CS 330, Fall 2025, Homework 4

Due: Wednesday, 10/01, at 11:59 pm on Gradescope

Collaboration policy If you choose to collaborate on some problems, you are allowed to discuss each problem with at most 3 other students currently enrolled in the class. Before working with others on a problem, you should think about it yourself for at least 45 minutes.

You must write up each problem solution by yourself without assistance, even if you collaborate with others to solve the problem. You must also identify your collaborators. If you did not work with anyone, you should write "Collaborators: none." It is a **violation of this policy to submit a problem solution that you cannot orally explain to an instructor or TA.**

Typesetting Solutions should be typed and submitted as a PDF file on Gradescope. You may use any program you like to type your solutions. LaTeX, or "Latex", is commonly used for technical writing (`overleaf.com` is a free web-based platform for writing in Latex) since it handles math very well. Word, Google Docs, Markdown or other software are also fine.

Solution guidelines For problems that require you to provide an algorithm, you must give the following:

1. a precise (and concise) description of the algorithm in English and pseudocode (*),
2. a proof of correctness,
3. an analysis of the asymptotic running time of your algorithm.

You may use anything we learned in class without further explanation. This includes using algorithms from class as subroutines, stating facts that we proved in class, e.g., correctness of subroutines, running time of subroutines and use the notation. Your description should be at the level that it is clear to a classmate who is taking the course with you.

You should be as clear and concise as possible in your write-up of solutions.

A simple, direct analysis is worth more points than a convoluted one, both because it is simpler and less prone to error and because it is easier to read and understand.

(*) It is fine if the English description concentrates on the high level ideas and doesn't include all the details. But the reader should not have to figure out your solution solely based on the pseudocode. You can also add comments to your pseudocode, in fact that is best practice.

David Bunger, Collaborators: None

1. Interesting Edges in DAG (10 points)

A team of software engineers is using a collaborative project management tool to organize a complex set of tasks. Tasks have dependencies: some must be completed before others can begin. These dependencies are modeled as a DAG, where each node is a task and each directed edge (u, v) means "task u must be completed before task v starts."

While scheduling, the team has noticed that some tasks are always tightly paired—that is, task v always follows task u immediately, no matter how the rest of the schedule is rearranged. They define such dependencies as "interesting". The team wants to automate the detection of such interesting dependencies to help identify tightly coupled tasks that might benefit from being bundled.

Formally, an edge (u, v) in a DAG G is said to be interesting if every topological ordering of G places u immediately before v, with no node between them. Below are examples of interesting edges in DAG:

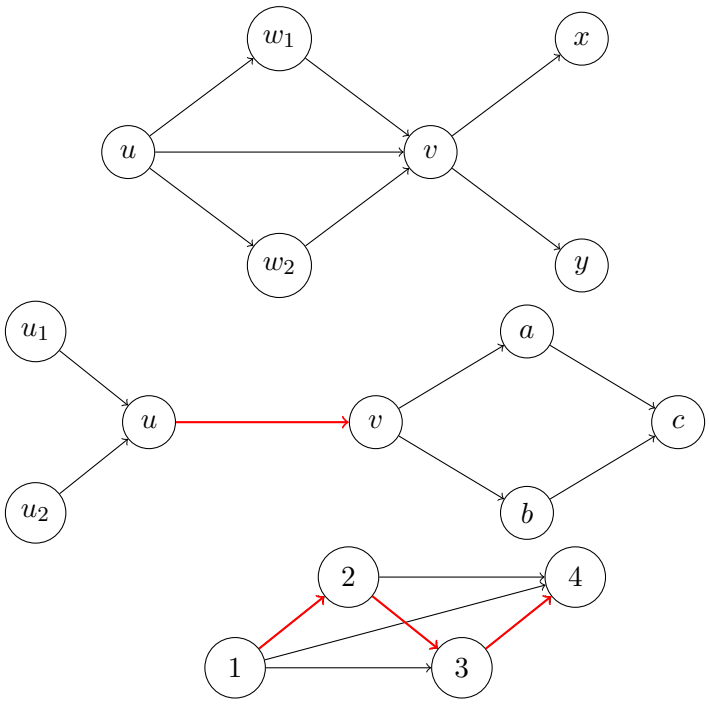


Figure 1: Examples of edges in DAGs: (a) edge (u, v) is not interesting, since w_1 or w_2 can come between; (b) (u, v) is the only interesting edge; (c) the complete DAG with $i \to j$ for $i < j$, where each consecutive $(i, i + 1)$ is interesting.

Your Tasks:

(a) Design an algorithm that takes as input a DAG $G = (V, E)$ and an edge (u, v) and returns True if the edge is interesting, and False otherwise.

(b) Give a *full* proof on why your algorithm is correct.

Hint: You can use the following lemma without proof: Let (u, v) be an edge in DAG G. If a node x is neither an ancestor of u nor a descendant of v (i.e., there is no path from x to u, and no path from v to x), then there exists a topological order of G where x is placed between u and v.

