Homework 5

CSOR W4231 - Analysis of Algorithms

Chandan Suri

UNI - CS4090

Collaborators: Arnavi Chedda (amc2476), Parth Jawale (pcj2105), Gursifath Bhasin (gb2760)

Submitted on -

May 04, 2022

Problem 1

Problem Description:
Input: I am given a set of patients (N), a set of hospitals (K) and a set of all the edges that are possible between the set of patients and the set of hospitals along with the driving times (in minutes) associated with the patient and the hospital.
Output: Returns the flow that consists of the matched patients and the hospitals if the crisis could be resolved. Otherwise, I return false signifying that the crisis couldn't be resolved.

Explanation of the Algorithm: As we are given the set of injured people (N), and the set of hospitals (K) along with the driving time for each patient to each of the hospitals as set of all the edges along with their associated driving times (T).
Following these, I create a bipartite graph at first as G which contains the vertices as patients and vertices as hospitals. In this bipartite graph, I sample only those edges which have weights (driving times) less than 30 minutes or half an hour. This new set of edges is denoted by E. Thus, this bipartite graph $G = (N \cup K, E)$ can contain at most $n * k$ edges, where n denotes the number of patients and k denotes the number of hospitals.
Next, we construct a flow network $G^{'} = (V^{'}, E^{'}, c, s, t)$ where $V^{'}$ is the set of all the vertices in the flow network, $E^{'}$ is the set of all the vertices in the flow network, c is the set of capacities for all the edges in the flow network. Also, s and t are the source and the sink nodes in the flow network formed. For reducing the bipartite matching problem of one to many, we reduce this problem to a flow one by following the steps as follows:

1. I introduce the nodes s and t (source and the sink node) where I connect the source to all the vertices for the patients and sink to all the vertices for the hospitals.

2. Basically, I set the set of vertices to include the source and sink nodes as $V^{'} = N \cup K \cup s \cup t$.

3. I also add the edges as stated above from the source and to the sink. Thus, the set of edges becomes: $E^{'} = E \cup (s, x) \forall x \in N \cup (y, t) \forall y \in K$. This makes the number of edges in the flow network to be: $|E^{'}| = |E| + n + k$.

4. Then, I set the capacity c for each edge in the graph to 1 when the $edge \in E'$ does not have an endpoint in the sink node (t).

5. Also, I set the capacity for all the edges in the flow graph equal to $\lceil \frac{n}{k} \rceil$. I do this step so, that at most a hospital can be loaded with $\frac{n}{k}$ number of patients.

Then, I just run the Ford-Fulkerson's algorithm for the max flow through which I get the matched edges in the max flow residual graph formed. This is the flow that gets returned which tells us the mapping between the patients and the hospitals. If the value of this flow is equal to the number of the patients, then the crisis can be resolved otherwise it cannot be.

Thus, I have solved the problem of managing the crisis.

Pseudocode of the Algorithm:

Algorithm 1: Algorithm to determine if all the patients can be matched with the hospitals thus managing the crisis.

```
Function manageCrisis(N, K, T = (u, v, t)):
    G = (V = N ∪ K, E = φ, c = φ);
    for (x, y, t) ∈ T do
        if t ≤ 30 then
            E ← E ∪ {x, y};
    end for
    V ← V ∪ {s} ∪ {t};
    for n ← N do
        E ← E ∪ (s, n);
    end for
    for k ← K do
        E ← E ∪ (k, t);
    end for
    for e = (x, y) ∈ E do
        if y ≠ t then
            c_e = 1;
        else
            c_e = ⌈n/k⌉;
    end for
    f ← FordFulkerson(G = (V, E, c), s, t);
    if |f| == length(N) then
        return (f, True);
    return False;
```

Correctness of the Algorithm: Firstly, we claim that the size of maximum matching in G equals the maximum flow that we get after running the "Ford-Fulkerson's algorithm". Essentially, the edges of the matching are the edges that carry flow from N to K in $G^{'}$.

For proving the equivalence of the instances, we need to first prove the forward direction of the reduction problem. As in the bipartite graph G, when we have all the mappings from the set of patients to the set of hospitals. The flows for these mappings will be maxed out to be equal to 1 that is the capacity of all these edges which basically tells us if that patient can be mapped to the associated hospital or not. So, if there are n patients mapped to k hospitals, that would mean that in the bipartite graph we have n mappings. Following this,

the maximum flow coming out of the source will be equal to n as there are n patients that need to be mapped to resolve the crisis.
Corresponding to these maximum flows, there will be n mappings between the patients and the hospitals. These number of mappings are restricted to be n by the flow going into the sink node. As the maximum allowed capacity out of each of the hospitals is $\lceil \frac{n}{k} \rceil$. Thus, the maximum flow that go into the sink combined for all the hospitals will be $\frac{n}{k} * k = n$. Therefore, by the law of flow conservation, the flow going out of the source equals the flow going into the sink which is bounded by n (the number of patients). When we combine this fact with the n edges between the source and the patients nodes each having a capacity of 1, this means that we can get at most n edges mapped from the patients nodes to the hospitals nodes in the flow network when we run the max flow algorithm (Ford-Fulkerson's algorithm). This proves the equivalence in the forward direction.

