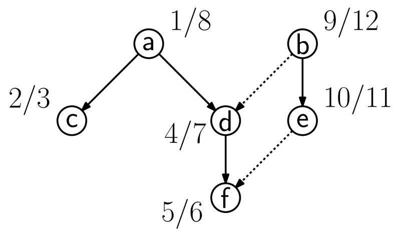
Solutions to the Midterm Exam

Solution 1: See Fig. 1. Edges (b, d) and (e, f) are cross edges. The rest are tree edges.



Topological order: ⟨b, e, a, d, f, c⟩

Figure 1: Depth-first search and topological sorting.

Solution 2:

(a) $\Theta(n^3)$: If you grind through the summations it comes out to roughly $n^3/6$. This is the same loop structure as the DP for chain-matrix multiplication.

(b) For a graph with n vertices and n edges:

- (i) This graph must be connected - False
- (ii) This graph must have at least one cycle - True (A tree has $n - 1$ edges and this is the greatest number of edges you can have without creating a cycle)
- (iii) The average vertex degree is $O(1)$ - True (The total degree is twice the number of edges, so the average degree is exactly 2)

(c) Dijkstra's algorithm on a directed graph with a negative cost cycle:

- (i) It may generate incorrect distance values - True
- (ii) It may go into an infinite loop - False (Dijkstra processes each vertex exactly once)
- (iii) It may abort due to indexing an array out of bounds - False

(d) Optimal for interval scheduling?

- (i) EFF (Earliest finish-time first) - Yes
- (ii) SDF (Shortest duration first) - No (But this yields a factor-2 approximation)
- (iii) FCF (Fewest conflicts first) - No (There is a counterexample)
- (iv) LSL (Latest start-time last) - Yes (This is the same as EFF on the reversed input)

(e) (ii) $\text{dist}(c_1, c_2) \geq \text{diam}(P)/2$. Given the first two points c_1 and c_2 of Gonzalez's algorithm, the point c_2 is chosen to be the farthest point from c_1. This implies that all the points lie within a ball of radius $r = \text{dist}(c_1, c_2)$ from c_1. The diameter of this ball, and hence the diameter of P is at most twice this radius.

(f) In Floyd-Warshall, the value k in $d^{(k)}(i,j)$, means that the shortest path can travel through a subset of the vertices $\{1, 2, \ldots, k\}$.

(g) Pushing flow along a forward edge (u,v), increases the flow on (u,v). Pushing flow along a backwards edge (v,u), decreases the flow on (u,v).

Solution 3:

(a) The network G' is the same as G, but with the following modifications. For each vertex v with a nonzero leakage capacity $h(v)$, we create an edge (v,t) and assign it capacity $h(v)$ (see the dashed edges in Fig. 2(b)). (This is a multi-digraph, since we may generate duplicate edges, but we can always merge edges and combine their capacities.)

To see that this is correct, suppose that there is a leaky flow f in G. For each vertex v, let $g(v) = f^{\text{in}}(v) - f^{\text{out}}(v)$ denote the amount of leakage. By definition of leaky flows, we know that $0 \le g(v) \le h(v)$. We assign flow of $g(v)$ to the edge (v,t) (see Fig. 2(d)). Call the resulting flow f' in G'. This additional flow compensates for the leaked flow at u, implying that f' satisfies flow conservation. Since the leakage cannot exceed $h(v)$, this flow satisfies the capacity constraint. The converse is similar.

