COMP 6651 ADT Winter 2023
Assignment 2
Submitted by: Rajat Sharma (40196467)

1. [You are supposed to have some elementary knowledge of graph theory for this question, review it or ask PoD for help if you need it.]
In lecture 3 we discussed the jobs (where jobs are represented by intervals) scheduling problem and its solution via a greedy algorithm. The job scheduling problem can also be represented by a graph: each job (interval) is represented by a vertex, and two vertices are connected by an edge whenever the corresponding two jobs have a conflict (i.e., intervals overlap). Such graphs are called interval graphs. See Figure 1 and Figure 2-(a) for an illustration. Convince yourself that the graph in Figure 2-(b) is not an interval graph, i.e., there are no 4 interval combinations whose graph representation gives you the graph in Figure 2-(b).

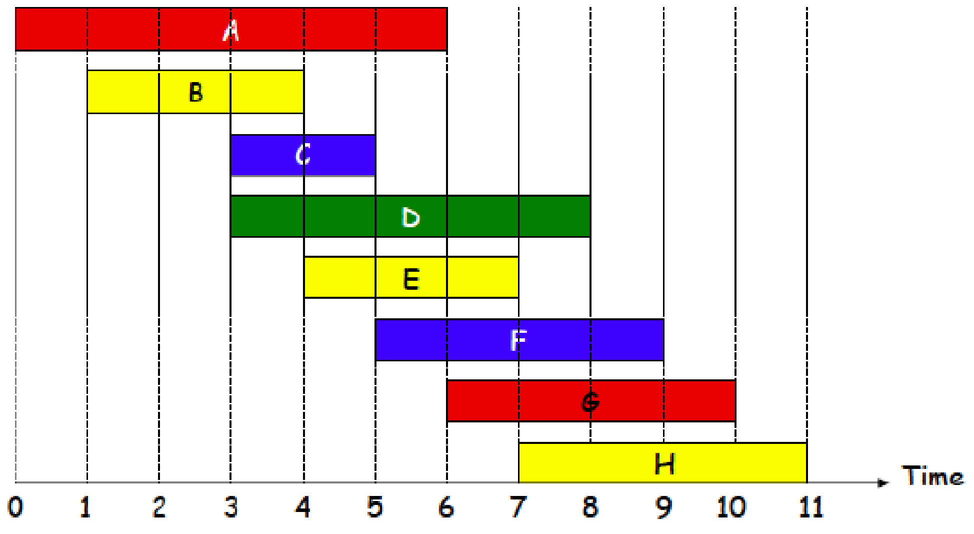


Figure 1: job scheduling problem

Observe that a subset of compatible jobs corresponds to a subset of vertices in the corresponding interval graph satisfying the following condition:
the Condition: no edges between any two vertices in this subset.
In graph theory, a subset of vertices satisfying the Condition is called an independent set.
In Figure 2, {B, E, H}, {A, G}, {C, F}, {D}, they are all independent sets of the graph, among which {B, E, H} is the maximum independent set of size 3. Clearly, finding a maximum subset of compatible jobs is equivalent to finding a maximum independent set in the corresponding interval graph representation.