For proving this in the backward direction: If we have n edges mapped from the patients nodes to the hospital nodes in the flow network (part of the max flow computed). Then, the mapped edges have the flow equal to 1 and is maxed out to be n edges by the law of flow conservation. Thus, the mapping between the patients and hospitals will contain only one edge coming out of each of the patient nodes which means that this mapping can be one to many from the patients nodes to the hospital nodes. If we simply remove the edges from the source node to the patient nodes and the edges from all the hospital nodes to the sink node, then we basically get the same mapping from the set of patient nodes to the set of hospital nodes that would resemble the mapping in the bipartite graph. As we can get the same mapping of edges from both the directions through performing the Ford-Fulkerson's algorithm on the flow network formed, thus, we have successfully proved the equivalence of the instances.

Also, as this is an integral flow where the law of flow conservation is applicable, the flow going out of the source node is equal to the number of patients (n) when there is a valid mapping available (with capacities equal to 1 for all the edges from the source node to the patient nodes). This, in turn, is equal to the flow to the sink node (when we get the max flow) which can be maxed out as $\frac{n}{k} * k = n$ where k is the number of hospitals. Thus, each node in N is the tail of at most one edge in the mapping formed (from source to the patients). Also, each node in K is the head of at least one edge in the mapping formed (from hospitals to the sink node) after the maximum flow algorithm has been performed. This is taken care of the fact that the maximum flow from any hospital node to sink node can be $\lceil \frac{n}{k} \rceil$. Thus, this maximum flow implementation on the flow network formed gives us a matching/mapping between the patient nodes and the hospital nodes in the bipartite graph.

Also, by the law of flow conservation the flow out of the source node can be at most n. and the flow to the sink node can be at most $\lceil \frac{n}{k} \rceil * k = n$. This tells us that the cut formed between the patient nodes and the hospital nodes can have the net flow equal to n (number

of patients). Thus, for a valid mapping, we will always get n edges that are included in the max flow between the patient nodes and the hospital nodes.
This completely proves the correctness and/or validity of the algorithm above.

Time Complexity Analysis: Lines 2 is a constant time $O(1)$ operation since it is simply an initialization step in which they take the union of the 2 sets of vertices and initialize the set of edges as empty and set of capacities for those edges as empty.
Lines 3-6 runs in nk time where n is the number of patients and k is the number of hospitals. Since, every injured person could potentially be mapped to a hospital nearby (within driving time of 30 minutes), so, we would potentially have to go over all the nk edges possible. Thus, this for-loop takes $O(nk)$ time.
Line 7 is a constant time operation since we just add a source and sink node to the set of vertices V.
The for-loop on line 8 would take (lines 8-10) $O(n)$ time as we are adding an edge from the source node to all the patient nodes in the bipartite graph. Similarly, the for-loop would take (lines 11-13) $O(k)$ time as we are adding an edge from all the hospital nodes to the sink node.
Then, we are just going over the set of all the edges (lines 14-19) in the flow network formed to add the capacities for each of the edges in the flow network. For any node not ending in the sink node, we set the capacity to be 1 (a constant time operation), otherwise we set it to ceil of $\frac{n}{k}$ (a constant time operation). As we are going over all the edges in the flow network, this would run $nk + n + k$ times which gives us the worst-case time complexity of $O(nk)$.
Line 20 calls the Ford-Fulkerson's algorithm which takes $O(nmU)$ time. The number of edges in the graph denoted by m equals nk in the worst case, while U will be equal to $\lceil \frac{n}{k} \rceil$. Thus, the Ford-Fulkerson's algorithm takes $O(n \cdot nk \cdot \lceil \frac{n}{k} \rceil) = O(n^3)$ time.
Lastly, lines 21-23 are constant time operations since they are conditional and/or return statements.
Combining all the analysis above, the maximum running time is for the Ford-Fulkerson's algorithm for finding the max flow which makes the worst-case time complexity of the algorithm equal to $O(n^3)$.

Solution 2

Input: A flow network with supplies where each edge $(i, j) \in E$ also has a cost a_{ij} , that is, sending one unit of flow on edge (i, j) costs a_{ij}.

Output: A feasible flow $f : E \to R+$, that is, a flow satisfying edge capacity and node supply constraints, that minimizes the total cost of the flow.

