E0 225 Design & Analysis of Algorithms **Assignment - 4**
Sai Phanindra Korlepara — **24142** — M.Tech CSA Dept.

Problem 1. *JE Book, Chapter 11, Problem 9*
Suppose we are given an $n \times n$ square grid, some of whose squares are colored black and the rest white. Describe and analyze an algorithm to determine whether tokens can be placed on the grid so that

- *every token is on a white square*
- *every row of the grid contains exactly one token*
- *every column of the grid contains exactly one token*

Your input is a two dimensional array $IsWhite[1..n, 1..n]$ of booleans, indicating which squares are white. Your output is a single boolean. For example, given the grid in Figure 11.13 as input, your algorithm should return True.

Solution. This problem can be reduced down to a flow problem based on the row and columns of all the white squares, then we can apply orlin's max flow algortihm to quickly find if there is a solution such that there is exactly one cross per row and column in the white spaces of the matrix.

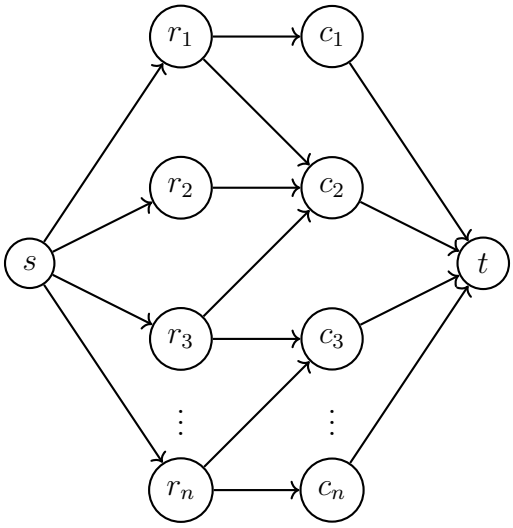

Figure 1: All the edges in the above graph have weight 1.

Algorithm

1. We start by creating a equivalent flow graph to the problem above. First create a source vertex s & target vertex t.
2. now create two sets of n vertices named as following $r_1, r_2, r_3.....r_n$ & $c_1, c_2, c_3.....c_n$ representing the rows and columns of the matrix. Create a directed edge from source s to each of the r_is & a directed edge from each of the c_is to t.

3. For each $IsWhite[i, j]$ that is true, place a directed edge from r_i to c_j.

4. Update all the edges to have a weight 1.

5. Now run any standard flow algorithm on the graph created above with source as s and sink as t.

6. If the max flow observed above is n, then there is a solution such that there is a way to place exactly one cross per each of the rows and columns, Hence return $True$. If the flow is less than n, return $False$.

Runtime Analysis

- Creating the vertices takes $O(n)$ time.
- Creating the edges and updating weights takes $O(n^2)$ time.
- Ford fulkerson runs in fV time where f is the max flow. In our case we cannot have a max flow greater than n, hence the runtime for ford fulkerson is $O(n^2)$.
- The overall runtime for this algorithm comes out to $O(n^2)$, which is same as the **Space complexity**, because we have to store the matrix of size $n \times n$.

□

Problem 2. *JE Book, Chapter 11, Problem 13*
The Department of Commuter Silence at Sham-Poobanana University has a flexible curriculum with a complex set of graduation requirements. The department offers n different courses, and there are m different requirements. Each requirement specifies a subset of the n courses and the number of courses that must be taken from that subset. The subsets for different requirements may overlap, but each course can be used to satisfy at most one requirement.

For example, suppose there are $n = 5$ courses A, B, C, D, E and $m = 2$ graduation requirements:

- *You must take at least 2 courses from the subset A, B, C .*
- *You must take at least 2 courses from the subset C, D, E.*

Then a student who has only taken courses B, C, D cannot graduate, but a student who has taken either A, B, C, D or B, C, D, E can graduate. Describe and analyze an algorithm to determine whether a given student can graduate. The input to your algorithm is the list of m requirements (each specifying a subset of the n courses and the number of courses that must be taken from that subset) and the list of courses the student has taken.

Solution. This problem similar to the one above and can be reduced to a flow problem based on courses, requirements. Instead of adding edges for all the courses from the source we will only add edges to courses that the student has taken.

Algorithm

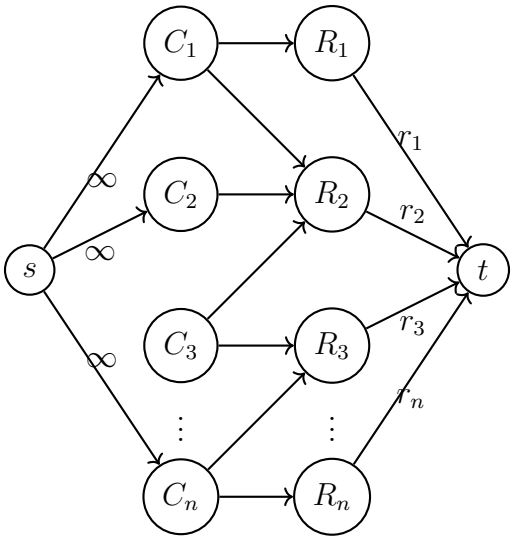

Figure 2: All the unmarked edges in the above graph have weight 1.

