CS 577-2: Introduction to Algorithms Fall 2019

Solutions to the Final Practice Problems

Instructor: Dieter van Melkebeek

Problem 1

Define $OPT(i, j)$ to be the side length of the largest square ending at (i, j). To find the largest square ending at (i, j). If (i, j) is 1, we just take the minimum size of squares ending at $(i-1, j-1)$,$(i-1, j)$ and $(i, j-1)$, and add 1. This is because the coverage of the new square cannot exceed the squares which are to the top, left, and left-top side of the grid (i, j).

Then we have

$$OPT(i,j) = \begin{cases} 0, & table[i][j] = 0 \\ 1, & (i = 1 \text{ or } j = 1) \text{ and } table[i][j] = 1 \\ min(OPT(i-1, j-1), OPT(i-1, j), & \\ \quad OPT(i, j-1)) + 1, & i > 1 \text{ and } j > 1 \text{ and } table[i][j] = 1 \end{cases}$$

We may maintain a variable $mlen$ recording the maximum side length we have ever encountered. Since we want to get the largest size, the final result is $mlen^2$.

To compute each entry in the table OPT, it takes constant time. Since there are $O(n^2)$ entries to compute, it takes $O(n^2)$ time in total.

Moreover, since we are using only the previous row and the current row to compute each entry, we may reduce the space complexity from $O(n^2)$ to $O(n)$ by saving only two rows.

Problem 2

This problem can be solved with a dynamic programming approach. Let us assume a coordinate system with $(0, 0)$ in the top left and row-major order. Let (x, y) denote the location of the white square. Note, here, x is rows (up and down) and y is columns (left and right). Each DP state will be a possible state of the game. Let $\text{Win}(u, d, l, r)$ denote whether the sub-rectangle with (u, l) in its upper left corner and (d, r) in its lower right corner is a winning state of the game for the current player. With this formulation, the answer will be $\text{Win}(0, n-1, 0, m-1)$.

As a base case we have:

$$\text{Win}(x, x, y, y) = \texttt{true}.$$

Then our recurrence is:

$$\begin{aligned}\text{Win}(u,d,l,r) = & \bigvee_{u<i<x} \neg Win(i,d,l,r) \\ & OR \bigvee_{x<i<d} \neg Win(u,i,l,r) \\ & OR \bigvee_{l<j<y} \neg Win(u,d,j,r) \\ & OR \bigvee_{y<j<r} \neg Win(u,d,l,j),\end{aligned}$$

where $\bigvee_{a<i<b}$ denotes an OR over all indices from a to b. The recurrence can be evaluated top-down or in increasing order of $d-u$ and $r-l$.

Correctness Each DP state denotes a possible state of the game because in each move Alice or Barbara breaks the current rectangle into two rectangles along an integer point and gives her opponent the rectangle that contains the white square. In this way, we only must keep track of the subrectangle of the original rectangle that contains the white square.

Furthermore, note that since each player is playing optimally, their goal will be to arrive at a state such that there is a strategy that can guarantee victory no matter what the opposing player does. With our state formulation, either Alice or Barbara can arrive at the same state, but since their goals and moves are symmetric, they will employ the same strategy. Further note that each state is either winning for the current player or winning for the opposing player since there are no ties. In this way, the state description captures all components necessary to determine winning strategies.

For a given player's move, then, we can just check all possible moves and see if they end up in a winning state for the opposing player. If not, this means it is a winning state for the current player, in which case we can take that move and the value of the current state should be `true`. Otherwise, if all moves result in a winning state for the opposing player, the value of the current state should be `false`.

Runtime In our DP state, u and d range from 0 to $n-1$ and l and r range from 0 to $m-1$; therefore, there are $O(n^2m^2)$ possible states. Each state varies the parameter l, r, u and d independently, taking $O(n)$ time to vary u and d and $O(m)$ time to vary l and r. Therefore each state takes $O(n+m)$ time to compute. In total, the algorithm then runs in $O(n^2m^2(n+m))$ time.