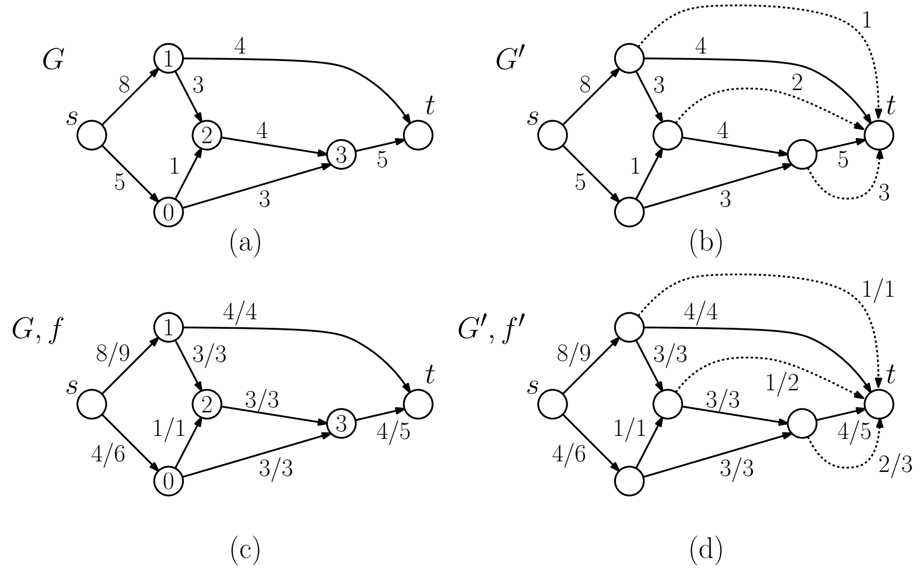


Figure 2: (a) A leaky network G, (b) the transformed (non-leaky) network G', (c) a leaky flow in G, and (d) the equivalent non-leaky flow in G'.

(b) See Fig. 2(b).

Solution 4:

(a) A counterexample is shown in Fig. 3(a). The optimal solution consists of two pins (see Fig. 3(b)). The depth-based algorithm places a pin in the center, which has the highest depth of four. It still needs to place two more pins to cover the leftmost and rightmost intervals (see Fig. 3(c)).

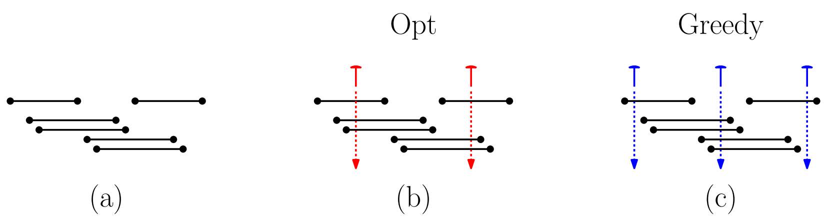


Figure 3: Counterexample to the depth-based heuristic.

(b) depth(I)/opt$(I) \leq \ln n$: This is essentially equivalent to the greedy set cover algorithm. We can restrict placement of pins to the n right endpoints of the intervals. (Any valid pin placement can be slid to the right until it hits a right endpoint.) Let B denote the set of right interval endpoints. Consider a set system (X, Σ) where the elements are intervals $(X = I)$, and for each right endpoint $b \in B$, we define set $S(b)$ to be the intervals that are stabbed by b. Let Σ denote this collection of sets.

A stabbing set is equivalent to computing a minimum-sized set of right interval endpoints $b \in B$ (pins) that stab all the intervals of I. From the perspective of our set system, this is equivalent to computing a minimum-sized collection of sets $S(b) \in \Sigma$ that cover all elements $X = I$. These two problems are equivalent, and depth-based pinning heuristic is clearly equivalent to the greedy set cover heuristic. We know that the approximation ratio of the greedy heuristic for set cover is $\ln n$, where $n = |X| = |I|$.

(c) The greedy solution is based on the idea of "delaying" the placement of each pin as far as possible to the right. To begin, sort the intervals by their right endpoints $b_1 < \cdots < b_n$. We visit the intervals in this order. Place a pin at the first endpoint in the list (initially at b_1). Since this interval must contain a pin, this is the farthest right we can place this first pin. Now, remove all the intervals that are hit by this pin, or equivalently, all those whose left endpoint appears on or before b_i (see Fig. 4(b)).

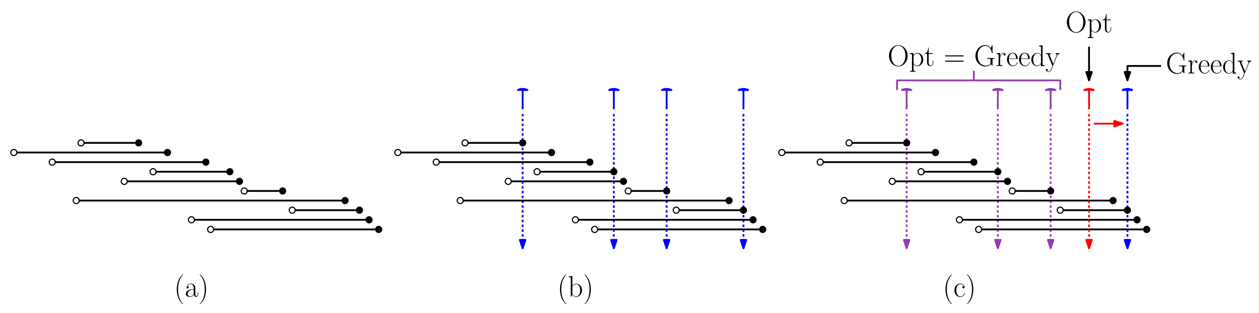


Figure 4: Greedy algorithm for 1-dimensional pinning.