To Prove: The max flow problem can be formulated as a min-cost flow problem. '

Formulation: If we need to convert or formulate the max flow problem as a min-cost flow problem, we need a logical mapping between the two. So, for a max flow problem, basically, we try to maximize the flow through the network considering some capacity constraints. However, for a min-cost flow problem, we try to minimize the total $cost * flow$ across the whole network. The flow with consideration of costs can be shown as follows (optimization problem for the min-cost flow problem):
Minimize :

$$\sum_{e \in A} c(e) f(e)$$

such that,

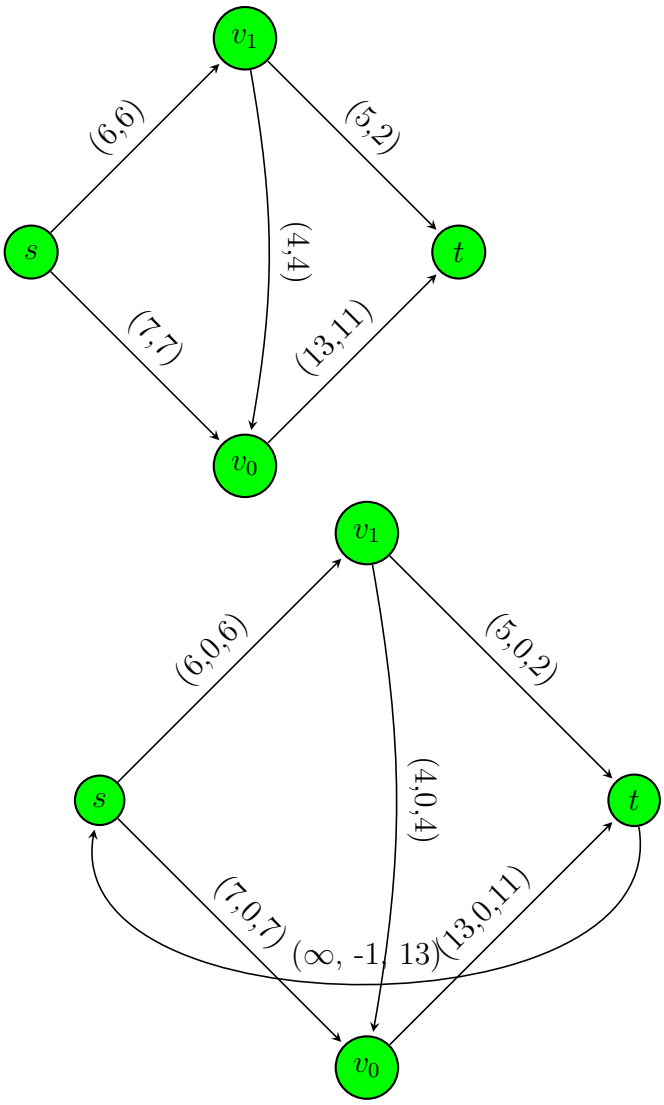
$$\sum_{e \in N^+(v)} f(e) - \sum_{e \in N^-(v)} f(e) = b(v) \quad \forall\, v \in V$$

, where N+ denotes the edges that give supply to the set of vertices and N- denotes the edges in the neighborhood of v that send out the supply to other set of vertices. Thus, this becomes the net supply/demand depending on the sign for the vertices. And,

$$l(e) \leqslant f(e) \leqslant u(e) \quad \forall\, e \in E$$

where l is the lower bound of the flow, u is the upper bound of the flow and f is the actual flow through a ny edge in the network.

This is quite synonymous with the shortest path problem. Thus, instead of sending an unbounded amount of flow along the path of minimum cost, we wish to send a maximum amount of flow from a specified source to a specified sink node through the network regardless of the cost.
Formally, the feasible solution for the maximum flow problem is that we need to send a maximum amount of flow from a specified source to a specified sink node, along with the fact that there is a maximum bound on the amount that we can send through the edges in the graph which is represented by the capacities of those edges. Thus, we can formulate the min-cost flow problem as follows:
We set the supply/demand ($b(v)$) to zero $\forall v \in V$, and cost for all edges in the graph equal to zero as well for all the edges in the graph. We do this while keeping the maximum bound intact. Following this, we introduce an arc from the sink node to the source node (t, s) with a cost of -1 and flow upper bound to be infinity. This is done this way because following



all these steps, minimum cost flow maximizes the flow on the arc from sink to the source node, but since any flow on this arc must travel through the source to the sink node path (including other edges in the network), we will end up maximizing the flow from the source node to the sink node in the original network.
The negative cost on the arc added from the sink to the source node helps changing the maximization function for the flow to a minimization one. Also, as we need to maximize the flow as much as possible which is unknown, we set the upper bound of the flow for the new arc added to be infinity. As the objective function changes for this, while trying to maximize the flow, we basically end up minimizing the cost as much as possible. Lastly, this formalization has been illustrated in the figures clearly. (the top one is the original network and the bottom one is the transformed graph for the minimum cost flow network)

Proof of Equivalence: This is more of a reduction problem, so, to complete the formula-

tion, we would need to prove the equivalence in both the directions.
Forward direction: When we have some max flow network and we know what the path of max flow would be. Then, if we convert this graph to a minimum cost flow graph, as the flow for the newly added arc from the sink node to the source node is infinity, it will let through all the possible flow but as the same flow has to pass through the path from the source to the sink node too, this will be upper bounded by those arcs too and thus, would be upper bounded in a similar way as for the max flow path. Additionally, as we are trying to minimize the cost here, when the flow is some value less than the upper bound, then the only way to minimize the $cost * flow$ value is to consider cost to be zero or very near to zero. Thus, we do the same here. Cost can only be greater than zero for a feasible and optimal solution when the flow is zero for that edge. All this implies the fact that we would get a similar mapping in the transformed graph. Thus, these 2 graphs are equivalent in the forward direction.

