

CS375 Weekly Assignment 3 Solutions

Due Mar 3rd 10PM ET

A general note for CS375: When writing up your homework, please write neatly and explain your answers clearly, giving all details needed to make your answers easy to understand. Graders may not award full credit to incomplete or illegible solutions.

You are encouraged to discuss the problems with your classmates, **but your submission should be your own**. When writing your solutions up, you should not be looking at any other student's submission.

Clear communication **is the point**, on every assignment. In general in CS375, unless explicitly specified otherwise, answers should be accompanied by explanations. Answers without explanations may not receive full credit. Please feel free to ask me any questions about explanations that might come up!

Necessary acknowledgements: many of these questions are reproductions or adaptations of questions of Eric Aaron's devising. Others are from our course texts as listed on the syllabus.

1. In class, we defined DEFERRED-ACCEPTANCE algorithm in an extremely hand-wavy way as:

```
def DA(n men, n women, their preferences):  
    until all men are paired off:  
        each unpaired man proposes to their highest preferred woman they have  
            not yet proposed to  
  
        each woman tentatively accepts their highest preferred man from their  
            options  
  
    return the matching formed by each woman and their current partner
```

- (a) Your job is to code it up. Download [this file](#) I've started for you. **You should only edit lines 57-60, namely the deferred acceptance method** - include the updated file in your submission. It's rare in this class that I'll ask you to code something up, but I think this one is fun enough that you should do it.

Solution: So I whipped [this](#) together.

- (b) Analyze your code: determine both its best-case and worst-case runtimes (I'm looking for a Θ bound for both). Note that there are a variety of ways to implement this algorithm so your answer could very well differ from your friends'.

Solution: Let's go through the code: initially we run a for-loop that takes $\Theta(n)$ time to initialize a few things. After this, we enter a while-loop the runtime of which is a little trickier to bound. What we need to do is determine how the algorithm makes progress. Notice that in each iteration of the while-loop, a man makes a proposal to a woman he has not yet proposed to. This can only happen n times for any particular man (as there are n women to whom he can propose), meaning that this while-loop can only iterate *at most* n^2 times. It must iterate *at least* n times in order for each initially unmatched man to propose to a woman. Combined, this tells us that the runtime is $\Omega(n)$ and $O(n^2)$ - to be more exact so that we can talk about best- and worst- case, let's look at some explicit instances. Each of the following instances will have n men $m_1, \dots, m_n$ and n women $w_1, \dots, w_n$ - what will differ about the instances are the preferences of each person over the members of the other sex.

- First, suppose that each man's m_i 's first choice is w_i (I don't care what the rest of the preferences look like). Then each man will propose to a unique woman and no man will ever become unmatched after their initial proposal. This means the while

loop will have exactly n iterations, making the runtime of this instance $\Theta(n)$. **As a result, the best-case runtime is $O(n)$ and the worst-case runtime is $\Omega(n)$.** Since we already knew the best-case runtime is $\Omega(n)$, this tells us that the best-case runtime is $\Theta(n)$. This tells us nothing new about the worst-case runtime, since we already knew it was $\Omega(n)$ and $O(n^2)$.

- Next, suppose every man's preference is $[w_1, \dots, w_n]$ while every woman's preference is $[m_n, m_{n-1}, \dots, m_1]$. Notice that the alg is using Python's `pop`, which will always take the last man in the list of unmatched men. To trace through the input:
 - i. m_n proposes to w_1 , who doesn't have a partner yet, so she tentatively accepts
 - ii. m_{n-1} proposes to w_1 who prefers him over m_n , so m_n returns to the pool of unmatched men.
 - iii. m_n then proposes to w_2 who doesn't have a partner yet, so she tentatively accepts.
 - iv. m_{n-2} then proposes to w_1 , who prefers him over m_{n-1} , so m_{n-1} goes back to the pool.
 - v. m_{n-1} then proposes to w_2 , who prefers him over m_n , so m_n goes back to the pool...
 - vi. and so on.