(c) Analyze the runtime of your algorithm as a function of $n = |V|$ and $m = |E|$.

Teaching Note

In this problem we want you to get an intuition for dependencies and doing math about dependencies. We want to strengthen your understanding of DAGs, topological orders, and dependency graphs in general.

(a)

Algorithm 1: InterestingEdge (G, (u, v))

```
1  GReverse ← {}
2  for i ∈ G do
3      for j ∈ i do
4          Greverse[j][i] = 1
   /* create a copy of G with its edges reversed                                  */
5  ConnectedU = BFS(G, u)/* performs BFS on G from node u to find its connected nodes  */
6  ConnectedV = (BFS(GReverse, v)/* performs BFS on GReverse from node v to find which
       nodes are connected to it                                                  */
7  if ConnectedU.union(ConnectedV).length()! = G.length() then
8      return false
   /* if ConnectedU and ConnectedV do not contain all the nodes in G, there must be some
       disconnected component, meaning the edge can never be interesting and false is returned
       */
9  I = ConnectedU.intersection(ConnectedV)/* finds the intersection between the set of
       nodes that u connects to and the set of nodes connected to v               */
10 if I == {} then
11     return true
   /* if there are no nodes in the intersection, the edge is interesting and true is returned
       */
12 else
13     return false
   /* if there are nodes in the intersection, there are nodes that can be placed between u and
       v in the topological order, meaning the edge is not interesting and false is returned  */
```

(b)

The reverse of the given lemma would be that if a node y exists that is both a descendant of u and an ancestor of v, then there exists a topological order of G where y is placed between u and v. This is true because, assuming all nodes are in the same connected component, if a node is not an ancestor of u, it must be a descendant of u, and if a node is not a descendant of v, it must be an ancestor of v. However, this doesn't account for nodes that are in a separate, disconnected component from u and v. If a node z exists within a disconnected component from u and v, then it is neither an ancestor of u not a descendant of v, meaning then there exists a topological order of G where z is placed between u and v by the given lemma. The algorithm uses BFS on G and u to find all of the descendants of u, and then it uses BFS on G with its edges reversed and v to find the ancestors of v. It then confirms that these two searches contain all the nodes in G, meaning G

is one connected component. Finally, it checks the intersection between the descendants of u and the ancestors of v. If that intersection contains any nodes, it would mean there is at least one node that is both a descendant of u and an ancestor of v, which would mean that node/those nodes can be placed between u and v in a topological order and the edge (u, v) is not interesting. If there are no nodes shared between these two sets, the edge (u, v) must be interesting and the algorithm returns true.

(c)

Creating the reversed graph of G iterates over every edge of the graph once, giving it a runtime of $O(n + m)$. We know the runtime of BFS is $O(n + m)$, and it is performed twice. Worst case, assuming the union of $ConnectedU$ and $ConnectedV$ contains all the nodes in G, then the union and length check both take $O(n)$, iterating over each node once. The intersection of $ConnectedU$ and $ConnectedV$ would also have to iterate through each node once to compare the sets, giving it a runtime of $O(n)$. Finally, the last check has a runtime of $O(1)$. Overall, the runtime for the algorithm is $O(n + m)$.

2. CFA Crawl (10 points)

This semester, you and your friends have gotten a little too into the student art scene. Every week, the CFA hosts pop-up galleries where student artists staff their exhibits. For your Art History class, you need to meet every artist at least once to get full credit for the assignment.

Each artist $i \in \{1, \ldots, n\}$ posts an interval $[s_i, f_i]$ during which they will be present. You want to find an *acceptable* set of times $\{t_1, \ldots, t_k\}$ such that each interval $[s_i, f_i]$ contains at least one t_j (i.e $s_i \leq t_j \leq f_j$). Your goal is to minimize k, the number of gallery visits.

In the example of Figure 2, the black vertical lines mark a set of acceptable visit times. Removing any one of them would make the set not acceptable. (But it is still not the smallest acceptable set.)

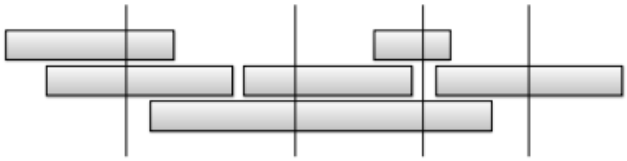


Figure 2: An acceptable set of four visit times.

(a) (Do not hand in) Find a smallest acceptable set of attendance times for the example in Figure 2.

(b) Your friend suggests a greedy strategy: always pick the time that covers the largest number of remaining artists. Give an example where this fails to find the minimum hitting set.

(c) Another friend observes: "If the input contains k mutually disjoint intervals, then every acceptable set of visit times must have size at least k." Is this true? Prove or disprove.