Problem 3

Part(a)

Fix an instance consisting of the sequence $p_1, p_2, \ldots, p_n$ of integers and the integer k. For integers b, f, and ℓ with $1 \le b \le f \le n$ and $1 \le \ell \le k$, let $\text{OPT}(b, f, \ell)$ denote the solution for the subinstance involving the positions in $[b, f]$ and ℓ intervals, i.e., the maximum value of $\sum_{i=1}^{\ell} \text{val}(x_i, y_i)$ over all choices of ℓ pairwise disjoint subintervals (x_i, y_i) of (b, f), where $\text{val}(x, y) \doteq p_y - p_x$. The answer to the given instance is $\text{OPT}(1, n, k)$.

The OPT-entries with $\ell = 1$ are essentially given by the blackbox. To be precise, let us denote by $\mathrm{BB}(b, f)$ the interval (x, y) that the blackbox returns on input (b, f). We have that $\mathrm{OPT}(b, f, 1) = \mathrm{val}(\mathrm{BB}(b, f))$.

To compute the OPT-entries for values $\ell > 1$, we make use of the following recurrence:

$$\mathrm{OPT}(b, f, \ell) = \max_{b \le m \le f} \left(\mathrm{OPT}(b, m, \ell - 1) + \mathrm{val}(\mathrm{BB}(m, f))\right). \tag{1}$$

The inequality $\ge$ in (1) follows because any selection of $\ell - 1$ pairwise disjoint subintervals of (b, m) can be combined with any interval in (m, f) to form a selection of ℓ pairwise disjoint subintervals of (b, f). The inequality $\le$ follows by considering the last choice made in an optimal selection (x_i, y_i), $1 \le i \le \ell$, of pairwise disjoint subintervals of (b, f), namely where to start (x_ℓ, y_ℓ). If this point is denoted by $m^* \doteq x_\ell$, then

$$\sum_{i=1}^{\ell-1} \mathrm{val}(x_i, y_i) \le \mathrm{OPT}(b, m^*, \ell - 1)$$

and $\mathrm{val}(x_\ell, y_\ell) \le \mathrm{val}(\mathrm{BB}(m^*, f))$; by summing those two inequalities we obtain the inquality $\le$ in (1).

In order to compute $\mathrm{OPT}(1, n, k)$ we actually only need to consider the OPT-entries with $b = 1$, which form a 2-dimensional array. We start by computing the first row, i.e., $\mathrm{OPT}(1, f, 1)$ for each $f \in [n]$ using the blackbox as above. We then apply the recurrence (1) to compute row ℓ out of row $\ell - 1$ for $\ell = 2, 3, \ldots, k$, i.e., $\mathrm{OPT}(1, f, \ell)$ for all $f \in [n]$ out of $\mathrm{OPT}(1, f, \ell - 1)$ for all $f \in [n]$.

The time spent per entry is constant for the first row, and $O(n)$ for all other rows. As there are kn entries with $b = 1$, the overall running time is $O(n \cdot 1 + (k - 1)n \cdot n) = O(kn^2)$.

As we only need access to the previous row in order to compute the next one, the memory space needed is that for two rows, i.e., $O(n)$.

Part (b)

The standard solution consists of storing in the table entries for $\mathrm{OPT}(1, f, \ell)$ with $\ell > 1$ not only the value of the maximum on the right-hand side of (1) but also a choice m^* of m that realizes the maximum. If we keep track of the entire table $\mathrm{OPT}(1, f, \ell)$ for $1 \le f \le n$ and $1 \le \ell \le k$, we can backtrack through the table from position $(f, \ell) = (n, k)$ to the first row. Collecting the optimal choices of m yields a sequence $m^*_{k-1} \ge m^*_{k-2} \ge \cdots \ge m^*_1$. An optimal choice of subintervals is then given by $\mathrm{BB}(m^*_{i-1}, m^*_i)$ for $i \in [k]$ where $m^*_0 \doteq 1$ and $m^*_k \doteq n$.

