CMPS102: Homework #5 Solutions

1 The Rabbit Problem

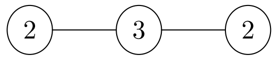
Suppose the rabbit wants to go through a distance of $i > 3$ feet. Consider the last hop; it can either be a short one (1 foot) or a long one (3 feet). If it is a short hop, then the rabbit must be at position $i - 1$ before the hop. If, however, it is a long one, the the rabbit must be at position $i - 3$ before the hop. Thus $f(i) = f(i-1) + f(i-3)$ for $i > 3$. Also note that $f(1) = f(2) = 1$ since the rabbit can go through the distances of 1 and 2 with only short hops. Furthermore $f(3) = 2$ since it can either take a long hop or three short ones. Therefore:

$$f(i) = \begin{cases} 1 & i \leq 2 \\ 2 & i = 3 \\ f(i-1) + f(i-3) & i > 3 \end{cases}$$

The values of $f(\cdot)$ are stored in a one-dimensional array $F[1 \ldots n]$. The algorithm should fill in the values in order of increasing i. So the first three cells are computed by the initialization, then using the recurrence we can find all the values from left to right ($F[n]$ will be computed last). Since we are filling in an array of size n each cell of which takes a constant time, the total time complexity is $O(n)$.

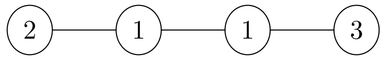
2 Kleinberg & Tardos, Ch. 6, #1

(a) The "heaviest-first" algorithm *does not* find an independent set of maximum weight on this graph:



The algorithm will first choose the node with weight 3 and eliminate both of its neighbors. Clearly, choosing the other two nodes yields a higher weight independent set.

(b) The "either odds or evens" algorithm *does not* find an independent set of maximum weight on this graph:



The algorithm will choose the odds ($\{1, 3\}$) while $\{2, 3\}$ is clearly a better choice.

(c) Solution 1

Consider this dynamic programming algorithm for finding a maximum weight independent set on a path.

Optimal Substructure Nodes in the path a denoted e_1 to e_n. If S is a maximal independent set for n nodes and it included the last node, then $S - e_n$ is a maximal independent set of the first $n - 2$ nodes. Otherwise (if e_n was not in S), then S is a maximal independent set of the first $n - 1$ nodes.

Recurrence $S(k)$ represents the weight of a maximal independent set of nodes from the path of nodes from e_1 to e_k. The general case of this recurrence decides between including the e_k (and a maximal set of the $k - 2$ lower, non-neighbor nodes) or not including e_k (and reusing the maximal set from the previous $k - 1$ nodes).

$$S(k) = \begin{cases} 0 & \text{if } k < 1 \\ \max\left(S(k-1), S(k-2) + w_k\right) & \text{otherwise} \end{cases}$$

Table The algorithm should fill a n-element table $S[k]$ with the a maximal weight of an independent set over nodes left-of and including each e_i.

Filling Strategy The algorithm should apply the above recurrence, in ascending order (so $S[1]$ is filled first and $S[n]$ last). The table will provide pre-computed values for $S(k)$ at each step, allowing each element to be processed in constant time.

Solution Recovery In a post-processing step, the elements of $S[\cdot]$ can be considered in reverse order (from high to low) to recover the set of elements used to achieve the maximal weight. If $S[k] = S[k - 1]$ then the e_k is not in the independent set (and $k - 1$ should be considered next), otherwise it is (and $k - 2$ should be considered next).

Running Time Filling the n element table using constant-bounded work at each cell and then performing a final $O(n)$ time post-processing step allows the complete algorithm to run in $O(n)$ time.

Solution 2

Consider this alternative dynamic programming algorithm for finding a maximum weight independent set on a path.

Optimal Substructure Suppose S is a maximal independent set for n nodes. S must include either e_n or e_{n-1} (not both). Once S includes a node, it cannot include the node right before, but it must include exactly one node from the two nodes that appear 2 and 3 steps back.