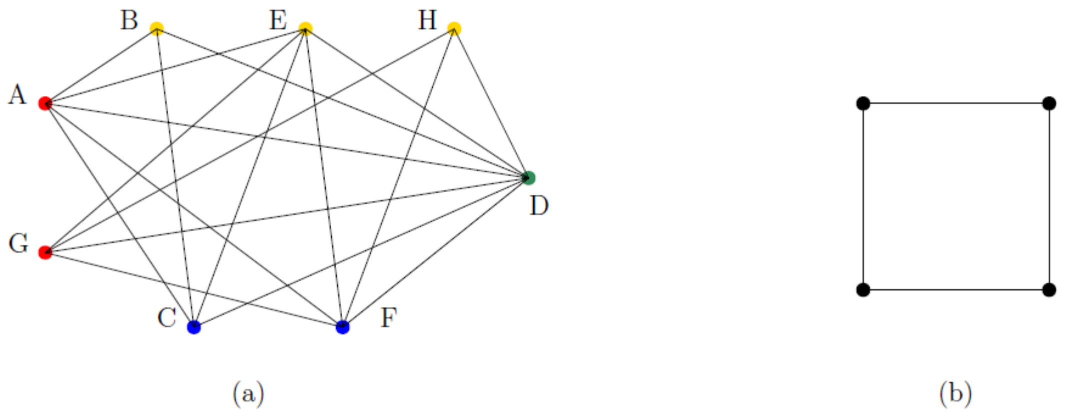


Figure 2: (a) Interval graph representation of job conflicts, (b) NOT an interval graph.

Below we discuss the independent set problem in arbitrary graphs: given a graph G = (V, E), find one maximum independent set. Since the goal is to find vertices where there are no edges between them, it seems natural to start with a vertex of the lowest degree. So, consider the following natural lowest-degree greedy algorithm A: find a vertex v ∈ V of the lowest degree, add v into a set S, remove v and all its neighbors, and then repeat until there are no more vertices
(1) Write pseudocode for the above greedy algorithm A. What is its time complexity?
(2) Prove that the subset S output by A is always an independent set of G.
(3) Consider the following graphs: the complete graph K_n, the complete bipartite graph $K_{n,n}$. If you run A on these graphs, do you get one maximum independent set? Give your reason.
(4) Is it true that for any graph G, A will output one maximum independent set of G? If yes, prove it; if no, give a counter-example.

Ans)

(1)
The greedy algorithm for the maximum independent subset is similar to the one to construct a single maximal independent set, but we want to choose vertices in each step such that the resulting set is not just maximal but maximum.
• Select a vertex of least degree v1 and add it to a set S. Take the induced subgraph G − N1, where N1 is the set of v1 and all vertices adjacent to v1.
• Select a vertex of least degree v2 and add it to S. Take the induced subgraph G−N1 − N2, where N2 is the set of v2 and all vertices adjacent to v3.
• Select a vertex of least degree v3 and add it to S. Take the induced subgraph G−N1 − N2 − N3, where N3 is the set of v3 and all vertices adjacent to v3.
.
.

.
• Continue until no viable vertices remain. Then S is a maximum independent set (hopefully).

Conjecture 1. The maximum independent subset must contain a vertex of the least degree.

However, now we are left with the natural question: What if two (or more) vertices in the same step have the same degree? Does it matter which one we choose?

Define: the secondary degree of a vertex v, denoted $\deg^2(v)$, as the sum of the degrees of each vertex adjacent to v.
Define: the tertiary degree of a vertex v, denoted $\deg^3(v)$, as the sum of the degrees of each vertex adjacent to a vertex adjacent to v, not including vertices already considered in a lesser degree.
Define: in general, the nth degree of a vertex v, denoted $\deg^n(v)$, as the sum of the degrees of vertices adjacent to the vertices considered for the $(n - 1)^{th}$ degree, not including vertices already considered in a lesser degree

If two vertices, u and v have the same degree, but $\deg^2(u) > \deg^2(v)$, then choosing u will eliminate the same number of vertices as choosing v. However, by eliminating the vertices with a greater total degree, the induced subgraph will be less connected, that is, this subgraph will have fewer edges. This implies that the sum of the degrees of the remaining vertices will be lower, so in general, they will be better candidates to be chosen in the next step than the vertices that would remain if we had chosen v. These vertices are “better” candidates because of the strength of the property of a lower degree sum.

Pseudocode for A:

```
INPUT: A graph $G = (V, E)$
OUTPUT: A maximum independent set S

1. Initialize an empty set S = {}
2. Repeat the following until V is empty:
   a. Find a vertex v in V with the lowest degree
      ● If there is more than one vertex of the least degree, choose the vertex with the greatest secondary degree.
           ★ If multiple of those vertices have the same secondary degree, choose the vertex with the least tertiary degree.
           ★ In general, if $d^n(v1) = d^n(v2)$, choose the vertex of greatest $(n + 1)^{th}$ degree if $n + 1$ is even and the least if $n + 1$ is odd.
           ★ If two vertices have the same $n^{th}$ degree for all n then either may be chosen.
   b. Add v to S
```

```
    c. Remove v and all its neighbors from V
3. Return S
```

The time complexity of this algorithm is O(n^2), where n is the number of vertices in the graph. This is because finding the vertex with the lowest degree in each iteration takes O(n) time, and there are n iterations, so the total time complexity is O(n^2).

