ (since the cost of flow on one edge is the flow on the edge times the cost on that edge) and by construction that will be the case for any instance of MAX-FLOW. We also maintain that $f^{out}(v) = f^{in}(v)$ for all nodes. Thus G' is a feasible supply circulation.

2 Problem 2 23 / 25

Problem 3

A tells us if there is a vertex cover at most k in G, If there is then there is also a vertex cover in $G-\{u\}$ ($G-\{u\}$ denotes removing u and all it's incident edges) $A(G-\{u\}, k-1)$ will evaluate to true if u is part of a vertex cover, since it's removal reduced the size of the cover, and A will evaluate to true if and only if there is *at most a vertex of size* $k-1$. If u was not part of a vertex cover, $A(G-\{u\}, k-1)$ would evaluate to false because there would still be a vertex cover of at most k, not $k-1$.

Psuedocode
$binSearchForMinK(G=(V,E), p, k)$

```
k = k+1;
while (p ¡ k) do
    q = p + (k - p) / 2;
    if (A(G, q) == 'yes') then
        k = mid;
    end
    else
        p = mid + 1;
    end
end
return p;
```

Correctness
$binSearchForMinK$ is an iterative binary search that leverages A. A tells us that there is a cover of at most size k, which is to say there can be smaller vertex covers. here we check if we can cut down the search space and find the minimum vertex cover, If A evaluates to true we can look in the lower half of the search space only, if not then we look in the right half of the search space. Notice that the search space is 'sorted' in that the space grows monotonically on the range [p,k] i.e if k = 5, then there might be also be a cover of size k = 4, or k = 3..., correctness follows from correctness of binary search, and the runtime is $O(log(k))$, and for large ks this can greatly cut down the runtime of VC below.

$VC(G = (V, E), S, k)$ where S is an empty set.

```
if A(G,k) then
    k' = $binSearchForMinK(G = (V, E), 0, k)$;
    k = k';
    for each edge $(u, v)$ in E do
        if A(G-{u},k-1) == 'yes' then
            G = G-{u};
            // remove u and all edges incident to u
            S.append(u);
            k = k-1;
        end
        else if A(G-{v},k-1) == 'yes' then
            G = G-{v};
            // remove v and all edges incident to v
            S.append(v);
            k = k-1
        end
        if S.size == k' then
            return S;
        end
    end
end
return No;
```

Correctness
If G has a vertex cover of size k then it has a vertex cover of size k-1. Assume S is a vertex cover in G of size k, let u be some arbitrary node in the vertex cover, then S-{u} must cover all edges not incident to u. Therefore S-{u} is a vertex cover. Conversely if S is a vertex cover of size k-1 then G has a vertex cover of size k, if it did not then G never had a vertex cover of size k, but we assumed A was correct, contradiction.

Time

We will assume freeing memory for the nodes and edges in G can be performed in constant time. $binSearchForMinK$ adds only a $\Theta(log(k))$ term in the worst case. A has a polynomial runtime in the worst case, let $poly(G)$ denote this worst case time bound where $poly(G)$ is an anonymous function in the size of G. We make at most $|E|$ calls to A in the forloop, thus we have a run time of $\Theta(|E| * Poly(f(G)))$. Since a vertex can have at most $n - 1$ incident edges, and a vertex cover of at most size k covers at most $n - 1$ edges for each node, we have $\Theta(kn * poly(f(G)))$ in the worst case.

3 Problem 3 25 / 25

Problem 4

Inputs: MAX-SAT($\phi(n,m)$), SAT($\phi(n,m)$), MAX-SAT-Decider($\phi(n,k), t$)

Decision version of MAX-SAT

Max-Sat is a optimization problem that asks what truth assignment satisfies the most clauses. We can formulate a decision problem by asking if given an integer k can we find a truth assignment that satisfies at least k clauses.

Certificate $t =$ a truth assignment that satisfies at least k clauses.

We can then pass the certificate to MAX-SAT-Decider which in turn will assign each truth assignment in t to its corresponding variable in ϕ and then evaluate ϕ. Clearly t is linear in the size of ϕ, and assignments can be done in polynomial time (Something like: for each clause, c, in ϕ: for each variable x in c, $x = t(.)$). If ϕ evaluates to 1 we return 'yes' else we return 'no'.

Since MAX-SAT-Decider uses a short certificate, and decides for k clauses in ϕ in polynomial time, Max-Sat is in NP.

SAT $\leq_p$ MAX-SAT

Now I will show that SAT is polynomial reducible to MAX-SAT by showing that an input in for SAT can be expressed as an input of MAX-SAT-Decider.

Algorithm: let $\phi(n,m)$ be an input to SAT, let k = m, call MAX-SAT-Decider($\phi(n,k)$, t) if it outputs 'yes' $\phi(n,m)$ is satisfiable, if 'no' then $\phi(n,m)$ is not satisfiable.