The standard solution takes time $O(kn^2)$ to construct the table (as before) and $O(k)$ time to find the sequence of m_i's and the corresponding subintervals. The space requirement is $O(kn)$ as we need to store all rows during the construction of the table in order to enable the subsequent backtracking phase.

In order to reduce the space complexity to $O(n)$ while maintaining a running time of $O(kn^2)$, we use a simlar idea as for the time- and space-efficient algorithm for finding an optimal sequence alignment.

Suppose we want to compute a sequence of subintervals of (b, f) realizing $\mathrm{OPT}(b, f, \ell)$ for $\ell \ge 2$. Using the algorithm from part (a), we can compute the values $g(m) \doteq \mathrm{OPT}(b, m, \lceil \ell/2 \rceil)$ for all $m \in [b, f]$ in time $O(\ell(f - b)^2)$ and space $O(f - b)$. Applying the same algorithm to the sequence obtained by flipping the given sequence around the x-axis and y-axis, we obtain in the same time and space the values $h(m) \doteq \mathrm{OPT}(m, f, \lfloor \ell/2 \rfloor)$ for all $m \in [b, f]$. We then set $m^*_{\lceil \ell/2 \rceil}$ to an $m \in [b, f]$

such that $g(m) + h(m)$ is maximized. Recursively applying this process to the intervals $[b, m^*_{\lceil \ell/2 \rceil}]$ and $[m^*_{\lceil \ell/2 \rceil}, f]$ results in a sequence $m^*_1 \leq m^*_2 \leq \cdots \leq m^*_{k-1}$, from which we can obtain an optimal choice of subintervals using the blackbox as before.

We run this recursive process on input $(b, f) = (1, n)$ and $\ell = k$. The recursion tree has $\log(k)$ levels. The local amount of work involved in a node corresponding to an interval of size $s \doteq f - b$ and with an allotment of ℓ subintervals is upper bounded by $c \cdot \ell \cdot s^2$ for some constant c. Fix any level $d \in \{0, 1, \ldots, \log(k)\}$ of the recursion tree. The nodes at level d correspond to a partition of $(1, n)$, which implies that their sizes $s_1, s_2, dots$ add up to n. Their individual allotment of subintervals is at most $\lceil k/2^d \rceil$. Thus, the aggregated amount of local work at level d is upper bounded by

$$\sum_i c \cdot \lceil k/2^d \rceil \cdot s_i^2 \leq c \cdot \lceil k/2^d \rceil \cdot (\sum_i s_i)^2 \leq c \cdot (k/2^d + 1) \cdot n^2.$$

Summing over all d yields a total running time of $O(kn^2)$.

The space needed to run this process is $O(n)$ for finding the m^*_i's, $O(k)$ for storing them, and $O(\log k)$ for keeping track of the recursion stack. As we never need more than n intervals, we can assume $k \leq n$. Thus, the resulting space complexity is $O(n + k + \log k) = O(n)$.

Problem 4

We use an idea similar to the one in the search-to-decision reduction for Boolean Satisfiability. In particular, we set values for the variables one by one while maintaining the property that the partial assignment we have thus far can be extended to a full assignment satisfying the requirements. In the case of Boolean Satisfiability, we plug in the value true for the next variable, simplify the formula (by eliminating clauses that have a true literal and removing literals that are false) and see whether the resulting formula remains satisfiable. If so, we set the variable to true and continue with the remaining formula. If not, we instead set the variable to false, and continue with the formula obtained by plugging in the value false instead of true for that variable and simplifying it.