So what's happening here? Every time a man m_{n-i} proposes for the first time, he causes a cascade of men ($m_{n-i+1}, \dots, m_n$) to be pushed down their preferences. That means each man m_{n-i} creates i additional resultant proposals in addition to his own. As a result, the total number of proposals made (and hence the total runtime of the code on this instance) is $\Theta(\sum_{i=1}^n i) = \Theta(n^2)$ (we've seen that sum enough that I assume we're comfortable with it at this point). This tells us that the best-case runtime is $O(n^2)$ (which tells us nothing new) and that the worst-case runtime is $\Omega(n^2)$, which closes the gap to the bound we already knew (namely that the worst-case runtime is $O(n^2)$), giving us that the worst-case runtime is $\Theta(n^2)$.

2. Here's a slightly more general version of the Stable-Marriage problem:

Input: A collection of m companies, each of which has a specified hiring capacity, and a collection of n students. Each company has separately ranked all possible students, and each student has ranked each company.

Output: An assignment of students to companies such that each company doesn't hire more than its capacity of students, and the assignment adheres to the following *stability* rules:

- **Stability Condition:** A matching is stable if there is no pair of student and company (s, c) such that:
 - Company c has not filled their hiring capacity or prefers student s over at least one of its current hires.
 - Student s is unassigned or prefers company c over their current assignment.

If such a pair exists, the matching is unstable, and a better matching can be made by swapping assignments between the pair.

Adapt the pseudocode for the DEFERRED-ACCEPTANCE algorithm to work for this problem and show that the solution it produces is stable under the above definition.

Solution: Just have the unmatched students apply to their next top choice until they have exhausted all their options. Have the companies tentatively accept students until they meet their restriction, at which point more favorable candidates boot out previously tentatively assigned students.

Let student s be any student who prefers some company c more than their current assignment (if such a student exists - if not, then obviously the solution is stable) - note that a possible

assignment for this student might be ‘unemployed’. By definition of the algorithm, then s must have applied to c at some point, but as they are not in c ’s final allotment then they must have been booted in favor of some more preferable student from c ’s perspective. But, just as in the original DEFERRED-ACCEPTANCE algorithm, the company’s assignments are getting better from their perspective over the course of the algorithm. Additionally, once a company has filled their quota they do not lose students later. This means that c has both a full hiring roster and prefers every student in their final assignment over s , so s and c will not cause instability. As a result, there can be no instabilities in the solution our algorithm creates, showing that our algorithm produces stable assignments.

3. In the following list, I’m borrowing a bunch of questions from various texts. I’ll use Erickson/-CLRS to refer to our two course texts as listed on our class homepage, but I’ll also borrow some questions from *Algorithm Design*, by Jon Kleinberg and Eva Tardos, which I’ll refer to by AD.

Do 3 more of the following problems; **your answer should use an exchange argument if you are showing an algorithm is correct.**

- i) (AD 4.3) You are consulting for a trucking company that does a large amount of business shipping packages between New York and Boston. The volume is high enough that they have to send a number of trucks each day between the two locations. Trucks have a fixed limit W on the maximum amount of weight they are allowed to carry. Boxes arrive at the New York station one by one, and each package B_i has a weight w_i . The trucking station is quite small, so at most one truck can be at the station at any time. Company policy requires that boxes are shipped in the order they arrive; otherwise, a customer might get upset upon seeing a box that arrived after his make it to Boston faster. At the moment, the company is using a simple greedy algorithm for packing: they pack boxes in the order they arrive, and whenever the next box does not fit, they send the truck on its way.

But they wonder if they might be using too many trucks, and they want your opinion on whether the situation can be improved. Here is how they are thinking: maybe one could decrease the number of trucks needed by sometimes sending off a truck that was less full, and in this way allow the next few trucks to be better packed. Prove that, for a given set of boxes with specified weights, the greedy algorithm currently in use actually minimizes the number of trucks that are needed.