Reverse direction: When we have the transformed graph for the minimum cost flow network with the added arc and the costs equal to zero for all the edges in the network, that has a positive flow less than the upper bound. This is an optimal solution based on the fact that a feasible flow f is optimal iff. there exists a potential or cost such that for all the edges:

1. If the cost is greater than zero, then the flow must be zero.
2. If the flow is positive but less than the capacity or the upper bound of the flow for an edge, the cost must be equal to zero.
3. If the cost is less than zero, then the flow for that edge must be equal to the upper bound of the flow through that edge (aka capacity).

So, if we have an optimal flow for such a network, that would mean that this would be analogous to the maximum flow through the network which would be the similar path that we get from the max flow network.
As both of these are equivalent in both the directions, thus, we can say that we have successfully formulated the max flow problem as the min cost flow problem.
Hence, Proved.

Problem 3

Problem Description:
Input: We are given an input graph G = (V, E), an integer k, and a polynomial-time algorithm A that is used to find whether a vertex cover exists of size at most k (Returns "yes" when vertex cover is there otherwise "no".
Output: Returns the independent set S that is the minimum vertex cover for the graph along with "Yes" if vertex cover of at most k could be found. Otherwise, it just returns "no".

Solution:
Description of the Algorithm:
For this problem, we have already been provided an algorithm A that tells us for a given number of vertices, whether there is a set of vertices in the graph G that can cover all the edges (vertex cover is possible or not) with at most k vertices.
So, now we need to find the minimum number of vertices given k (less than or equal to k) for which the vertex cover was possible. Thus, I run a linear search from number of vertices as one till k. As soon as we get a vertex cover with some number of vertices, I end the loop and return that value. As we are running it from the lowest possible number of vertices, we would return the minimum set of vertices that can cover all the edges in the graph. If we can cover all the edges with that number of vertices (at that point in time in the loop), then we store it as min_k and return the same. This is done by calling the algorithm A to find out whether we can find a vertex cover with $start\_idx$ number of vertices. I run the loop till we have reached k. In case we don't get the case when we can find the vertex cover with some number of vertices less than or equal to k, then we return -1 which signifies that we cannot form vertex cover with $\leq k$ vertices. For this, we just return "no".
Otherwise, if we indeed find minimum value possible min_k for solving vertex cover problem, we sample those number of vertices from the set of vertices in the original graph and create a list of covered edges with that min_k number of vertices. If the number of the covered edges is equal to the number of edges in the original graph, then, we have got the independent set that was the current set of vertices that solves the vertex cover problem. Thus, the candidate solution S becomes the independent set which we return at the end along with the "yes".
Thus, this algorithm gives us the minimum vertex cover with at most k number of vertices if possible. Otherwise, it just returns "no".

Pseudocode:

Algorithm 2: Returns the vertex cover by finding the minimum vertex cover size.

```
1  Function FindMinK(G = (V, E), k):
2      start_idx ← 1;
3      end_idx ← k;
4      min_k ← −1;
5      while start_idx ≤ end_idx do
6          is_vertex_cover = A(G, start_idx);
7          if is_vertex_cover == "yes" then
8              min_k = start_idx;
9              break;
10         start_idx = start_idx + 1;
11     end while
12     return min_k
13 Function FindVertexCover(G = (V, E), k):
14     min_k = FindMinK(G, k);
15     total_num_edges = |E|;
16     if min_k == −1 then
17         return "no";
18     for S ∈ V when |S| = min_k do
19         covered_edges = φ;
20         for v ∈ S do
21             covered_edges = covered_edges ∪ E_v;
22         end for
23         if |covered_edges| == total_num_edges then
24             return {"yes", S};
25     end for
26     return "no";
```

Correctness of the Algorithm:
As we are using Linear search here for the first part, this should work correctly as taught in the class. Also, as the algorithm A has been given to us, we can assume that the algorithm A works correctly and returns yes when the vertex cover can be found with that corresponding number of vertices in the graph. If that works correctly, then we are just linearly searching through the list of number of vertices and we return the control as soon as we get the vertex cover to be true for the first time. This would return us the number of vertices in the minimum vertex cover because we are running the algorithm linearly from the value of 1 and as soon as the condition for checking the vertex cover becomes true, we break from the loop. Thus, this will always be the minimum one.
Following the first part of the correctness above, once we have got the minimum value of k

for which we get vertex cover, we start sampling min_k number of vertices out of the total vertices in the original graph. As min_k is a valid number of vertices that has a valid vertex cover, we will always get a subset of min_k number of vertices that covers all the edges in the original graph. Also, since choosing k vertices out of a set of n vertices is basically an exhaustive search over all the possible combinations, we are bound to get the set of vertices that form the vertex cover. Thus, this would always result in a valid and correct solution.
Therefore, we have proved the correctness of the algorithm.