(2)

Proof by contradiction.

To prove: That the subset S output by the algorithm is always an independent set, we need to show that there are no edges between any two vertices in S.

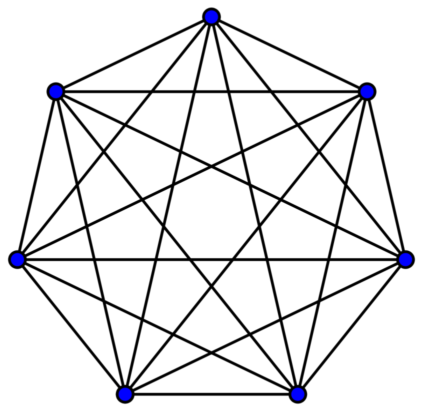
Suppose, for the sake of contradiction, that there is an edge (u, v) between vertices u and v in S. This means that u and v were both added to S in different iterations of the while loop.

Since u and v are connected by an edge, they must have been neighbors at some point in the algorithm. This means that either u was added to S before v, or v was added to S before u. Without loss of generality, let's assume that u was added to S before v.

When u was added to S, all its neighbors, including v, were removed from V. This means that when v was considered for addition to S, it was no longer a neighbor of u. However, v still had an edge to u, which contradicts the assumption that it was removed from V.

Therefore, our assumption that there was an edge between u and v in S was false, and S must be an independent set of G.

(3)

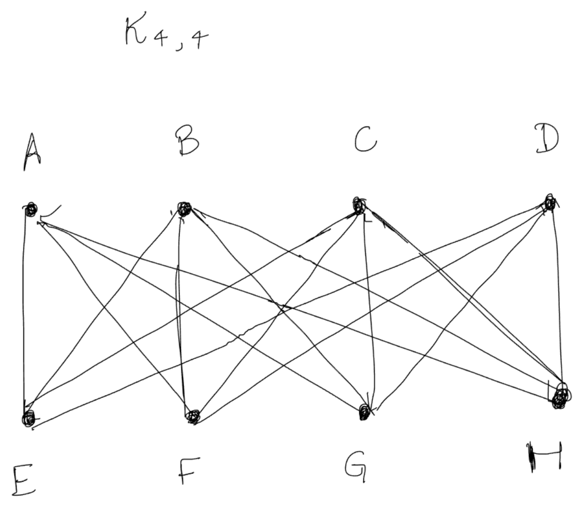


Consider the graph given above it is K_7 each vertex is connected to all other 6 vertices.

Running A on the complete graph K_n:
No, A will not output one maximum independent set of K_n. The algorithm always adds a vertex with the lowest degree to the independent set S, and in the complete graph K_n, every vertex has n-1 neighbors, so every vertex has the same degree, which is n-1. Thus, the algorithm will always add at least one vertex to S. If I consider the algorithm written above by me, after assuming v to be the starting vertex after the first iteration, it will remove all n-1 vertices in the graph. So, the starting vertex can be any vertex for every vertex, the answer will be 1, and the maximum set size will be 1 containing only the starting vertex.

For K_7 it will be max({A}, {B}, {C}, {D}, {E}, {F}, {G})

So for K_n it will output n maximum independent set with size 1.



No, A will output two maximum independent sets of $K_{n,n}$. The maximum independent set of $K_{n,n}$ is the set of vertices from one part since all vertices in one part have the same degree (n) and the algorithm will add all vertices from one part to S.