To see that this algorithm is correct, observe that each time we place a pin, we eliminate exactly those intervals that are stabbed by this pin, therefore every interval will be stabbed when the algorithm terminates. To see that this is optimal, consider list in pins in order from left to right. By its greedy nature, the greedy solution puts the each pin in the farthest right position possible. Thus, if greedy differs from opt, in their first difference, opt's pin will lie to the left of greedy's pin (see Fig. 4(c)). We can slide this pin of opt to the right as far as possible, until it hits the greedy pin. Since greedy is valid, the result of this sliding will also be a valid solution. By repeating this, eventually opt will be turned into greedy, without adding any more pins, and therefore greedy and opt use the same number of pins.

The DP algorithm is also based on sorting by right endpoints. For $0 \leq i \leq n$, define $P(i)$ to be the minimum number of pins needed to stab the first i intervals in this order. For the basis case, $P(0) = 0$. In general, to compute $P(i)$, we place a pin at the farthest left location, namely a_i. Define pred$[i]$ to be the largest index j such that $b_j < a_i$ (or 0 if no such index exists). All the intervals from $j+1$ through i are stabbed by the pin at a_i. Thus, all that remains are to stab the first pred$[i]$ intervals, whose value is just $P(\text{pred}[i])$. This leads to the following DP formulation:

$$P(i) = \begin{cases} 0 & \text{if } i = 0 \\ 1 + P(\text{pred}[i]) & \text{otherwise} \end{cases}$$

(This formulation has the property that it generates pins as far to the left as possible, while the greedy places pins as far to the right as possible. Clearly, these two strategies are just mirror images of each other. Even though it is written in DP form, it is effectively a greedy algorithm.)

Solution 5: We assume we have access to a utility function len(i,j), which, for $1 \leq i \leq j \leq n$, returns the sum of word lengths $\sum_{k=1}^{j} w_k$. Our solutions are based on a recursion that works down from n, but there are symmetrical solutions where the recursion works up from 0.

(a) (This is identical to the problem from Homework 3, except that the cost function is different.) For $0 \leq j \leq n$, let MG(j) denote the minimum value of the max-gap for typesetting the first j words. For the basis case we have MG$(0) = 0$, since there are no words and hence no penalty. To compute MG(j) for $j \geq 1$, let us hypothesize that some word w_i is the first word of the line for $1 \leq i \leq j$. Clearly, len$(i,j) \leq L$. This line has $j - i + 1$ words. The gap size for this line is gap(i,j), where

$$\text{gap}(i,j) = \begin{cases} 0 & \text{if } i = j \\ \dfrac{L - \text{len}(i,j)}{j - i + 1} & \text{otherwise.} \end{cases}$$

We should take the best possible segmentation of the prior $i - 1$ words, which has a max gap of MG$(i-1)$. Among the available options, we select the one that produces the lowest value. Thus, we have the following formulation:

$$\text{MG}(j) = \begin{cases} 0 & \text{if } j = 0 \\ \displaystyle\min_{\substack{1 \leq i \leq j \\ \text{len}(i,j) \leq L}} \max\left(\text{gap}(i,j), \text{MG}(i-1)\right) & \text{otherwise.} \end{cases}$$

The overall answer is MG(n).

The above solution involves a 1-parameter function, but requires a loop to determine the best split. Another approach (which yields the same running time, but takes more space) is based on a 2-parameter function. For $1 \leq i \leq j \leq n$, let $\mathrm{MP}'(i,j)$ denote the smallest achievable max-gap for typesetting the first j words, under the assumption that the last line starts *on or before* w_i. The final answer will be $\mathrm{MP}'(n,n)$. To avoid dealing with cases where the total word length exceeds the line length, let us define

$$\mathrm{penalty}(i,j) \;=\; \begin{cases} L - \mathrm{len}(i,j) & \text{if } \mathrm{len}(i,j) \leq L \\ \infty & \text{otherwise} \end{cases}$$

