Indian Institute of Technology Delhi

COL351 Analysis and Design of Algorithms: Assignment 4

Ishita Chaudhary: 2019CS10360

Mohammed Jawahar Nausheen: 2019CS10371

Contents

Flow and Min-cuts

Let G = (V, E) be a directed graph with source s and $\mathbb{T} = \{t_1, ..., t_k\} \subseteq$ V be a set of terminals. For any X $\subseteq$ E, let R(X) denote the vertices v $\in \mathbb{T}$ that remain reachable from s in G $-$ X and r(X) = |R(X)|
We are required to find X such that r(X) + |X| is minimum. Note that X is then a cut that partitions V into S, S' where s $\cup$ R(X) $\subseteq$ S and $\mathbb{T}$-R(X) $\subseteq$ S'.

We construct an auxiliary graph G' = (V',E') where V' = V $\cup$ t, E' = E $\cup\{(t_1, t), ..., (t_k, t))\}$. A flow network is defined on G' with every edge in E' having capacity 1, source as s and sink t.

Claim: A cut X in G can be mapped to an (s,t) cut in G' and vice-versa. Moreover, the capacity of corresponding (s,t) cut for X is r(X) + |X|.

Consider a cut X in G partitioning the graph into S, S'. Define X' = X $\cup\{(t_i, t)|t_i \in$ R(X)$\}$ that partitions V' into (A,A').

Claim: X' is an (s,t) cut.
To show that X' is an (s,t) cut it is sufficient to show that given s $\in$ A, t $\in$ A'. Assuming the contrary, t $\notin$ A' implies t is reachable from s in G - X'. Since t is only connected to terminals in $\mathbb{T}$, the path from s to t will use at least one edge of the kind (t_i, t).Also, since edges of the kind (t_i, t) can only be used to reach t, the reachability of all other vertices is same as in G - X , i.e, S $\subseteq$ A and S' $\subseteq$ A'. If $t_i \in S' \implies t_i \in A'$, s does not have a path to t_i. Otherwise if $t_i \in S \implies (t_i, t) \in X' \implies (t_i, t) \notin G - X'$. Thus t is not reachable from s in G-X', a contradiction. Also, capacity of (s,t) cut, X' from its definition is |X| + r(X) since (t_i, t) are outgoing edges from A.

Claim: A corresponding cut X can be found for every (s,t) cut X'.
Let $B = \{t_i|t_i \in$ A $\}$, then $\forall t_i \in B, (t_i, t) \in X'$ since X' is an (s,t) cut. Define X = X' - $\{(t_i, t)|t_i \in$ B $\}$. $\forall t_i$ not reachable from s, $t_i \in A'$ and $t \in A'$ thus $(t_i, t) \notin X'$. Therefore, X does not contain any edge of the kind (t_i, t). X partitions G into S,S' where S = A and S' = A'-t from the same argument used above.
Again, X = X' - $\{(t_i, t)|t_i \in$ A $\}$=X' - $\{(t_i, t)|t_i \in$S $\}$
= X' - $\{(t_i, t)|t_i$ is reachable from s $\}$ = X' - r(X).

Since we have successfully mapped every cut X to (s,t) cut X' of capacity r(X) + |X| and X' back to X what remains is minimizing r(X) + |X|, i.e, finding an (s,t) cut of minimum capacity. From the max-flow min-cut theorem, max-flow in a flow network is equal to the capacity of min-cut. Thus, we find the max-flow using the Ford-fulkerson algorithm in G', that also provides us with a min-cut X' in the end. Mapping back X' to X we attain the required X.

Note: In a flow-network we work with the assumption that s has only outgoing edges. This may not be true in G. However since reachability of a terminal from s is of consequence here and any simple path from s to t_i will not contain an edge (x,s), we can safely remove such edges from G.