For the considered graph above, the two sets of $K_{n,n}$ are

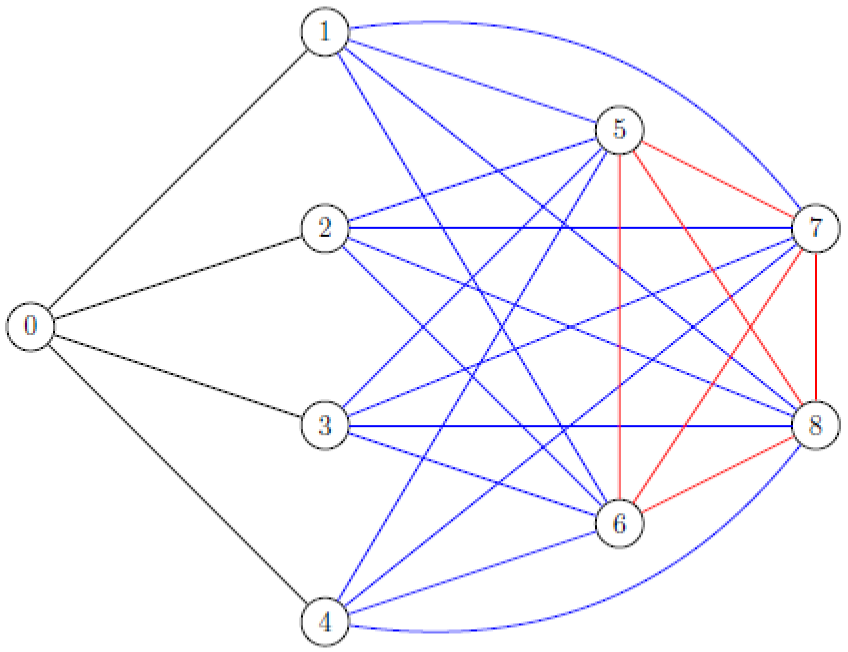
max({A, B, C, D}, {E, F, G, H}) this is because each vertex on one side is connected to n vertex on the other side. And both of them will have the same size n.

So for $K_{n,n,}$ it will output 2 maximum independent sets with size n.

(4)

No, it is not true that A will output one maximum independent set of G for any graph G. A is a greedy algorithm, meaning that it makes locally optimal choices at each step with the hope of finding a globally optimal solution. However, this is not always the case, and there may exist graphs where the locally optimal choices made by A do not result in a globally optimal solution.

Example:



Theorem 1. *The MIS does not necessarily contain a vertex of least degree.*

(Reference: Figure taken from Deterministic Greedy Algorithm for Maximum Independent Set Problem in Graph Theory
Joshua C. Ballard-Myer
December 18, 2019)

For example, in this graph with vertex with least degree is 0 but it can be clearly seen that the maximum independent subset will be {1, 2, 3, 4}. We can see that the algorithm fails on the very first step because the vertex of the least degree is not in the maximum independent set. Selecting the vertex of the least degree eliminates the maximum independent subset from the graph.

2. You are given a set of m constraints over n Boolean variables x1, . . . , xn. The constraints are of two types:

- **equality constraints: $x_i = x_j$ for some $i \neq j$;**
- **inequality constraints: $x_i \neq x_j$ for some $i \neq j$.**

Design an efficient greedy algorithm that, given the set of equality and inequality constraints, determines if it is possible or not to satisfy all the constraints simultaneously. If it is possible to satisfy all the constraints, your algorithm should output an assignment to the variables that satisfy all the constraints.

(1) Choose a representation for the input to this problem and state the problem formally using the notation Input: ..., Output: ....
(2) Describe your greedy algorithm in plain English. In what sense is your algorithm "greedy"?
(3) Describe your greedy algorithm in pseudocode.
(4) Briefly justify the correctness of your algorithm.
(5) State and justify the running time of your algorithm. The more efficient algorithm, the better. Inefficient solutions may lose points.

(1) The input to this problem can be represented as a set of m constraints over n Boolean variables {x1, x2, ..., xn}, where each constraint is either an equality constraint of the form xi = xj, for some i != j, or an inequality constraint of the form xi != xj, for some i != j.