In the case of this problem there is an additional complication due to the fact that we cannot simply plug in the values of the variables that we are setting and simplify the formula as above. Consider the instance $(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_4 \vee x_5)$. There exists a solution in which x_1 is true. However, we cannot simply drop the first clause as that would leave the values of x_2 and x_3 free, whereas they both need to be set to false if x_1 is set to true. We also cannot eliminate the literal $\overline{x_1}$ from the second clause as we require the clauses to contain exactly three literals. Instead, we add the clause $\overline{x_1} \vee z \vee \overline{z}$, where z is a fresh variable. As exactly one of z and $\overline{z}$ is true under any assignment, the requirement that the clause $\overline{x_1} \vee z \vee \overline{z}$ contains exactly one true literal forces $\overline{x_1}$ false, or equivalently, x_1 to true.

Let ϕ be an instance of 1-in-3-SAT with n variables $x_1, x_2, \ldots, x_n$. Our search-to-decision reduction proceeds as follows.

1. Ask the black box B if ϕ is a yes-instance for 1-in-3-SAT. If not, report that there is no solution.

2. Add the clause $C_1 = (\overline{x_1} \vee z \vee \overline{z})$ to ϕ, where z is a variable that does not appear elsewhere in the formula.

3. Ask the black-box if this new instance is a yes-instance. If it is, then we output the value 1 for x_1 and keep the clause C_1 in ϕ. If not, then we know that every solution has to set x_1 to 0, so we output the value 0 for x_1, and remove the clause C_1 from ϕ.

4. Repeat this for each of the original variables, starting with the modified instance ϕ.

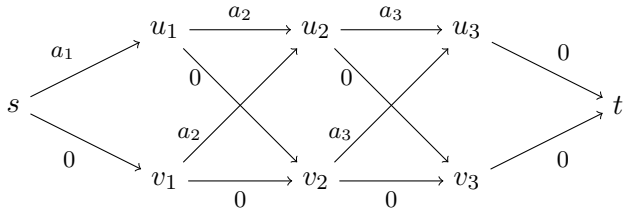
If ϕ is a false instance for 1-in-3-SAT, then the algorithm immediately reports that there is no solution. If ϕ is a true 1-in-3-SAT instance, then the procedure above constructs a solution for ϕ variable by variable. The procedure makes at most $O(n)$ calls to the black-box and does some polynomial amount of extra work (adding and removing the extra clauses), thus the overall reduction runs in polynomial time.

Problem 5

Call this problem the "exact-weight path" problem. We present a reduction to it from Subset Sum. We first reduce to a generalization of the "exact-weight path" problem where the input graph is *directed*. Then we show how to modify the reduction to the case of undirected graphs.

Let $a_1, \ldots, a_n \geq 0$ and T be an instance of Subset Sum. We construct an instance of the exact-weight path problem on directed graphs as follows. (An example for $n = 3$ is drawn in Figure 1.) Make a graph with $2n + 2$ vertices, named s, u_1, u_2, $\ldots$, u_n, v_1, v_2, $\ldots$, v_n, and t. Make edges $s \to u_1$; $s \to v_1$; $u_i \to u_{i+1}$, $u_i \to v_{i+1}$, $v_i \to u_{i+1}$, and $v_i \to v_{i+1}$ for $i = 1, \ldots, n-1$; $u_n \to t$; and $v_n \to t$. For each $i = 1, \ldots, n$, all edges entering u_i have weight a_i and all edges entering v_i have weight 0. The edges entering t both have weight 0. Let the resulting weighted, directed graph be G.

Figure 1: Example of directed graph G constructed for $n = 3$ case of Subset Sum.



We show that there is a subset $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = T$ if and only if there is a path in G from s to t of weight exactly T.

$\Longrightarrow$ Let $I \subseteq \{1, \ldots, n\}$ be such that $\sum_{i \in I} a_i = T$. Consider the path that starts at s and visits u_1 or v_1, then u_2 or v_2, etc., up to u_n or v_n, and finally stepping to t. The choice of u_i or v_i is determined by whether $i \in I$: u_i is visited when $i \in I$, and v_i is visited otherwise. The weight of this path is evidently $\sum_{i \in I} a_i = T$.