Time Complexity Analysis: For the $FindMinK$ function: Lines 2-4 are just constant time operations ($O(1)$) as these are just assignment operations.
Then line 5 has a while loop that runs k times and all the lines inside the loop from line 7 to line 11 are just conditional, assignment, and break statements which are constant time operations but the line 6 is the call to the algorithm A. Line 6 runs in polynomial time.
Thus, this full while loop runs in polynomial worst-case time complexity. Thus, this function also runs in polynomial worst-case time complexity.
For the $FindVertexCover$ function: Line 14 is just a call to the $FindMinK$ function that runs in polynomial worst-case time.
Line 15 is just an assignment statement in which we get the total number of edges which is a contant time operation.
Then, lines 16-17 is just a conditional statement and a return statement which runs in constant time.
Next, line 18 is a for loop which runs for $\binom{n}{k}$ times. Also, line 20 is a for loop which runs for all the vertices in the candidate set S. As this can at max contain n vertices, thus, this can run n times. Other statements inside these for loops are just conditional, assignment and return statement which are all constant time operations. Thus, this complete looping from line 18-25 takes $O(n * \binom{n}{k}) = O(n * n^k) = O(n^k)$ time.
Thus, this complete algorithm is a polynomial time algorithm with the worst-case time complexity of $O(n^k)$.

Problem 4

Given/Input:
The input is a SAT formula ϕ with m clauses and n variables. Output for the same is the truth value for that formula when we can satisfy maximum number of clauses.

Decision Version:
The SAT problem in general is the satisfiability problem where we need to satisfy some clauses in the SAT formula provided with some m clauses. When we need to satisfy the maximum number of clauses along with the maximum variable assignments in our independent set, this problem becomes a max SAT problem. Thus, the decision version of the problem becomes:
For a given SAT formula, can we find the maximum number of assignments of the variables that can help satisfy all the clauses of the SAT formula such that the number of assignments required (not arbitrary ones) is denoted by k. ($k \leq m$)

Proof for NP:
Now, to prove that the decision problem formulated above is NP-complete, we first need to prove that the problem is NP. Also, to prove the NP-completeness, we will need to make it a reduction problem to the independent sets.
Let's take a SAT formula (which is in CNF form) to start this proof as :

$$\phi = (a \vee b \vee c \vee d \vee e) \wedge (\overline{c} \vee g \vee h) \wedge (\overline{b} \vee j)$$

This formula can be represented as a graph below that consider taking only one of variables true when we have two variables as complements of each other. The SAT formula above has $m = 3$ clauses and $n = 10$ variables (including the complemented ones).
We need to find independent sets such that the whole statement becomes true when each clause is true so, at least one variable/vertex should hold a true value. Thus, we need to find k variables that we can assign some value such that we get truth assignment for the whole statement to be true (meaning that we can satisfy all the clauses in the statement). This becomes a combinatorics problem of choosing k variables out of the n variables present that can maximally satisfy the whole statement. The time complexity for this problem, given a value for k, will be:

$$\binom{n}{k} = O(\binom{n}{k}) = O(n^k)$$

The above algorithm has to hold such that we choose at least 1 variable out of each of the clauses to be true. As we can see above, due to the algorithm being in polynomial time and as it is non-deterministic, we can conclusively say that this decision problem is NP.

Reduction Explanation:
For the SAT formula, we need to convert it to a graph that is basically a consistency gadget