Solution: We’ll show this by means of an exchange argument - consider any other valid packing P of boxes, compare it to the solution produced by our algorithm, and look for the first point of disagreement. In particular, a disagreement for us would mean which truck a particular box B gets placed in. Note that this is the **first point of disagreement** - so all previous boxes were put in the same trucks. As a result, their solution could not have put B in an earlier truck than us - we fill trucks to their capacity, so if we were forced to start a new truck to fit B , then they must have to, because they had the same contents in the previous truck as we did. So their solution must have placed B in a later truck than we did. Let’s construct a new packing P' that does precisely the same packing as P , but instead places B in the same truck that we had. Again, this is was the first point of disagreement, so if we could fit B in its truck in our solution, then we can do so in P' as well. By doing so, we get that

- P' uses no more trucks than P - I haven’t added any new trucks to handle this exchange
- P' is still valid - I haven’t overfilled any trucks, because the only truck which has a new box in P' is the one we’ve moved B to, which cannot have been overfilled because our algorithm had space for it **and they agreed with us until this point.**
- P' agrees with our solution for an extra step - it places B in precisely the same truck as we did.

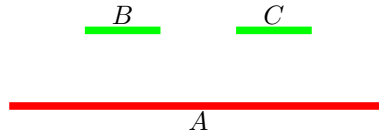
At this point, we can declare victory - for each future disagreement between the solution our algorithm produces and P' , we can apply the same exchange step, and eventually arrive at the same solution our algorithm produces. At each exchange, the adjusted solution never gets worse and remains valid - as a result, our solution is no worse than any other solution. In particular, this means that the solutions our algorithm produces is

no worse than *optimal solutions*, meaning the solutions our algorithm produces are also optimal.

- ii) (Erickson 4.1) (**This problem counts as two problems** - a midterm question will look exceptionally similar to this one almost certainly, so I'd recommend doing it regardless) The GREEDYSCHEDULE algorithm we described for the class scheduling problem is not the only greedy strategy we could have tried. For each of the following alternative greedy strategies, either prove that the resulting algorithm always constructs an optimal schedule, or describe a small input on which it fails. Assume that all algorithms break ties arbitrarily (that is, in a manner that is completely out of your control). **Hint: Three of these algorithms are actually correct.**

- i. Choose the course x that ends last, discard classes that conflict with x , recurse.

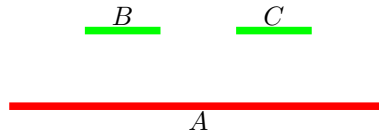
Solution: This algorithm is incorrect. Consider the following instance:



In this instance, only A would be chosen, yet B and C are optimal.

- ii. Choose the course x that starts first, discard classes that conflict with x , recurse.

Solution: This algorithm is incorrect. Consider the following instance:



In this instance, only A would be chosen, yet B and C are optimal.

- iii. Choose the course x that starts last, discard classes that conflict with x , recurse.

Solution: This algorithm is correct! We'll show its correctness by means of an exchange argument. Consider any valid solution S to our given input and look for the first point of disagreement (from the perspective of our algorithm – note that our algorithm proceeds from the right to the left). Let A be the interval we select and let B be the next earliest interval S includes. By definition of our algorithm, A is the interval which starts last out of all intervals not overlapping with intervals selected thus far, *which S has also included as this is the first point of disagreement*. As a result, it must be the case that B starts no later than A (or otherwise S would not be a valid solution). Note that the interval A (which we selected) cannot overlap with *any* of the intervals included in S (other than possibly B):

- A doesn't overlap with any intervals to the right of B in S because we agreed on those intervals and A didn't overlap with any of our intervals (by definition of our algorithm)
- A doesn't overlap with any intervals to the left of B because all such intervals must end before B starts, *which is no later than when A starts*.

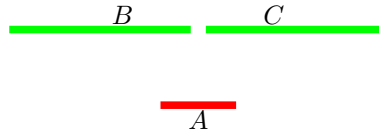
Consider an adjusted solution S' that is identical to S except that B has been replaced by A . Then

- S' is no worse than S - it includes exactly the same number of intervals selected.
- S' is a valid solution - as pointed out above, A doesn't overlap with any of the other intervals included in S' .
- S' agrees with our algorithm's input for an additional step.

As above, we can immediately declare victory.

- iv. Choose the course x with the shortest duration, discard classes that conflict with x , recurse.