For the basis case, observe that if $i = 1$, then we are putting all the words on a single line, and the overall penalty is $\mathrm{penalty}(i,j)$. Otherwise, $i \geq 2$. Either the last line starts with w_i, in which case the penalty for this last line is $\mathrm{penalty}(i,j)$. The remaining subproblem is to typeset the first $i-1$ words, which is $\mathrm{MP}'(i-1,i-1)$. The overall max-penalty is the maximum of these two. Otherwise, the last line starts earlier than w_i (that is, on or before w_{i-1}), in which case the overall penalty is given by $\mathrm{MP}'(i-1,j)$. As always, we take the better of these two options.

$$\mathrm{MP}'(i,j) \;=\; \begin{cases} \mathrm{penalty}(i,j) & \text{if } i = 1 \\ \min(\max(\mathrm{penalty}(i,j), \mathrm{MP}'(i-1,i-1)), \mathrm{MP}'(i-1,j)) & \text{otherwise.} \end{cases}$$

Note that once $\mathrm{len}(i,j)$ exceeds L, $\mathrm{MP}'(i,j)$ will be ∞, thus if we were to implement this, we could add this additional check to avoid unnecessary recursive function calls.

If implemented, both solutions would take $O(n^2)$ time.

Here is yet another DP solution. In a manner analogous to matrix multiplication, we define $\mathrm{MP}''(i,j)$ to be the smallest achievable max-gap for typsetting words w_i through w_j. If the words fit on a single line, we take the penalty. Otherwise, the DP looks for the best intermediate word w_k about which to split the line.

$$\mathrm{MP}''(i,j) \;=\; \begin{cases} \mathrm{penalty}(i,j) & \text{if } \mathrm{len}(i,j) \leq L \\ \min_{i \leq k \leq j-1}(\max(\mathrm{MP}''(i,k-1)), \mathrm{MP}''(k,j)) & \text{otherwise.} \end{cases}$$

If implemented, this would take $O(n^3)$ time.

(b) Two approaches to this problem come to mind. The first is to augment the DP formulation of part (a) to account for line numbers. The second is slower, but it is quick and easy to describe. It is based on just "guessing" where to the split the text, and applying the solution to part (a) to each of the parts, one with line length L_1 and the other with L_2.

Let's describe the DP first. The idea is to use an additional parameter that stores the line number, call it ℓ. Unfortunately, our formulation, which recurses down from n, is difficult to adapt, because we don't know what the final line number will be. Instead, let's recurse upwards. For $0 \leq i \leq n$ and $\ell \geq 1$, let $\mathrm{MG}(i,\ell)$ denote the minimum value of the maximum gap for typesetting words w_i through w_n words assuming that w_i will be placed on line ℓ. The initial call is $\mathrm{MG}(1,1)$. The formulation is similar to part (a), except that we search forward from i and increment ℓ in our recursive calls. Also, let's define $L(\ell)$ to be L_1 if $\ell \leq 3$ and L_2 otherwise. Define $\mathrm{gap}(i,j,L(\ell))$ to be the same gap function, but the third parameter gives the line length to use.

$$\mathrm{MG}(i,\ell) \;=\; \begin{cases} 0 & \text{if } i > n \\ \displaystyle\min_{\substack{i \leq j \leq n \\ \mathrm{len}(i,j) \leq L(\ell)}} \max\left(\mathrm{gap}(i,j,L(\ell)), \mathrm{MG}(i+1,\ell+1)\right) & \text{otherwise.} \end{cases}$$

If implemented, the running time will be essentially the same as the part (a) solution, that is, $O(n^2)$. (By the way, we could have used our count-down recursion, but we would need to "guess" the final number of lines. Since this quantity is at most n, we just try all the possibilities, but this would increase the running time by a factor of n.)

The second approach is much easier to describe. Observe that there is one decision to be made, namely which is the last word w_j to appear on the third line. Following the DP-credo, let's just try all possible choices for j, $1 \leq j \leq n$. For any candidate value j, we invoke the solution from part (a) on the words $\langle w_1, \ldots, w_j \rangle$ using line length L_1, and then we invoke on the remaining words $\langle w_{j+1}, \ldots, w_n \rangle$ using L_2. We will take the maximum of the two solutions. In order to enforce the 3-line condition, we will check that the solution produced by the first invocation fits onto three lines. (We will need to add some hooks to record this information.) If not, we reject this value of j from the list of candidates.

This is not super efficient. It necessitates invoking (a) up to n times, so the running time will be worse than (a) by a factor of $O(n)$, which will be $O(n^3)$.