I claim that $\phi(n,m)$ for SAT is satisfiable if and only if $\phi(n,k)$ is satisfiable. Equality is bi-directional; m=k, as such $\phi(n,m) = \phi(n,k)$. Which is to say, the input $\phi(n,m)$ for SAT is *exactly the same* as the input passed to MAX-SAT-DECIDER, as such an input to SAT is the same as an input to MAX-SAT. Therefore $\phi(n,m)$ for SAT is satisfiable if $\phi(n,k)$ is, and conversely $\phi(n,k)$ is satisfiable if $\phi(n,m)$ is, thus proving the claim.

I have shown that MAX-SAT $\in NP$, and is at least as hard as SAT, therefore MAX-SAT $\in NPC$.

4 Problem 4 **20 / 20**

Problem 5

Inputs: MAX-CLIQUE($G = (V, E)$), 3SAT($\phi(n, m)$), MAX-CLIQUE-DECIDER($G = (V, E), k, t$) where k is an integer and t is a certificate.

MAX-CLIQUE-DECIDER

MAX-CLIQUE is a problem that asks to find a clique of maximum size in $G = (V, E)$. We transform to a decision problem, MAX-CLIQUE-DECIDER by asking if there is a clique of at least size k in some arbitrary graph $G = (V, E)$.

Let certificate $t =$ A subset of vertices V' of V of at least size k.

We can then pass the certificate t to MAX-CLIQUE-DECIDER($G = (V, E), k, t$), which in turn will check for each vertex pair $u, v \in V'$ to see if there is a corresponding (u, v) edge in E for each pair, if so return 'yes', else return 'no'. Clearly the size of t is polynomial in V (it is a subset of V), and checking each pair is clearly a polynomial time operation in the size of the input (it can be done with some small number of loops over V' and E). As such t is a short certificate, and MAX-CLIQUE-DECIDER decides in polynomial time.

Therefore MAX-CLIQUE-DECIDER is in NP.

3SAT $\leq_p$ MAX-CLIQUE

Now I will show that 3SAT is polynomial reducible to MAX-CLIQUE by showing that any arbitrary input for 3SAT can be transformed into an instance of MAX-CLIQUE decidable by MAX-CLIQUE-DECIDER.

I will construct $G = (V, E)$ by way of gadgetry as follows:

Clause Constraint Gadget

> For each clause, up to k clauses, in $\phi(n, m)$, introduce a 3-tuple of nodes where each node is labeled by a variable in the current clause. Hence our graph consists of 3-tuples where each tuple represents the variables to a corresponding clause in $\phi(n, m)$. This maintains Clause Constraints since if ϕ was satisfiable then there would be at least one node in each 3-tuple with a label that corresponds to a true value.

Clause Consistency Gadget

> For each node in each 3-tuple create an edge to nodes in every other 3-tuple under the condition that no label $x_i \neq \neg x_i$ (that is if x_i appears in a node in one 3-tuple and it's negation in another, do not add an edge between them) this will ensure that we will not be adding edges that are inconsistent. Notice that we are not adding edges between nodes in the same 3-tuple since these would form triangles, i.e cliques, and would cause the decider to decide 'yes' for false ϕs. For instance an expression $\phi = (x_1 = 0, x_2 = 0, x_3 = 0)$ would be a triangle in G causing the decider to say "yes" for k=3.

Using these two gadgets, we have constructed a graph, $G = (V, E)$ from m clauses (we can always set k = m). Notice that *Clause Constraint Gadget* requires little work, and can easily be accomplished in $o(n^2)$ time, Similarly *Clause Consistency Gadget* can be done by checking each node, against every other node, adding an edge only if labels are consistent, and the nodes do not belong to the same clause (or 3-tuple). This can easily be accomplished in $o(n^3)$ time. Thus the construction G from $\phi(n, m)$ is a polynomial time operation.

ϕ is satisfiable if and only if $G = (V, E)$ has a clique of size k=m

assume ϕ has a satisfying assignment, then each clause in ϕ has at least one literal that evaluates to true, and each true literal will have a corresponding node in the 3-tuple corresponding to the clause. Adding these nodes that evaluate to true to V' we get a subset of V. For any pair $u, v \in V'$ where u, v do not belong to the same clause we have that they must have a value of true given by the satisfying assignment, as such these corresponding literals are not complements. And by construction any u, v pair in V' has an edge in E. Therefore they form a clique. Since u, v pairs in V' only have an edge if consistent, and V' contains only one node from each 3-tuple, we can assign a true value to each label, and assign a true value to its corresponding variable in ϕ. As such each clause will be satisfied and ϕ will be satisfied.

5 Problem 5 **25 / 25**

6 Extra credit **5 / 5**