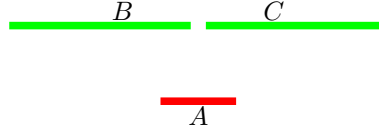
Solution: This algorithm is incorrect. Consider the following instance:



In this instance, the algorithm would select only A , yet B and C are the optimal solution.

- v. If none of the courses conflict, choose them all; otherwise, discard the course of longest duration, recurse.

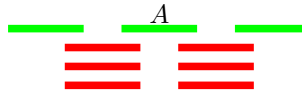
Solution: This algorithm is incorrect. Consider the following instance:



In this instance, the algorithm would select only A , yet B and C are the optimal solution.

- vi. If none of the courses conflict, choose them all; otherwise, discard the course which conflicts with the most other courses, recurse.

Solution: This algorithm is incorrect. Consider the following instance:



In this instance, the algorithm would start by removing A . As soon as it does so, it is impossible to reach a solution containing 3 non-overlapping intervals. However, the optimal solution would include each of the 3 green intervals.

- vii. Let x be the class with the earliest start time, and let y be the course with the second earliest start time.
- If x and y are non-overlapping, choose x and recurse on everything but x (including y).
 - Else if x completely contains y , discard x and recurse.
 - Otherwise, discard y and recurse.

Solution: These next two are a little tricky - note that the reason they are fundamentally different from the previous ones is that they *process first, then select*, as opposed to the ones above which *select first, then process* (here, I'm largely using process to mean discard overlaps). However, as a result, the language of the exchange argument that will follow will look a bit different from the other ones. I'd consider these last two a little above the difficulty of what I'd ask on an exam.

We'll demonstrate this algorithm's correctness by means of an exchange argument. But first, a little note about this algorithm: consider the sequence of intervals $(x_1, x_2, \dots)$ which are selected as ' x ' in the above algorithm *leading up to a specific interval x being selected*. Note that by definition of our algorithm, a candidate for x can only be knocked out by the second rule - as a result, **each x_i is completely contained by the previous one**.

Now let's do the exchange argument. So we're comparing the solution produced by our algorithm to another valid solution S and we're looking at the first point of disagreement: namely, we've chosen interval A , yet their next earliest interval is B . I'm going to start out by ruling out any possibility that B ends earlier than A . Note that if B ended earlier than A , then **since every candidate x_i for x leading up to the algorithm's selection of A must have A** , none of them could have forced B 's discarding from the pool:

- if B started after a given x_i , then since it ends before A and x_i must contain A , it must be that both A and x_i entirely contain B , and so could not have knocked B out.
- if B started before a given x_i , then the only rule that could cause B to be knocked out would be the first one, but the algorithm *discards the interval which started*

later, which would not be B .

Either way, B would not be discarded. As a result, **B would still be in the pool when A is selected by our algorithm.** But this means that when our algorithm selected A , B was still in the pool - but this is also impossible, because either

- B started after A , but this would be impossible because A could only be selected if the next earliest interval still left started after A ended (and we are talking about the case where B ends before A does)
- or B started before A , but then A would not be the earliest starting interval when it was selected!

So all this is just to say that it's impossible for B to end before A , which means B must end after A .

From here out, we can just proceed with a standard exchange. We'll make a new solution S' that is identical to S except that it swaps out B for A . We know that

- A doesn't overlap with any earlier intervals because the earlier intervals in S / S' are exactly in agreement with ours and A cannot overlap with any earlier intervals in our solution by the definition of our algorithm
- A doesn't overlap with any later intervals in S / S' because the next interval in S must start after B ends, which is after A ends by the above arguments.

This tells us that S' is a valid solution. It also has exactly the same number of intervals as S since we only performed a swap. Hence S' is no worse than S and agrees with our solution for an additional step. As per usual, this is when we can declare victory.

- viii. If any course x completely contains another course, discard x and recurse. Otherwise, choose the class y that ends last, discard all classes that conflict with y , and recurse.

Solution: As in the previous part, this algorithm discards *before* selecting, so the argument gets a little complicated.