Complexity analysis: We will first be analysing ford-fulkerson's algorithm. Constructing the edge list takes |E| time. Next BFS takes O(m+n) time and the loop runs until the max-flow is achieved. Since only integer increments are possible and the flow always increases the maximum iterations can be |f|. Thus, this is of O(|f|(m+n)).

```
Function FindX(G(V,E), T, s):
    V' := V ∪ t
    E' := E forall v ∈ T do
        E'.insert((v, t))
    end
    X := FordFulkerson(G'(V',E'), s, t) forall e ∈ X do
        if tail(e) = t then
            X.remove(e)
        end
    end
    return X
Function FordFulkerson(vertices V, edge list E, source s, sink t):
    initialise graph matrix R[V][V] //residual graph
    initialise ArrayBFS [V], AdjacencyList [V]
    initialise minCutSet
    forall edge ∈ E do
        R[edge[0]][edge[1]] = 1
        AdjacencyList [edge[0].add(edge[1])
        AdjacencyList [edge[1]].add(edge[0])
    end
    while BFS(R, s, t, ArrayBFS, AdjacencyList) do
        intialise pathFlow := +inf
        initialise vertex= t
        while vertex ≠ s do
            parent = ArrayBFS [vertex]
            pathFlow= min(R[parent][vertex], pathFlow)
            vertex = ArrayBFS [vertex]
        end
        vertex= t
        while vertex ≠ s do
            parent = ArrayBFS [vertex]
            R[parent][vertex]= R[parent][vertex]- pathFlow
            R[vertex][parent]= R[vertex][parent]+ pathFlow
        end
    end
    //flow is max now, call dfs to find reachable vertices
    initialise visitedArray [V]
    forall vertex ∈ V do
        visitedArray [vertex] = false
    end
    DFS (R, visitedArray, s, AdjacencyList)
    //add edges from reachable vertex to non-reachable vertex in G to minCutSet
    forall edge ∈ E do
        if visitedArray[edge[0]] = true&visitedArray[edge[1]] = false then
            minCutSet.add(edge)
        end
    end
    return minCutSet
```

```
Function DFS(graph matrix R[V][V], bool array visitedArray, source s,list
 AdjacencyList):
    visitedArray [s] = true
    forall vertex ∈ AdjacencyList[s] do
        if R[s][vertex] > 0 & visitedArray [vertex]= false then
            DFS (R, visitedArray, vertex)
        end
    end

Function BFS(graph matrix R[V][V], source s, sink t, vertex array ArrayBFS, list
 AdjacencyList):
    initialise visitedArray [V], queue
    forall vertex ∈ V do
        visitedArray [vertex] = false
    end
    visitedArray [s] = true
    ArrayBFS [s] = -1
    queue.push(s)
    while queue is not empty do
        firstVertex = queue.front()
        queue.pop()
        forall vertex ∈ AdjacencyList[firstVertex] do
            if visitedArray[vertex] = false&R[firstVertex][vertex] > 0 then
                visitedArray [vertex]= true
                queue.push(vertex)
                ArrayBFS [vertex]= firstVertex
            end
        end
    end
    if visitedArray[t] = true then
        return true
    end
    return false
```

Conversion of G to G' takes $|\mathbb{T}|$ time. Finding the max-flow and min-cut using the Ford-fulkerson algorithm takes O($|f|(m+n)$) time where f is the max-flow. Since the only edges that enter t are of the type (t_i, t) each with capacity 1, maximum flow that can flow into t is $|\mathbb{T}|$. X' can be converted back to X again in $|\mathbb{T}|$ time. Thus, the complexity of the algorithm is O($|\mathbb{T}||E|$)) .

Hitting Set

The Hitting-Set Problem (HS) for the input $(U, A_1, ..., A_m)$ is to decide if there exists a hitting-set $S \subseteq U$, of size at most k, i.e. S is such that $i \in [1, m], S \cap A_i \neq \phi$.

2.1 Prove that Hitting Set Problem is in NP Class

To show that HS $\in$ NP class of problems, it is sufficient to show a polynomial time algorithm that takes a set S along with $U, A_1, ...A_m$ as input and outputs a yes if it is a solution to HS($U, A_1, ...A_m, k$) and no otherwise. The algorithm `VerifyHS` is proposed for this.

```
Function VerifyHS(Universe U, list of A_i A[i],Set S,size-bound k):
    if |S| > k then return "NO"
    forall e ∈ S do
        if e ∉ U then
            return "NO"
        end
    end
    forall i := 1 to m do
        intersectionFound := false
        forall e ∈ S do
            if e ∈ A[i] then
                intersectionFound ← true
                break
            end
        end
        if not intersectionFound then return "NO"
    end
    return "YES"
```

Complexity Analysis for `VerifyHS`: The complexity of the algorithm depends on how we check if an element belongs to a set. Checking naively takes |S| time. This is the worst case and this is used for further analysis. Note that there are better ways of implementing sets to make this log|S| or even constant.

In the first block, thee for loops run |S| and |U| times, making the complexity O(|S||U|). In the next block of code, the outer for loop runs m times. For a given i, the inner for loop runs $|S|$ times. Checking if e belongs to A_i can take worst-case $|A_i|$ time. A total of $\sum_i |S||A_i|$, is the time-complexity of this block. The algorithm thus takes O(|S||U| + $\sum_i |S||A_i|$).

2.2 Prove that Hitting Set Problem is NP-Complete

To show HS $\in$ NP-Complete,it suffices to show a polynomial reduction of Vertex-Cover to HS since we know that Vertex-Cover $\in$ NP-Complete set of problems.

Consider an instance of the Vertex-Cover problem,(G, V, E, k). We derive an instance of HS as $U := V$, $k := k$ and $A_e := \{x, y\}$ where $e = (x, y) \in E$.

Claim: A set S is a solution to the Vertex-Cover problem iff it is a solution to the HS problem derived as above.

If S is a vertex-cover of G, then $\forall e = (x, y) \in E, x \in S \vee y \in S \implies \forall A_e = \{x, y\}, x \in S \vee y \in S$. Thus, $A_e \cap S \neq \phi$.
Similarly, if S is a HS(U, $A_1, ...A_{|E|}$), then $\forall A_e = \{x, y\}, A_e \cap S \neq \phi \implies x \in S \vee y \in S$. Since there is an A_e for every $e \in E$, S is a Vertex-Cover.