1. We start by creating a equivalent flow graph to the problem above. First create a source vertex s & target vertex t.
2. now create two sets of n vertices named as following $C_1, C_2, C_3.....C_n$ & $R_1, R_2, R_3.....R_n$ representing the courses that are available and different requirements of the degree.
3. Create a directed edge from source s to each of the C_is that a given student is taking& a directed edge from each of the R_is to t.
4. For each course, C_i that is acceptable as par of a given requirement say R_j, place a directed edge from C_i to R_j.
5. For each edge from R_j to t the weight is updated the number of courses to be completed from it's list of acceptable courses.
6. For each edge from s to C_t the weight is set to be ∞, because finishing a single course might contribute partly to mutliple requirements.
7. Update all the remaining edges to have a weight 1.
8. Now run any standard flow algorithm on the graph created above with source as s and sink as t.
9. If all the edges from R_j to t are saturated in the max flow, then we can be sure that all the requirements of the student have been met, If not the student has not taken all the requisite courses to satisfy the requirements.

Runtime Analysis

- Creating the vertices takes $O(n)$ time. Creating the edges and updating weights takes $O(n^2)$ time.

- The overall runtime for a flow algorithm is $O(VE)$, in a dense graph it comes out to $O(n^3)$.
- The **Space complexity** of this algorithm is $O(E)$ as the edges are the largest in size of all the things that need to be in the memory. In a dense graph it will be $O(n^2)$.

□

Problem 3. *JE Book, Chapter 12, Problem 5d.*
Suppose you are given a magic black box that can determine in polynomial time, given an arbitrary boolean formula , whether is satisfiable. Describe and analyze a polynomial-time algorithm that either computes a satisfying assignment for a given boolean formula or correctly reports that no such assignment exists, using the magic black box as a subroutine.

Solution. If there were a way to find if any given boolean formula is satisfiable or not, There is a way to find a solution that any given boolean formula will satisfy.

Any boolean formula is made up of literals say $a_1, a_2, a_3....a_n$. A boolean formula can be reorganised such that $Booleanformula = (a_1 \wedge B) \vee (a_1^{'} \wedge C) \vee D$, where B, C, D are made up of literals $a_2, a_3....a_n$.

Now if the original Boolean formula has a solution where a_1 is true, then $B \vee D$ must be satisfiable. similarly if the original Boolean formula has a solution where a_1 is false then $C \vee D$ must be satisfiable.

By verifying either of these two with the given black box, we can determine for any given literal a_i is true or false for a given satisfying solution.

We already know from class transformations of boolean formula can be done in polynomial time, by succesively identifying the truth values of all the individual literals and substituting them into the original boolean formula and simplifying it.

We will eventually find all the truth values of the individual literals that make the given Boolean formula satisfiable give that there exists a blackbox algorithm that predicts if any given Boolean formula is satisfiable □

Problem 4. *JE Book, Chapter 12, Problem 26.*
Let $G = (V, E)$ be a graph. A dominating set in G is a subset S of the vertices such that every vertex in G is either in S or adjacent to a vertex in S. The DominatingSet problem asks, given a graph G and an integer k as input, whether G contains a dominating set of size k. Prove that this problem is NP-hard.

Solution. To prove that the *DominatingSet* problem is NP-hard, we will perform a reduction from the well-known NP-hard problem $VertexCover$.

Definition of $VertexCover$:

Given a graph $G = (V, E)$ and an integer k, the $VertexCover$ problem asks whether there exists a subset $C \subseteq V$ of size at most k such that every edge in E has at least one endpoint in C.

Reduction Construction:

Given an instance (G, k) of $VertexCover$, we construct an instance (G', k') of $DominatingSet$ as follows:

1. For each edge $e = (u, v) \in E$, introduce a new vertex w_e.
2. The vertex set of G' is $V' = V \cup w_e \mid e \in E$.
3. The edge set of G' is $E' = E \cup (u, w_e), (v, w_e) \mid e = (u, v) \in E$.
4. Set $k' = k$.

Illustration of the Reduction:

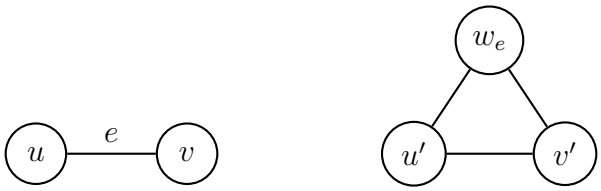

Figure 3: Transformation of edge $e = (u, v)$ in G to G'.

Correctness Proof:

We need to show that G has a vertex cover of size at most k if and only if G' has a dominating set of size at most k'.

($\Rightarrow$) If G has a vertex cover C of size at most k, then C is a dominating set of G' of size at most $k' = k$.

- Since C is a vertex cover in G, every edge $e = (u, v)$ has at least one endpoint in C.
- For each new vertex w_e, at least one of its neighbors (u or v) is in C. Therefore, all w_e vertices are adjacent to vertices in C.
- Any vertex $v \in V \setminus C$ is adjacent to some vertex in C. Thus, every vertex in G' is either in C or adjacent to a vertex in C, so C is a dominating set of G'.

($\Leftarrow$) If G' has a dominating set D of size at most $k' = k$, then D is a vertex cover of G of size at most k.

- We can assume that $D \subseteq V$, because including any w_e in D is not necessary; w_e only connects to u and v, and choosing u or v is more efficient.
- Since D is a dominating set in G', each w_e is either in D or adjacent to a vertex in D.
- For each w_e, at least one of its neighbors (u or v) must be in D to dominate w_e.
- Therefore, for every edge $e = (u, v)$, at least one of u or v is in D.
- Thus, D covers all edges in G, so D is a vertex cover of G of size at most k.

Conclusion:

Since we have a polynomial-time reduction from $VertexCover$ to $DominatingSet$, and $VertexCover$ is NP-hard, we can conclude that $DominatingSet$ is NP-hard.

□