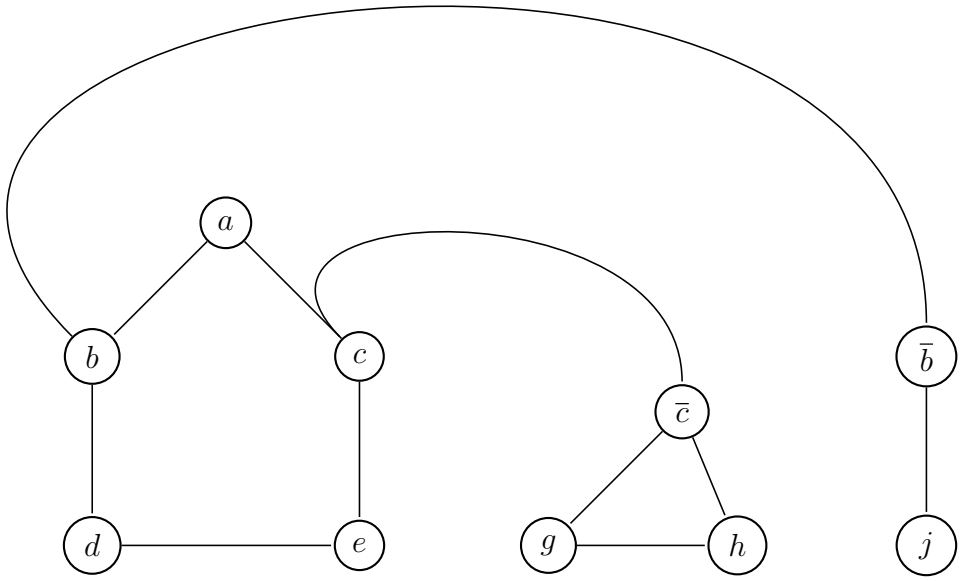


Figure 1: Graph G showing the formulation in CNF

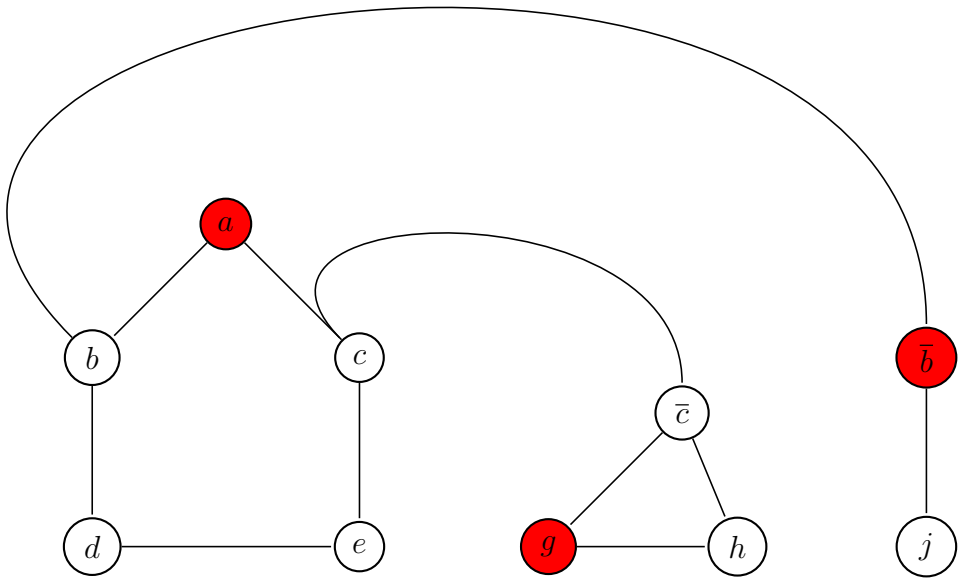


Figure 2: Graph G showing the formulation in CNF with independent set

for us. Firstly, we form a connected graph for each of the clauses in our statement. For instance, for the SAT formula given above, we have formed a pentagon for the 1st clauses, a triangle for the second one and a line for the third one. Additionally, for the truth assignment to be consistent, we form the edges between the complemented variables such that only one of them can be true at a time.
This reduction step basically requires polynomial time as well. This reduction is also illustrated in the figure 1 that shows all the variables joined by edges which signifies the clauses and the consistency of the truth assignment. Furthermore, this reduced graph is passed to the algorithm for finding independent sets that would signify that we can get some variable assignments that give us a truth assignment satisfying all the clauses.

Equivalence of Instances - Proof:
Now, we need to prove that ϕ is satisfiable if and only if G has an independent set of at least k size and at most m ($k \leq m$). Firstly, let's prove this in the reverse direction:
Suppose, we have an independent set of size k. Then, every shape (connected components for each of the clauses) would contribute at least one node/variable to the independent set S. So, we will set the corresponding literals for the nodes to 1 and will set all the other variables arbitrarily (0 or 1) if it's not a complement of some variable that was already chosen to be true. If the complement if a variable was chosen to be 1, we will have to set that to be 0 and vice-versa.
Next, we need to show that the truth assignment is:

1. is valid: Some variable x and it's complement cannot both appear in the independent set S, so, this will be consistent and is thus, valid. This is enforced by constructing reduced graph (through reduction step).

2. satisfies ϕ: Since, every shape (connected components for each of the clauses in the graph constructed) contributes at least one node to the independent set, this would mean that each clause in the SAT formula will be satisfied.

For instance, an independent set for our example of the SAT formula could be written as $S = (a, g, \overline{b})$. Thus, the derived truth assignment will be $a = 1, g = 1, \overline{b} = 1$. This has also been illustrated in the figure.
Thus, we have proven the equivalence in the reverse direction.

Next, we need to prove the equivalence in the forward direction:
Now, we first suppose that there is a satisfying truth assignment for ϕ. Then, we know that there will at least be one true literal in every clause. Next, to construct an independent set S, we will take only those literals that are true in the truth assignment. For instance, in our example shown, I can take an independent set to be $S = (a, g, \overline{b})$ (of at least size k). We claim that the S constructed is actually the independent set. And this is indeed the independent set because:

1. S would not be an independent set if there are any variables or corresponding nodes in the independent set that has edges between them. As we can see that this won't be the case here as we are only considering 1 variable from each of the clauses, this would mean that no edges will be there between those nodes.

2. Also, as S contains only true literals, thus, by construction we know that we can't have both a variable and it's complement at the same time in the independent set.

Thus, the S that we are getting is actually the independent set. This proves the equivalence in both the directions.

Conclusion:
Thus, this problem was NP and after proving the equivalence of instances, we know that the decision version of the "max SAT" problem is an NP-complete problem. This is perfectly illustrated with the help of the figures.

Problem 5

Given/Input:
We are given a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges in the original graph.

Decision Version:
Since, we need to convert the problem of finding the clique in the graph G of maximum size, we need to find that maximum number of vertices which forms the clique. Thus, we would first need to perform a linear search from $|V|$ to 0 to find the k that would the maximum number of vertices that forms a clique.
Thus, the decision problem of "Max Clique" becomes:
If we are given a graph $G = (V, E)$ a target value k, does the graph G have a clique (a complete sub-graph which will be a dense graph) on at least k vertices?