$\Longleftarrow$ Let P be a path from s to t in G of weight exactly T. P must start at s, step to u_1 or v_1, then to u_2 or v_2, and so on, up to u_n or v_n, before finally stepping to t. The weight of P can be written

$$\left(\sum_{i=1}^{n} \begin{cases} a_i & : \text{ if step to } u_i \\ 0 & : \text{ if step to } v_i \end{cases} \right).$$

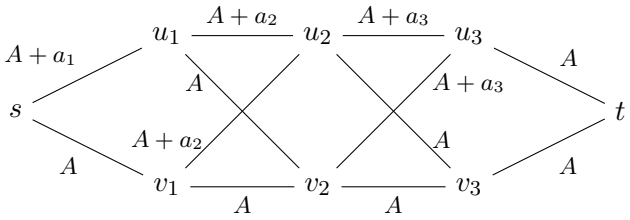
Let I be the set of indices i so that the path steps to u_i (and not v_i). Then this expression equals $\sum_{i \in I} a_i = T$, as desired.

This essentially completes the reduction to the generalized exact-weight path problem where the input graph is permitted to be directed. What remains is to reduce to the problem where the input graph is undirected. A first attempt here is to just make all the edges undirected. Then there is still a nice way, given a set I of numbers whose sum is T, to make a path from s to t of weight exactly T. However, there can be more paths than these. For example, the path from s to u_1 to v_2 to v_1 to s to u_1 to u_2 to u_3 to t in the example has weight $2a_1 + a_2 + a_3$, and this corresponds to no selection of items.

To fix this, we observe that the paths we do want to consider are exactly those with exactly $n+1$ edges in them, and all other paths use more edges. Thus it works to our advantage to add a large number A to the weight of every edge. The paths we want to consider will increase in weight by only $(n+1)A$, while all other paths will increase in weight by at least $(n+2)A$. Taking A sufficiently large, any path that doesn't go straight from s to t will be forced to have too high of weight.

In more detail, let $A = 1 + T$. Create an undirected graph G' as before, where the edges are not directed, and all weights are increased by A. For example, Figure 1 changes to look like Figure 2.

Figure 2: Example of undirected graph G' constructed for $n = 3$ case of Subset Sum. Every path from s to t of length at most $n + 1 = 4$ has to start at s, visit u_1 or v_1, then u_2 or v_2, then u_3 or v_3, then t.



We now show that there is a subset $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = T$ if and only if there is a path in G' from s to t of weight exactly $(n+1)A + T$.

$\Longrightarrow$ Let $I \subseteq \{1, \ldots, n\}$ be such that $\sum_{i \in I} a_i = T$. Consider the path that starts at s and visits u_1 or v_1, then u_2 or v_2, etc., up to u_n or v_n, and finally stepping to t. The choice of u_i or v_i is determined by whether $i \in I$: u_i is visited when $i \in I$, and v_i is visited otherwise. The weight of this path is

$$\left(\sum_{i \in I} (A + a_i)\right) + \left(\sum_{i \notin I} A\right) + A = (n+1)A + T.$$

$\Longleftarrow$ Let P be a path from s to t in G' of weight exactly $(n+1)A + T$. Each edge in G' has weight at least A, so if P has k edges in it, then its weight is at least kA. For $k > n+1$, this is larger than $(n+1)A + T$, so P must have at most $n+1$ edges. Every path from s to t with at most $n+1$ edges starts at s, steps to u_1 or v_1, then to u_2 or v_2, and so on, up to u_n or v_n,

before finally stepping to t. The weight of P can be written

$$\left(\sum_{i=1}^{n} A + \begin{cases} a_i & : \text{ if step to } u_i \\ 0 & : \text{ if step to } v_i \end{cases}\right) + A.$$