Complexity Analysis for reduction: Constructing a set for every edge e takes O($|E|$) time, which is polynomial.

Feedback Set

Consider the undirected graph G= (V, E). A feedback-set is a set X: X $\subseteq$ V and G - X has no cycle.

The Undirected Feedback Set Problem (UFS) has to decide if there exists a feedback set of size atmost k, given the graph G and k.

3.1 Prove that Undirected Feedback Set Problem is in NP Class

To prove that UFS is in NP class, the verifying algorithm for a solution must terminate in polynomial time. Given a graph's vertex set V and edge set E, Consider the set of vertices X. To verify if this set X is a feedback set of graph G= (V, E) with size atmost k, the algorithm will first check if the size of set X is $\leq$ k and if X $\subseteq$ V. Further, it will remove all the vertices which belong to X from V, and all corresponding edges from E. Now, it will check if the resulting graph has a cycle or not by calling DFS on every vertex.

```
Function DetectCycle(vertex set V, edge set E, feedback set X, integer k):
    if |X| > k then
        return false
    end
    forall vertex ∈ X do
        if vertex ∉ V then
            return false
        end
        V.remove(vertex)
    end
    forall edge(u, v) ∈ E do
        if u ∉ V or v ∉ V then
            E.remove((u,v))
        end
    end
    initialise visited := φ
    forall vertex ∈ V do
        visited.append(v, false)
    end
    forall i ∈ visited do
        if visited[i].value = false then
            if DFS(visited[i].key, visited, φ) then
                return false
            end
        end
    end
    return true
```

Time Complexity Analysis: Checking the size of set X takes constant time. To remove the vertices of X from V and the corresponding edges from E, the two loops iterate for O($|V|$) and O($|E|$) time. Removing takes constant time. Creating the visited array takes O($|V|$) time. The length of visited array is $|V|$. The for loop iterates over visited array and calls DFS on every vertex. Time complexity of DFS is O($|V| + |E|$). Hence, the time complexity of the algorithm

```
Function DFS(vertex v, key-value array visited, parent predecessor):
    visited [v].value = true
    adjcentVertices = adj[v]
    forall vertex ∈ adjacentVertices do
        if visited[vertex] = false then
            if DFS (vertex, visited, v) then
                return true
            end
        end
        else if vertex ≠ predecessor then return true
    end
    return false
```

is O($|V|(|V|+|E|)+2|V|+|E|$), i.e., O($|V|(|V|+|E|)$), which is a polynomial time complexity.

3.2 Prove that Undirected Feedback Set Problem is NP-Complete

To prove that Undirected Feedback Set Problem is NP-Complete, it is enough to show a polynomial reduction of Vertex-Cover to Undirected Feedback Set Problem, as we know Vertex-Cover belongs to NP-Complete set of problems.

Consider the graph G = (V, E) and create a graph H= (V+$V^{'}$, E+$E^{'}$) from G, i.e., all vertices of graph G belong in graph H, along with some additional vertices. The edge set $E^{'}$ is defined as: ∀ e= (v_1, v_2) ∈ E, ∃ a vertex $v_e \in V^{'}$: (v1, v_e), (v_2, v_e) ∈ $E^{'}$.

Claim: A set S is a solution set to the Vertex-Cover problem iff it is a solution to the Undirected Feedback Set Problem described above.
Proof: Proof by implication.
Forward Implication: Given the set S is a Vertex-Cover of G of size k, i.e., for each edge e= (v1, v2) ∈ E, the set S has atleast one of v1 and v2.
We can argue that the set S is a feedback set of graph H. Because, any cycle in graph H would be of form C= $v_a, v_b, ...v1, v2, ..., v_k, v_a$, i.e., it must contain the pair v1, v2, where e= (v1, v2) ∈ E, an edge in G. Since, S is a Vertex-Cover in G, S contains atleast one of v1, v2.
Hence, if we remove set S from graph H, it would contain no cycle. Thus, we can conclude that S is a feedback set of H, as S ⊆ V ⊆ V+$V^{'}$, and H-S has no cycles.

Backward Implication: Given set S is a solution to UFS Problem for graph H, i.e., S ⊆ V+$V^{'}$ and H-S has no cycle, S has size at most k.
We can argue that the set S need not contain any $v_e \in V^{'}$. Because, for an edge e= (v_1, v_2), H will contain a cycle of v_1, v_2, v_e, and H-S has no cycle. The feedback set S contains at least one of the three vertices.Note that the only edges of v_e are to v_1 and v_2 so a cycle containing v_e will surely contain both v_1 and v_2. So, even if v_e did exist in set S, it can be replaced by v_1 or v_2. Hence, for every edge e= (v_1, v_2), the set S must contain either v_1 or v_2.
Since, there is a v_e for every e=(v_1, v_2) ∈ E, the set S is a vertex cover in graph G.

Proved both the implications. Hence, proved.

Complexity Analysis of Reduction: For every edge e ∈ E, we build upon G by adding a vertex v_e and two edges incident on it as described above to create G'. This can be done by looping over the edges once. Thus, the reduction takes $|E|$ time which is polynomial in terms of input.