Proof for NP:
Now, to prove that this decision problem is NP-complete, we first need to prove that this problem is NP. To prove the same, we need a certifier and a certificate.
The certificate for the same is:
$S = \{v_1, v_2, ..., v_k\}$ where S is the subset of vertices such that all possible edges between the vertices in S appear in E (all the edges in the graph G).
The certifier can be formulated as follows:
$\forall u \in S$ and $\forall v \in S$, there $\exists (u, v) \in E$. This gives us a candidate solution S. Thus, given an instance G, k and a candidate solution S, we can confirm that G, k is a "yes" instance of finding the Clique, i.e., G has a clique on k vertices. For this candidate solution, firstly, we need to check whether S has k vertices. Then, for every pair of vertices in S, we need to confirm that there is an edge present between them. Thus, if there are v vertices in the graph, this will take $O(v * (v-1)/2) = O(v^2)$ time. Thus, when we perform this algorithm, this takes a polynomial amount of time. As this would give us different solutions because we can have many candidate solutions or independent sets for the same (non-deterministic) and the time complexity comes out to be some polynomial time. Thus, using the certifier and the certificate, we have proven that this is an NP problem.

Reduction Explanation:
For the reduction, we basically form a complemented version of the original graph G as G'. In the graph G', we will have only those edges which weren't present in the original graph G. Thus, if we are given an instance $(G = (V, E), k)$ from the independent set, we will construct an instance $(G' = (V, E'), k)$ where E' precisely contains only those edges that do not appear in G. This transformation from G to G' requires going over all the sets of vertices and then checking if that edge is present in the original set of edges. Such a transformation from G to G' happens as shown below in the figure 3 to figure 4. As we can see in the figures below, the graph G in the figure 3 has some edges missing between the nodes b and c, and b and d.

These edges are the only ones present in the graph G' after the reduction. If it is not present, then we add that edge in the corresponding place in the G' graph. Thus, this will also take polynomial time.

Equivalence of Instances - Proof:
Thus, for an independent set S in G that forms a clique is clearly visible in G'. As we can see in the transformation in the figures 5 and 6, when there is a clique present, then the corresponding graph G' does not have any edges present for that clique. The independent set here consists of $k = 5$ vertices. For proving the equivalence of instances:

1. When some edges are not present between any of the set of vertices in the graph G that forms a clique, then those edges will be present in the graph G' as we are taking only those edges in the graph G'. Thus, such a reduction will always help us give the answer "yes" or "no" depending on the target variable k passed. For the figure 4 and 5 shown, we can see that when we pass k = 5 and the graph G, then none of the edges are present in the graph G' signifying that the max clique size is 5 (k) here. This will always occur after the reduction step. Thus, the algorithm for finding the clique will always result in a "yes" or "no" when this reduced graph is passed as an input to the algorithm after the reduction.

2. For proving the same equivalence in the reverse direction, if some edges are present in the graph G', then that means that those edges weren't present in the original graph G. As shown in the figure 5 and 6, no edges were present in the graph G' as all the possible edges were present in the original graph G which actually formed a clique of size 5. Here all the vertices will be the part of the independent set S. As shown in the figures 3 and 4, if some edges were shown in the graph G', then those edges were the only ones missing in the original graph G. Thus, the algorithm for finding the clique with target size of less than equal to 2 will result in a "yes" as we will able to find a clique. For the maximum value of k for which we can still find a clique in the modified graph G', this algorithm will result in a "yes" and thus, that k will be the size of maximum clique. According to the figures 3 and 4, that will be 3 and S will be as $S = \{a, e\}$. Thus, the graph G' will always give us the edges not present in the graph G thus, resulting in the similar type of mapping that helps us find the clique.

Thus, we have successfully proven the equivalence of instances through the illustration and the explanation.

Conclusion:
Thus, this problem was NP and after proving the equivalence of instances, we know that the decision version of the "Max Clique" problem is a NP-complete problem. This is perfectly illustrated with the help of the figures shown.