Formally, we can define the problem as follows:

Input:

A set of m constraints C = {c1, c2, ..., cm}, where each constraint ci is either an equality constraint of the form xi = xj, for some i != j, or an inequality constraint of the form xi != xj, for some i != j, and n Boolean variables {x1, x2, ..., xn}.

Output:

If it is possible to satisfy all the constraints simultaneously, output an assignment to the variables that satisfy all the constraints. We can output a boolean string representing {x1,x2,...,xn}.

If it is not possible to satisfy all the constraints simultaneously, output "UNSAT" to indicate that no such assignment exists.

(2)
The problem we are trying to solve is to find a valid assignment of either true or false values to a set of variables such that all the given equality and inequality constraints are satisfied. We can

represent these constraints using a graph, where each variable is a node in the graph, and each constraint is an edge connecting two nodes.

To solve this problem, we can use a greedy algorithm that works as follows:

- Create two arrays of "buckets" to represent the equality and inequality constraints, where each bucket contains the nodes that are connected by the corresponding type of edge.
- Initialize an array of the same length as the number of nodes in the graph, with each element representing the assigned value of the corresponding node.
- Initially, all nodes are set to an unknown value represented by "?".
- For each node that has not been assigned a value yet, set its value to true and check if this assignment satisfies all the constraints in its equality and inequality buckets. If so, continue to the next unassigned node. If not, try setting its value to false and check again. If neither assignment satisfies the constraints, then we know that there is no valid assignment for the entire graph.
- If we have successfully assigned values to all the nodes, then we have found at least one valid assignment for the graph. Output this solution.

The algorithm is "greedy" in the sense that it tries each possible value of each variable in turn, selecting the first one that doesn't violate any constraints. It is also recursive, backtracking whenever it encounters a violation of a constraint.

(3)

Pseudocode:

```
func set(i, val):
  global solution, equalities, inequalities
  if solution[i] != '?':
    return solution[i] == val
  solution[i] = val
  for j in equalities[i]:
    success = set(j, val)
    if not success:
      return False # Contradiction found
  for j in inequalities[i]:
    success = set(j, not val)
    if not success:
      return False # Contradiction found
  return True # No contradiction found

func solve():
```

```
global solution, equalities, inequalities
solution = ['?' for _ in range(len(equalities))]
for i in range(len(solution)):
  if solution[i] != '?':
    continue # value has already been set/checked
  success = set(i, True)
  if not success:
    print("No solution")
    return
print("At least one solution exists. Here is a solution:")
print(solution)
```

(4)

The correctness of the algorithm can be justified as follows:

The Set function recursively sets the value of each index in the solution list to either true or false if the value is initially ?. It does this by checking for contradictions, which occur if a variable is set to both true and false or if it has a direct inequality constraint with a variable that has already been set to the same value. If a contradiction is found at any point, the Set function returns false, indicating that no valid solution can exist.

The Solve function initializes each index in the solution list to ? and then loops through each index, calling Set on each index that has not yet been set. If Set returns false for any index, indicating that no valid solution exists, the Solve function prints "No solution" and returns. If Set returns true for all indices, the Solve function prints "At least one solution exists. Here is a solution:" followed by the solution list.

The algorithm is correct because it exhaustively searches all possible solutions by recursively setting the value of each variable, checking for contradictions at each step, and backtracking if necessary. If there is a valid solution, the algorithm will find it, and if there is no valid solution, the algorithm will correctly report that no solution exists.

(5)

The time complexity of this algorithm is O(m+n), where m is the number of constraints and n is the number of nodes in the graph. This is because we need to read all the constraints to create the "bucket" arrays, and we need to check each node's value and its corresponding constraints, which can be done in O(m+n) time.

It's worth noting that if there are multiple connected components in the graph, we can solve each component independently, which reduces the number of nodes we need to check. The space complexity of this algorithm is also O(m+n), as we need to store the "bucket" arrays and the array of assigned values.