Recurrence $S'(k)$ represents the weight of a maximal independent set of nodes from the path of nodes from e_1 to e_k which also includes the kth node. The general case of this recurrence decides between including the e_{k-2} (and a maximal set of the $k-2$ lower) or e_{k-3} (and a maximal set of the $k-3$ lower).

$$S'(k) = \begin{cases} 0 & \text{if } k < 1 \\ w_k + \max\left(S'(k-2), S'(k-3)\right) & \text{otherwise} \end{cases}$$

Note that S' in Solution 2 is related to S in Solution 1:

$$\text{For all } 1 \le k \le n \quad S(k) = \max(S'(k), S'(k-1))$$

Thus the final answer for Solution 2 will be $\max(S'(n), S'(n-1))$.

Table The algorithm should fill a n-element table $S'[k]$ with the a maximal weight of an independent set over nodes left-of and including each e_i.

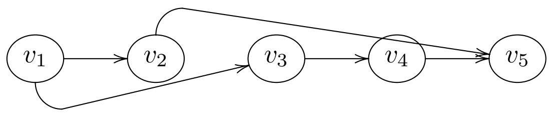
Filling Strategy The algorithm should apply the above recurrence, in ascending order (so $S'[1]$ is filled first and $S'[n]$ last). The table will provide pre-computed values for $S'(k)$ at each step, allowing each element to be processed in constant time.

Solution Recovery In a post-processing step, the elements of $S'[\cdot]$ can be considered in reverse order (from high to low) to recover the set of elements used to achieve the maximal weight. If $S'[k] = S'[k-2] + w_k$ then the e_{k-2} is in the independent set (and $k-2$ should be considered next), otherwise e_{k-3} is (and $k-3$ should be considered next).

Running Time Filling the n element table using constant-bounded work at each cell and then performing a final $O(n)$ time post-processing step allows the complete algorithm to run in $O(n)$ time.

3 Kleinberg & Tardos, Ch. 6, #3

(a) The "small as possible" algorithm *does not* find the a longest path on this ordered graph:



The algorithm will choose edge $\{v_1, v_2, v_5\}$, while $\{v_1, v_3, v_4, v_5\}$ is clearly a longer path (and the correct answer).

(b) Consider this dynamic programming algorithm for finding the longest path in an ordered graph.

Optimal Substructure The longest path through the first node in ordered path is one edge longer than the longest path through any node to which the first node is directly connected.

Recurrence

$$L(k) = \begin{cases} 0 & \text{if } k = n \\ -\infty & \text{if } v_k \text{ has no outgoing edges} \\ \max_{c \in \text{children}(k)} (1 + L(c)) & \text{otherwise} \end{cases}$$

Table The algorithm should fill in a n element table $L[\cdot]$ recording the length of the longest path from v_k to v_n and which node won the max expression.

Filling Strategy The algorithm should fill in the table in decreasing order (so $L[n]$ is processed first and $L[1]$ last). This order ensures that each application of the recurrence will only inspect pre-computed values from the array in $O(n + m)$ total time for an ordered graph with n nodes and m edges.

Solution Recovery While $L[1]$ encodes the length of the longest path from v_1 to v_n, we should apply a linear time post-processing step to recover the literal elements of the longest path. Starting at v_1, we should consult the table to follow which node won each max expression, eventually reaching v_n. If $L[1] \leq 0$ then v_1 was not connected to v_n.

Running Time As previously described, both the table-filling and solution recover run in $O(n + m)$ time.

4 Kleinberg & Tardos, Ch. 6, #6

This problem is very similar to segmented least squares problem. Consider this dynamic programming algorithm to decide how to pretty print lines, minimizing the sum squared line slacks. For convenience, let $f(i, k)$ represent the cost of placing words $i + 1$ through k on a single line:

$$f(i,k) = \begin{cases} \infty & \text{if words } i+1 \text{ through } k \text{ do not fit} \\ \left(L - \left[\sum_{j=i+1}^{k-1} (c_j + 1)\right] + c_k\right)^2 & \text{otherwise} \end{cases}$$

Optimal Substructure The optimal pretty printing of a paragraph includes an optimal pretty printing of words before the last line followed by the last line (similarly for splitting off the first line).