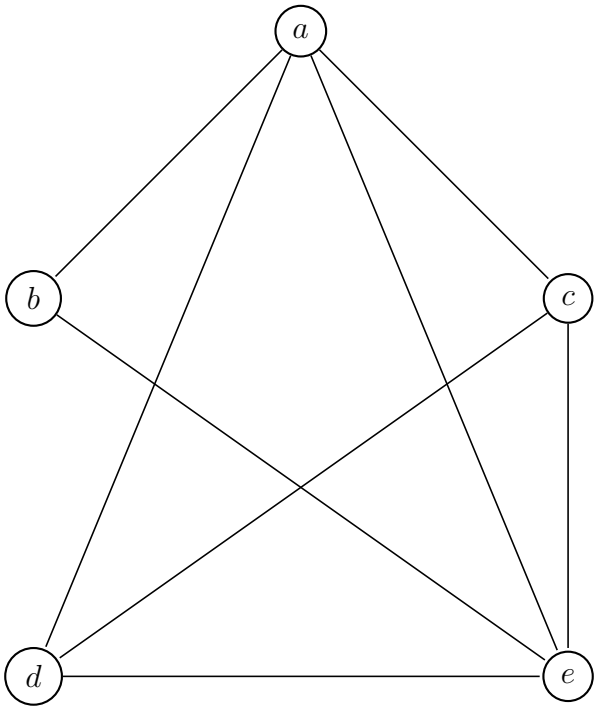


Figure 3: Graph G with some edges missing

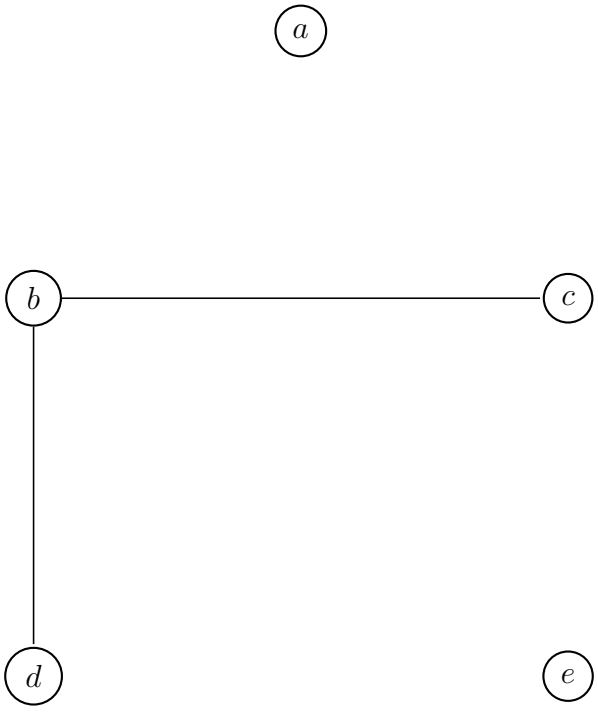


Figure 4: Graph $G^{'}$ consisting of edges not present in Figure 2

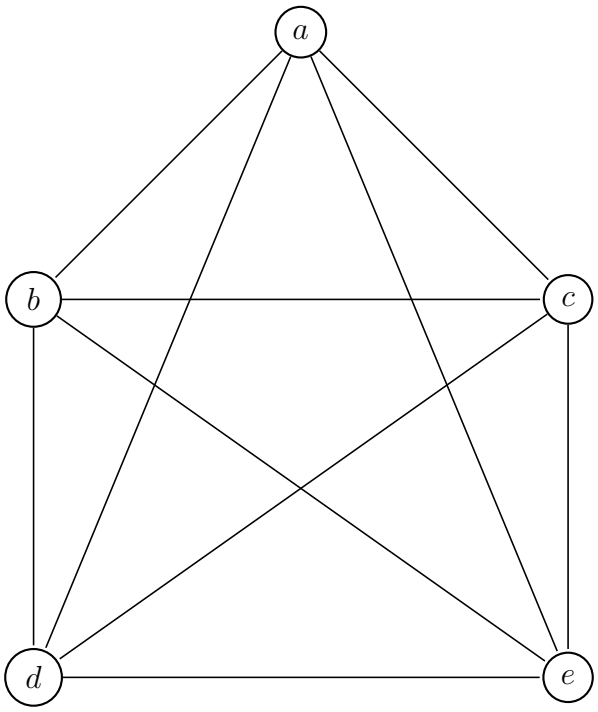


Figure 5: Graph G consisting of all the possible edges

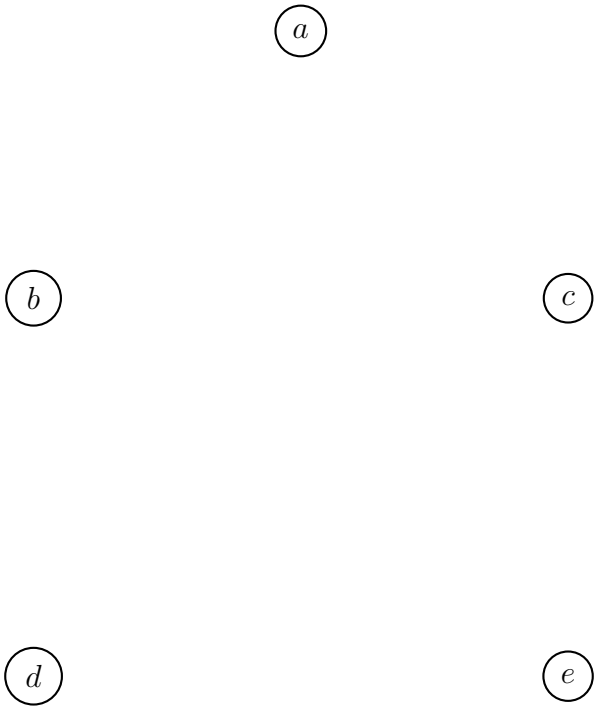


Figure 6: Graph $G^{'}$ consisting of edges not present in Figure 4