Let I be the set of indices i so that the path steps to u_i (and not v_i). Then this expression equals

$$\left(\sum_{i=1}^{n} A\right) + \left(\sum_{i \in I} a_i\right) + A = (n+1)A + \sum_{i \in I} a_i.$$

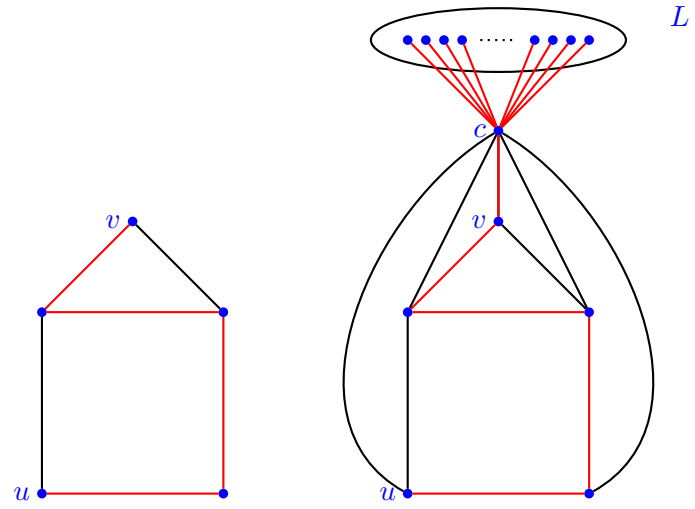
Since this equals $(n+1)A + T$, it follows that $\sum_{i \in I} a_i = T$.

The construction of G' can be done in time linear in the size of G'. G' has $O(n)$ vertices and $O(n)$ edges, and the weights have bitlength equal to the bitlengths of the a_i's, so the overall running time is polynomial (in fact linear) in the input length. Since Subset Sum is NP-hard, this proves that the exact-weight path problem is too.

Problem 6

Lets call this problem the 577-Tree problem. We present a reduction from the Hamiltonian path problem. Note that a Hamiltonian path in a graph G is precisely a spanning tree of G with 2 leaves.

Let $G = (V, E)$ be an instance of the Hamiltonian Path problem. We construct an instance of our problem as follows: Introduce a new vertex c and connect it with all vertices in G. Introduce a set L of 576 $(= 577 - 1)$ vertices ℓ_i and connect each ℓ_i to c. Let the resulting graph be H. An example is depicted in Figure 3.



(a) The graph G for the Hamiltonian path.

(b) The graph H for the 577-Tree problem

Figure 3: The red path in G is a Hamiltonian path, and correspondingly the red tree in H is a spanning tree with at most 577 leaves.

We now show that there exists a Hamiltonian path in G if and only if there exists a spanning tree with *at most* 577 vertices in H.

$\Longrightarrow$ If there exists a Hamiltonian path P with endpoints u and v in G, then we can find a spanning tree T as follows: Let T include P, the edge $\{v, c\}$ and the edges $\{c, \ell_i\}$ for all i. By construction, all vertices in H are visited by T and T is a tree with 1 (from the Hamiltonian path endpoint u) + 576 (from L) = 577 leaves.

$\Longleftarrow$ If there exists a spanning tree T in H with at most 577 leaves, we construct a Hamiltonian path in G as follows. Note that each vertex in L is by construction a leaf in H, and thus must be a leaf in T. Therefore, T only has 1 leaf in G. Let T' be the resulting tree of T after removing L from H. Thus T loses 576 leaves, but the vertex c becomes a new leaf in T'. Therefore, T' is a spanning tree of the graph $H - L$ (the graph obtained by removing the vertex set L from H) with only two leaves. Further removing c from T' results in a Hamiltonian path for G.

The construction runs in polynomial time since introducing 576 vertices takes constant time and connecting them to c takes constant time. Since Hamiltonian path is NP-hard, the 577-Tree problem problem is also NP-hard.