As per our standard exchange argument set up, let S be any valid solution for our given input, let A be the interval we select and B the interval they select on our first disagreement (note that we are processing from the right to the left on this algorithm). Similarly to the last problem, we will show that B must start before A . More specifically, we'll show that it's impossible for B to start after A . Suppose it did. We know that A is the interval that ends last that is still in the pool in our algorithm:

- If B is still in the pool when A was selected, then since A is the interval which ends latest we'd have that B would be entirely contained in A ; but our algorithm only makes selections after all containing intervals have been deleted, so this is impossible.
- If B isn't still in the pool when A was selected, then either
 - It was knocked out by an earlier interval we selected, but this is impossible because we only knock out overlapping intervals and B doesn't overlap with anything to the right in S which are precisely the things we selected earlier too!
 - It was knocked out in the pre-processing step that got rid of intervals which contain other intervals. In this case, look at the longest chain of intervals $I_1 = B, I_2, \dots, I_k$ where I_i knocked out I_{i-1} due to it being contained by it. So we have some interval I_k which is still in the pool at the start of the actual choosing steps of the algorithm, which is entirely contained in B . For the same reasons as above, it could not have been removed by an earlier interval that was selected, as it also doesn't overlap with any previously chosen intervals. But then,
 - * if I_k ends before A does then it is entirely contained by A (since I_k must start no earlier than B which is assumed to start after A), which can't happen because then A couldn't have been selected.
 - * if I_k ends after A , then A is not the latest ending interval left, which contradicts the definition of our algorithm.

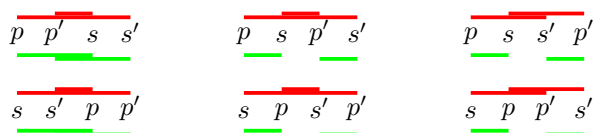
So the take-away is that B cannot possibly start after A . So this means B must start before A , which allows us to perform the exchange as usual: we make a new solution S' that swaps out B for A but keeps the rest of the intervals from S . It's uses the same number of intervals as S so it's no worse than S . It agrees with out solution for an extra step. And it's still valid - since B starts earlier than A , the next interval to the left of B in S ends earlier than A starts. As per usual, this is where we declare victory.

As a reminder, those last two were definitely harder than I would put on a midterm.

- iv) (Erickson 4.17) Suppose we have n skiers with heights given in an array $P[1..n]$, and n skis with heights given in an array $S[1..n]$. Describe an efficient algorithm to assign a ski to each skier, so that the average difference between the height of a skier and her assigned ski is as small as possible. (**Hint: You'll probably want to do some drawing for this one**)

Solution: The algorithm I propose is to assign the smallest ski to the smallest skier, then recurse on what's left.

We will show that it's better than any other valid solution S . Let p be the smallest person and s the smallest ski who are not assigned to each other in S ; let s' and p' be their assigned ski and person respectively. Consider all possible ways of ordering p, p', s , and s' (conditioned on $p < p'$ and $s < s'$):



In each way, I've recorded the matching of S in red: in green I've drawn a proposed adjusted solution S' where we match (s, p) and (s', p') , without adjusting any other matches. Although the drawings may not necessarily be drawn to scale, it doesn't really matter - in every single drawing, the sum of the red lines cannot be less than the sum of the green lines, no matter how you stretch the space (without breaking the ordering). This means that S' :

- is valid, as it assigns every skier a ski
- is no worse than S , as the swap cannot have increased the sum of distances
- agrees with our algorithm's solution for an additional step.

As per usual, this is where we can declare victory.

- viii) (AD 4.7) The wildly popular Spanish-language search engine El Goog needs to do a serious amount of computation every time it re-compiles its index. Fortunately, the company has at its disposal a single large supercomputer, together with an essentially unlimited supply of high-end PCs. They've broken the overall computation into n distinct jobs, labeled $J_1, J_2, \dots, J_n$, each of which can be performed completely independently of one another. Each job consists of two stages: first it needs to be preprocessed on the supercomputer, and then it needs to be finished on one of the PCs. Let's say that job J_i needs p_i seconds of time on the supercomputer, followed by f_i seconds of time on a PC. Since there are at least n PCs available on the premises, the finishing of the jobs can be performed fully in parallel - all the jobs can be processed at the same time. However, the supercomputer can only work on a single job at a time, so the system managers need to work out an order in which to feed the jobs to the supercomputer. As soon as the first job in order is done on the supercomputer, it can be handed off to a PC for finishing; at that point in time a second job can be fed to the supercomputer; when the second job is done on the supercomputer, it can proceed to a PC regardless of whether or not the first job is done (since the PCs work in parallel); and so on. Let's say that a schedule is an ordering of the jobs for the supercomputer, and the completion time of the schedule is the earliest time at which all jobs will have finished processing on the PCs. This is an important quantity to minimize, since it determines how rapidly El Goog can generate a new index. Determine the ordering which obtains a minimal completion time of jobs.