Recurrence $C(k)$ is a recurrence for the cost of an optimal pretty printing of words 1 through k. The value $C(n)$ represents the cost of pretty printing the entire paragraph.

$$C(k) = \begin{cases} 0 & \text{if } k = 1 \\ \min_{1 \leq i < k} (C(i) + f(i,k)) & \text{otherwise} \end{cases}$$

Table The algorithm should fill in a n element array $C[\cdot]$.

Filling Strategy In order of increasing k, the algorithm should apply the recurrence to each cell, recalling table values instead of applying the recurrence for smaller word ranges.

Solution Recovery While $C[n]$ represents the cost of an optimal pretty printing, a $O(n)$ post-processing step is needed to recover the line breaks from the table by examining which split won the min expression (which could be stored in the array to ease this process).

It is worth noting that $f(i,k)$ takes $\Theta(k-i)$ (which is $O(n)$) time to compute in isolation. Applying it as-is, the algorithm (even with memoization of $C[\cdot]$) would yield a $\Omega(n^3)$ running time (considering the loop over k in filling the table, the loop over i in computing a single cell, and the loop over a j in the implementation of $f(i,k)$). This is pretty bad!

A $\Theta(n^2)$ algorithm can be used to tabulate values of $f(\cdot,\cdot)$ to allow constant time recall in the larger algorithm, resulting in a $\Theta(n^2)$ running time overall.

Because the cost of a line quickly hits the infinite case as more words are considered, many values of i in the min are needlessly considered. Observe that L itself is an upper bound on the number of words that can fit on a line (imagine working with only zero-length words, L spaces alone would fill the line). As a result, certain ranges in the algorithm could be reduced to account for this constant bound on feasible line lengths. Pre-computing $f(i,j)$ could be done in $\Theta(n)$ time and, using the result, each min operation could be done in $\Theta(1)$ time. Thus, we can trade one of the factors of n in the running time for a constant to achieve an attractive $\Theta(n)$ running time using essentially the same recurrence and filling strategy as described above!

5 Kleinberg & Tardos, Ch. 6, #24

In the gerrymandering problem we are asked, essentially, to say (yes or no) whether there is a partitioning of n equal-sized precincts (each with a different balance of supporters for two parties) into 2 districts such that a particular party has a majority in both. We focus on party A and assume that n and the size m of each precinct is even.

Let a be the total number of supporters for party A. Clearly for A to have the majority in both precincts we need $a > \frac{nm}{\mathbf{2}} + 2$ because it has to have at least $\frac{nm}{\mathbf{4}} + 1$ in each precinct.

More precisely, A wins both precincts if there is a subset of $n/2$ precincts for which its total vote count s is at least $\frac{nm}{4} + 1$ and the in the complimentary set of precinct party A also has at least $\frac{nm}{4} + 1$ votes, i.e. $a - s \geq \frac{nm}{4} + 1$ as well. This means gerrymandering is possible iff $\frac{nm}{4} < s < a - \frac{nm}{4}$.

Focusing on the concrete problem of selecting subsets of from a particular range of elements, of a particular count, totaling to a particular value, we can use a dynamic programming formulation similar to the simple knapsack problem (where we are looking for a subset of elements which totals to a specific value exactly). Note that, in contrast to the rest of the problems in this homework, this is a decision problem as opposed to an optimization problem. This means that our recurrence will be operating over Boolean values, yes or no, as to whether certain subsets can exist.

Problem Substructure Among the precincts 1 to q, there exists a subset of k precincts with total weight s if among the precincts 1 to $q-1$, there exists a subset of $k-1$ precincts with $s-a_q$ total weight assuming that precinct q has weight (number of supporters) a_q. Alternatively, a subset of the same parameters (q, k ,s) exists if over the smaller set of elements 1 to $q-1$, if there is an equal size subset with the same total weight.

Recurrence $T(q,k,s)$ is a recurrence over precincts from P_1 through P_q for the existence (a boolean) of a subset S of size k for which the total number of included precincts has a total number supporters $s = \sum_{i \in S} a_i$.

$$T(q,k,s) = \begin{cases} \text{true} & \text{if } q = 1,\ k = 1, \text{ and } s = a_1 \\ \text{true} & \text{if } T(q-1,k-1,s-a_q) \\ \text{true} & \text{if } T(q-1,k,s) \\ \text{false} & \text{otherwise} \end{cases}$$

Alternatively, the recurrence can be expressed in symbolic logic.

$$\begin{aligned} &T(1,1,a_1) \\ &T(q,k,s) \leftarrow T(q-1,k-1,s-a_q) \\ &T(q,k,s) \leftarrow T(q-1,k,s) \end{aligned}$$

Table The algorithm should fill in a n by $n/2$ by $a - \frac{mn}{4} - 1$ table T.