(d) Design a polynomial-time algorithm that, given $(s_1, f_1), \ldots, (s_n, f_n)$, outputs a minimum-size set of visit times. [*Hint:* modify the interval scheduling algorithm.]

(e) What is the running time of your algorithm in term of n?

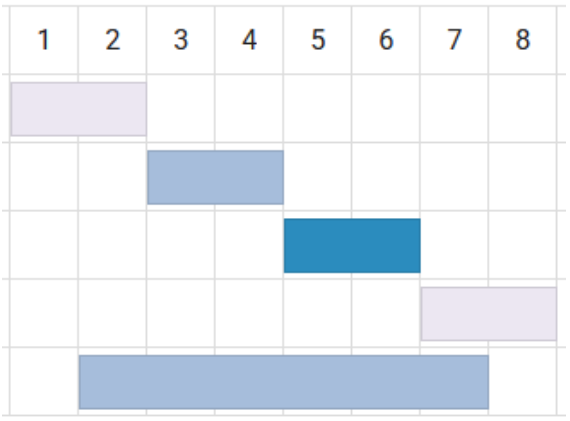
(f) Prove your algorithm is correct. [*Hint:* Use the lower-bound observation from part (c).]

Teaching Note

In this problem we want you to get an intuition for greedy algorithms. How do you come up with a greedy algorithm? How can you prove your greedy algorithm is optimal? How can you take the techniques from class and apply them to a new situation?

Solution:

(b)



This example shows three different points where there are three artists remaining: t_3, t_5, t_7. Since they are tied for the most artists remaining, the strategy could choose any of the three. If it chooses t_5 first, there will be two remaining artists that cannot be visited together, resulting in three visits. If t_3 is chosen first, then it will choose t_7 next and all artists will be covered in two visits. In this case, the strategy given can not guarantee it will pick the correct set of artists.

(c) Yes this is true. If there are any disjoint intervals, there are no points where they overlap, meaning they cannot be visited at the same time. It is possible that they will need to be visited multiple times based on the other, non-disjoint intervals, but there is no way to visit any disjoint intervals simultaneously, Therefore, if there are k mutually disjoint intervals, there must be at least one visit for each of those intervals, meaning there must be at least k visits.

(d)

Algorithm 2: EarliestFinishTimeFirst $(i = 1...n : s_i, f_i)$

```
1 sorted_artists=mergeSort(f_1, f_2, ..., f_n)/* sorts the artist intervals by earliest finish time
    first                                                                              */
2 V ← []/* creates an empty list to store the visit times                              */
3 for i ∈ sorted_artists do
  |  /* loops through each artist interval                                            */
4 |  if V = [] or s_i > V[−1] then
5 |  |  V.add[f_i]
  |  |  /* if the list of visit times is empty or if the start time of the current interval
  |  |     is later that the most recent visit time, add the finish time of this interval to
  |  |     the list of visit times                                                     */
6 return V
```

Solution:

(e) The runtime of mergeSort is $O(nlogn)$. Creating an empty list has a runtime of $O(1)$. The for loop accesses each artist interval once, compares the start time of that interval with the last value in the list ($O(1)$), and then adds the finish time of that interval to the list ($O(1)$). This gives the loop a runtime of $O(n)$, meaning the total runtime is $O(nlogn)$.

(f) Based off the observation from part (c), we know the minimum number of visits in a schedule would be k, where k is the number of mutually disjoint intervals. We also know that the maximum number of visits is equal to the total number of intervals.

Given an optimal solution as close to k as possible, we want to prove that the greedy solution is equivalent to the optimal solution.

If we assume the greedy solution is equivalent to the optimal solution up to the visit time r, we want to select the next visit time $r+1$ in a position as good as or better than the optimal solution. Since we have selected visit times based on earliest finish time first, all intervals with finish times less than or equal to r have been visited. This means the next interval, which we can call x, with a start time after r must be disjoint from some of the prior intervals, since it has not been visited yet. Given that, the next visit time needs to include x in order for the solution to have at least k visits. The greedy solution would select the very last moment during x, its finish time, in order to include it. The optimal

solution may have selected an earlier time for $r+1$, but it cannot select a later time, as that wouldn't include x. Since the finish time of x was chosen by the greedy solution, any unvisited intervals that are not disjoint with x will be included in this visit, as they must start after r and finish after the finish time of x. The optimal solution would also need to include these intervals, as x may be the only interval they are not disjoint with. (If they are not chosen here, they would require an extra visit to be chosen). Considering the greedy $r+1$ includes the same intervals as the optimal $r+1$, we can set the optimal $r+1$ equal to $r+1$ in the greedy solution. Now, the optimal solution and the greedy solution are equal all the way to $r+1$. This allows us to do the same with $r+2$ and so on until all the visits in the optimal solution are equivalent to all the visits in the greedy solution, without changing the total number of visits in the optimal solution.