Solution: For this problem, let's schedule the longest jobs by PC processing time (the f value) first on the super computer.

Consider some other valid schedule S : we will use an exchange argument to show our solution finishes no later than the schedule of S . Consider the first point of disagreement - let J be the job we run at this time, let J' be the job that S runs *directly before* J in its schedule (note that I'm not necessarily talking about the job S runs at the time we run J , but rather the job that S runs right before it runs J later in its schedule, **which it must do because it is a valid schedule**). Let T be the time at which J' starts in their schedule. Then in the schedule of S :

- J' finishes at time $T + p(J') + f(J')$
- J finishes at $T + p(J') + p(J) + f(J)$

By definition of our algorithm it must be that $f(J) \geq f(J')$. Suppose we swap the order of J and J' in S : then now

- J' finishes at time $T + p(J) + p(J') + f(J')$ (which is earlier than when J finished before)
- J finishes at $T + p(J) + f(J)$ (which is earlier than its own finish time before)

Both jobs now finish before J had in the original schedule of S , which must have been no later than the original completion time of S . As a result, the completion time of S does not increase under this new schedule. By repeating this swap until J is run at the same time our algorithm runs it, we produce a schedule S' that

- is valid - all jobs eventually get run
- is no worse than S - as per the above argument, each swap downward for J doesn't increase the completion time
- agrees with our solution for an additional step.

As per usual, this is where we declare victory.

- xi) (AD 4.29) Given a list of n non-negative integers $d_1, d_2, \dots, d_n$, show how to decide in polynomial time whether there exists an undirected graph $G = (V, E)$ whose node degrees are precisely the numbers $d_1, \dots, d_n$. (That is, if $V = v_1, \dots, v_n$, then the degree of v_i should be exactly d_i .) G should not contain multiple edges between the same pair of nodes, or loop edges with both endpoints equal to the same node.

Solution: Consider the following algorithm:

```
def buildGraph(d[0, ..., n - 1]):
    initialize graph so that every vertex v[i] has no connections

    for _ in range(n):
        let v[i] be the vertex which has the most remaining connections to
                                build
        let d[v[i]] be the number of remaining connections to build for v[i]
        build a connection between v[i] and the next d[v[i]] vertices by
                                priority of remaining
                                connections to build
        if any degree requirements would need to be violated to satisfy v[i],
                                declare d to be
                                unsatisfiable

    declare d to be satisfied by our construction
```

Our algorithm prioritizes vertices which need the most connections, and greedily builds connections with other vertices, prioritizing vertices which still have the most remaining connections to build.

So if our algorithm is wrong, then that would mean that there is a construction of a graph $u[0], \dots, u[n - 1]$ so that the degree of each vertex $u[i]$ is equal to $d[i]$ (using our sort). **Note that this is a little different from how we do things, because usually we just compare ourselves to other valid solutions, but for this problem it's actually explicitly more helpful later on if we just compare ourselves to optimal solutions.**

First, let's observe that our algorithm builds 'valid' solutions in the sense that we never build a graph with a vertex whose degree is higher than it should be.

Additionally, observe that we 'visit' each vertex once to build connections from it - if we are unable to satisfy it on that round, we declare defeat. Otherwise, after 'visiting' a vertex, we have satisfied its degree requirement. As a result, in our exchange argument, let's 'order' the vertices in the order that our algorithm 'visits' them. So $v[0]$ is the vertex that has initially has maximal degree requirement, then $v[1]$ is the vertex that has the maximal degree requirement remaining *after processing* $v[0]$, and so on.