Filling Strategy In order of increasing q, and any order for k and s, the algorithm should fill in the table applying the rules above.

Solution Recovery Having filled the table completely, what remains is to check all of the values in the row $T[n, \frac{n}{2}, s]$ for all $\frac{nm}{4} < s < a - \frac{nm}{4}$. If any of these elements is marked true then the problem instance is gerrymanderable.

Running Time Assuming constant-time lookups for a_q given q, then the whole table can be filled with a constant-bounded amount of work per entry. This leads to a $O(n^3m)$ bound ($n \times \frac{n}{2} \times a - \frac{mn}{4} - 1 < (n)(n)(nm) = n^3m$). Adding a $O(nm)$ traversal to check for values on a final row does not affect the asymptotic running time, so $O(n^3m)$ is a bound for the whole gerrymandering-susceptability algorithm.

6 Kleinberg & Tardos, Ch. 6, #28

(a)

Claim. A no-lateness scheduling problem has an optimal solution J (i.e., a set of maximum size) in which the jobs in J are scheduled in increasing order of their deadlines.

Proof. We will show, by gradual adjustment, that any optimal solution that does not schedule jobs in increasing order of their deadlines can be transformed into a solution that does order them this way without destroying their optimality.

Suppose we have an optimal solution K which in which jobs are not scheduled in order. Because the jobs are out of order, we can always find a point in the schedule in which two jobs are inverted (that is $j < k$ and $d_j > d_k$). In general, an unordered schedule will have many such "inversions".

Consider swapping the position of jobs j and k in the schedule. The resulting schedule is still feasible because the originally-later job (k) is only done sooner and the originally-earlier (j) job is still done within its deadline (it is now finished by d_k which was actually sooner than required). The resulting schedule is still of maximal size because we have not de-scheduled any jobs. This swap has reduced the number of inversions in K by at least one.

Continuing this swapping operation, we can remove all inversions in an unordered optimal solution, yielding an optimal ordered schedule. Thus the desired ordered optimal schedule J always exists. □

(b) Consider this dynamic programming algorithm for finding a maximal schedulable set of jobs with no lateness. It will produce solutions ordered by increasing deadline (assembly schedules in a latest-deadline-last fashion).

Optimal Substructure If J is a maximal schedule for k jobs and it included the last job using a total work time of t, then $J - \{n\}$ is a maximal schedule of $k-1$ jobs finishing withing $t - s_k$. If the last job (k) was not included, clearly J is also a maximal schedule for the $k-1$ previous jobs finishing in the same total time t.

Recurrence $W(k,t)$ represents the maximum number of jobs scheduled by an optimal scheduling of jobs 1 to k using at most t working time. $W(n,D)$ represents the total work time for a schedule for all of the jobs in the original problem.

$$W(k,t) = \begin{cases} 0 & \text{if } k = 0 \\ \max\{W(k-1,t), W(k-1,t-s_k)+1\} & \text{if } d_k < t \text{ and } t - s_k \geq 0 \\ W(k-1,t) & \text{otherwise} \end{cases}$$

Table The algorithm should fill a table $W[k,t]$ of size n by D.

Filling Strategy The algorithm should fill the table in increasing order of k (and then by t). No initialization is necessary as the recurrence includes special cases for both $k = 0$ and $t < 0$ (for which all jobs are infeasible).

Solution Recovery While $W[n, D]$ represents the time taken for an maximal schedule of all n jobs, a $O(n)$ time post-processing step should recover the set of scheduled jobs from the table by following which term won the max expression.

Running Time Given concrete values for k and t, it is possible to determine if job k is feasible in constant time. Thus, the overall running time of the algorithm is $O(nD)$ (including the $O(n)$ post processing).