3. Consider the Longest Increasing Subsequence (LIS) problem (see slides for problem definition).
(1) Give a greedy algorithm for LIS. You should: first, describe your algorithm in English, then write down the pseudocode, and finally point out its time complexity.
(2) Your algorithm most likely is not optimal. Give one example in which your greedy algorithm outputs an optimal solution. Give another example in which your greedy algorithm outputs a suboptimal solution.

(1)
Greedy Algorithm for Longest Increasing Subsequence (LIS)
Algorithm Description:
The greedy algorithm for LIS works as follows:

- We choose the first element of the array to be part of the longest increasing subsequence.
- We keep track of the longest increasing subsequence we have seen so far, updating it whenever we find an element greater than the last appended element in the list.
- At each step, we choose the next element in the input sequence that can be appended to the end of the current longest increasing subsequence.
- We repeat this process until we have processed all elements in the input sequence.

Pseudocode:

```
function LIS(arr)
  n = length of arr
  lis = arr[0]
  for i = 2 to n
      if arr[i] > lis[lis.size -1]
                  lis.append(arr[i])
  return len(lis)
```

Time Complexity:
The time complexity of the greedy algorithm for LIS is O(n), where n is the length of the input sequence.

(2)

Examples
Example 1:
Input: [1, 2, 3, 4, 5]
Output: 5 (LIS = [1, 2, 3, 4, 5])
Explanation: In this example, the greedy algorithm outputs an optimal solution as the LIS of the input sequence is the input sequence itself.

Example 2:
Input: [10, 22, 9, 33, 21, 24, 25, 50, 41, 60, 80]
Output: 6 (LIS = [10, 22, 33, 50, 60, 80])
Explanation: In this example, the greedy algorithm outputs a suboptimal solution as the length of the LIS is 6, but the actual LIS of the input sequence is [10, 21, 24, 25, 50, 60, 80], which has length 7.

4. Consider the maximum-subarray problem. We've seen a divide and conquer algorithm for it using Θ(n log n) time. Use dynamic programming to design a linear time, i.e., Θ(n), the algorithm for it. You can refer to exercises 4.1-5 on page 75 of our textbook (CLRS, Introduction to algorithms, 3rd ed.) for ideas.

(1) First describe your algorithm in English, then write down the pseudocode, then point out why its time complexity is Θ(n).
(2) Run your algorithm on the following example:

13,−25, 18, 20,−7, 12,−5, 4.
Specifically, this means writing down the array used in your dynamic programming algorithm and filling in the values as your algorithm will do. [Double check that your algorithm indeed outputs the optimal solution: this is not a part of the work to be submitted, just verify yourself.]
Explanation:
The idea of the maximum subarray problem involves iterating over the input array and keeping track of two values: the maximum subarray sum seen so far and the maximum subarray sum ending at the current index. At each index, we update the maximum subarray sum ending at that index by checking whether the previous maximum subarray sum ending at the previous index plus the current element is greater than the current element itself. If it is, we update the maximum subarray sum ending at the current index. We then update the overall maximum subarray sum if the new maximum subarray sum ending at the current index is greater than the current maximum subarray sum. After iterating over the entire input array, the maximum subarray sum found is returned.
Pseudocode:

1) Set both max_sum_ending_here and max_sum_so_far to 0.
2) For each element in the input array:

    1. Update max_sum_ending_here to be the maximum of the current element and the sum of the previous maximum subarray sum ending at the previous index and the current element.
    2. Update max_sum_so_far to be the maximum of the current max_sum_so_far and max_sum_ending_here.
3) Return the final value of max_sum_so_far as the maximum subarray sum.

The time complexity of this algorithm is linear in the size of the input array. This is because we only iterate over the input array once and perform constant-time operations at each index, resulting in a time complexity of Θ(n), where n is the length of the input array.

Index	**Element**	**Max subarray sum ending at index**	**Max subarray sum so far**
0	13	13	13
1	-25	0	13
2	18	18	18
3	20	38	38
4	-7	31	38

5	12	43	43
6	-5	38	43
7	4	42	43

Therefore, the maximum subarray sum is 43, which corresponds to the subarray [18, 20, -7, 12].