Look at the first disagreement between our algorithm and the connections made in u , specifically the first time we make an edge that u does not include. Note that if this never happens, then certainly u cannot make additional edges, because we always make edges whenever possible to satisfy degree requirements. So there must be some edge that we have made that u disagrees with. So let that edge be between $v[i]$ and $v[j]$ where at the time of creation we know that $v[i]$ has greater demand for edges than $v[j]$. Since $v[j]$ still needs to be satisfied, we know that $u[i]$ must have a connection to some other $u[j']$ where the demand remaining for $v[i]$ is at least the demand for $v[j]$ which must be at least the demand for $v[j']$ at the time of creation of edges from $v[i]$. As a result, because we know one of $u[j']$'s edges is going to $u[i]$, there must be some vertex $u[k]$ such that $u[k]$ is connected to $u[j]$ but not $u[j']$. So here's how we'll resolve our disagreement: build u' such that

- Keep all edges that are not between i, j, j' , and k
- Remove the edge between i and j'
- Add the edge from i to j
- Remove the edge from j and k
- Add the edge between j' and k .

As a result,

- No vertex's degree has changed
- No edges have been added 'twice' to a vertex - the whole disagreement was that there wasn't an edge between i and j so creating that edge wasn't a problem, and we picked k because it was a vertex that we *knew* wouldn't have an edge to j' .
- This graph agrees with our algorithm's construction for an additional step.

As a result, once again we can declare victory.

- xiii) (AD 1.6, kinda) For this problem, imagine you own a fleet of freight trains and an equal number of stations. Each one of these trains T_i has a schedule S_i for this month (you can assume months are arbitrarily long in this universe), during which every train visits every station exactly once. You've been informed by your engineers that every one of these trains has a problematic flaw in design and will need to be held in one of your stations for the rest of the month. These problems aren't exactly immediate - they can be held until the end of the month. Unfortunately, your stations aren't very big - each station can only hold one train, and another train cannot pass through the station while another is held there. You've been asked to truncate (end early) each train's schedule at a unique station: whatever this station is, it will remain there for the remainder of the month. Your goal is to do this so that:

- i. No other train passes through a station that holds another train
- ii. The number of stops made by all trains is maximized - equivalently, the number of stops lost due to truncation is minimized.

As an example, imagine that we had two trains and two stations, where the schedule was:

Train	Day 1	Day 2	Day 3	Day 4
T_1	S_1	.	S_2	.
T_2	.	S_1	.	S_2

If T_1 were truncated at S_1 , then T_2 wouldn't be able to pass through the station, and thus no valid truncation would follow. Hence T_1 must be truncated at S_2 , meaning T_1 visits both S_1 and S_2 while T_2 must truncate at S_1 , losing its visit to S_2 .

Determine an algorithm that successfully achieves your goals. (**Hint: How is this just a Stable-Marriage problem?**)

Solution:

Solution: We're going to create an input for the DA algorithm, run the DA algorithm, and use the assignment produced to choose our truncations. Specifically, we will equate the set of men as the stations and the set of women as the trains. For each man/station, we create rankings over the women/trains according to reverse chronologic order (so each station prefers most the train that arrives last, etc.). For each woman/train, we create rankings over the men/stations according to chronologic order (so each train most prefers the station it visits first). After running DA and getting the assignment M it produces, we will truncate each train's schedule at the station it gets assigned to. Suppose in the schedule produced that a train t visits a station s after another train has already been truncated at that station. Then s prefers t over its assigned partner, and t prefers s over its assigned partner: this means that (s, t) are a 'rogue couple' in M , so M would be unstable. Note that the reverse also holds: that is, 'rogue couples'/instability is equivalent with the goal (a) above being violated. Since we know that DA produces stable solutions, we know that the goal (a) is achieved by our schedule. Moreover, we know that DA produces male-optimal solutions: every station ends up holding the train which visits last over every stable assignment, ie over every assignment that satisfies goal (a). Hence there is no way to come up with an assignment which satisfies goal (a) where any station gets assigned a train which visits later. As such, the maximal number of stops are made by this truncated schedule.