CS 170 Algorithms MT2 Soln
Fall 2014 David Wagner

Problem 1. [True or false] (9 points)

Circle TRUE or FALSE. Do not justify your answers on this problem.

(a) TRUE or [✓ FALSE]: Let $(S, V-S)$ be a minimum (s,t)-cut in the network flow graph G. Let (u,v) be an edge that crosses the cut in the forward direction, i.e., $u \in S$ and $v \in V-S$. Then increasing the capacity of the edge (u,v) necessarily increases the maximum flow of G.

Explanation: In the following graph, $S = \{s,u\}$ forms a minimum (s,t)-cut, but increasing the capacity of the edge (u,v) doesn't increase the maximum flow of G.

$$\begin{tikzcd} s \arrow[r, "2"] & u \arrow[r, "2"] & v \arrow[r, "2"] & t \end{tikzcd}$$

(b) TRUE or FALSE: All instances of linear programming have exactly one optimum.

Comment: We decided not to grade this question (all answers will be accepted as correct). The intended answer was "False": a linear program can have multiple solutions that all attain the optimum value (e.g., maximize $x+y$, subject to the requirement that $x+y \leq 5$). However, some students pointed out that it was not entirely clear whether "optimum" refers to the value of the solution or to the solution itself. Therefore, we decided not to grade this question, to avoid deducting points from someone who did understand the material but were thrown off by the ambiguity in the question.

Some linear programs have no feasible solution and thus arguably have no optimum (e.g., maximize x, subject to the requirement that $x \leq 5$ and $x \geq 6$), but one could also consider that case to be one where the maximum value is $-\infty$; in any case, we judged that this is too "tricky" to make a good basis for a question.

(c) [✓ TRUE] or FALSE: If all of the edge capacities in a graph are an integer multiple of 7, then the value of the maximum flow will be a multiple of 7.

Explanation: One proof is to notice that each iteration of the Ford-Fulkerson algorithm will increase the value of the flow by an integer multiple of 7. Therefore, by induction, Ford-Fulkerson will output a flow whose value is a multiple of 7. Since Ford-Fulkerson outputs a maximum flow, the value of the maximum flow must be a multiple of 7.

(d) [✓ TRUE] or FALSE: If we want to prove that a search problem X is NP-complete, it's enough to reduce 3SAT to X (in other words, it's enough to prove 3SAT $\leq_P X$).

(e) TRUE or [✓ FALSE]: If we want to prove that a search problem X is NP-complete, it's enough to reduce X to 3SAT (in other words, it's enough to prove $X \leq_P$ 3SAT).

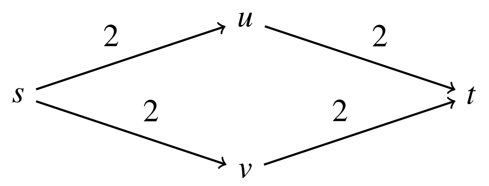
Explanation: For instance, 2SAT $\leq_P$ 3SAT (since you can solve 2SAT using an algorithm for 3SAT), but 2SAT is in P and hence not NP-complete (unless $P = NP$).

(f) TRUE or $\boxed{\text{FALSE}}$: For every graph G and every maximum flow on G, there always exists an edge such that increasing the capacity on that edge will increase the maximum flow that's possible in the graph.

Explanation: See the example graph shown in part (a).

(g) TRUE or $\boxed{\text{FALSE}}$: Suppose the maximum (s,t)-flow of some graph has value f. Now we increase the capacity of every edge by 1. Then the maximum (s,t)-flow in this modified graph will have value at most $f + 1$.

Explanation: In the following graph, the maximum flow has value $f = 4$. Increasing the capacity of every edge by 1 causes the maximum flow in the modified graph to have value 6.



(h) TRUE or $\boxed{\text{FALSE}}$: There is no known polynomial-time algorithm to solve maximum flow.

(i) $\boxed{\text{TRUE}}$ or FALSE: If problem A can be reduced to problem B, and B can be reduced to C, then A can also be reduced to C.

(j) $\boxed{\text{TRUE}}$ or FALSE: If X is any search problem, then X can be reduced to INDEPENDENT SET.

Explanation: INDEPENDENT SET is NP-complete.

(k) TRUE or $\boxed{\text{FALSE}}$: If we can find a single problem in NP that has a polynomial-time algorithm, then there is a polynomial-time algorithm for 3SAT.

Explanation: 2SAT is in NP and has a polynomial-time algorithm, but that doesn't necessarily mean that 3SAT has a polynomial-time algorithm.

(l) $\boxed{\text{TRUE}}$ or FALSE: If there is a polynomial-time algorithm for 3SAT, then every problem in NP has a polynomial-time algorithm.

Explanation: 3SAT is NP-complete.

(m) $\boxed{\text{TRUE}}$ or FALSE: We can reduce the search problem MAXIMUM FLOW to the search problem LINEAR PROGRAMMING (in other words, MAXIMUM FLOW $\leq_P$ LINEAR PROGRAMMING).

Explanation: We saw in class how we can express the maximum flow problem as a linear program.

(n) $\boxed{\text{TRUE}}$ or FALSE: We can reduce the search problem LINEAR PROGRAMMING to the search problem INTEGER LINEAR PROGRAMMING (in other words, LINEAR PROGRAMMING $\leq_P$ INTEGER LINEAR PROGRAMMING).

Explanation: You can reduce anything in P to anything in NP, and LINEAR PROGRAMMING is in P and INTEGER LINEAR PROGRAMMING is in NP. Or: You can reduce anything in NP to anything NP-complete, and LINEAR PROGRAMMING is in NP and INTEGER LINEAR PROGRAMMING is NP-complete.

(o) $\boxed{\text{TRUE}}$ or FALSE: Every problem in P can be reduced to 3SAT.

Explanation: This follows from the Cook-Levin theorem. Every problem in P is in NP. The Cook-Levin theorem says that everything in NP can be reduced to CircuitSAT, which can in turn be reduced to 3SAT. Or: Every problem in P is in NP, and every problem in NP can be reduced to any NP-complete problem. 3SAT is NP-complete.

(p) TRUE or $\boxed{\text{FALSE}}$: Suppose we have a data structure where the amortized running time of Insert and Delete is $O(\lg n)$. Then in any sequence of $2n$ calls to Insert and Delete, the worst-case running time for the nth call is $O(\lg n)$.

Explanation: The first $n-1$ calls might take $\Theta(1)$ time and the nth call might take $\Theta(n \lg n)$ time. This would be consistent with the premises, as it would mean that the total time for the first n calls is $\Theta(n \lg n)$.

(q) TRUE or $\boxed{\text{FALSE}}$: Suppose we do a sequence of m calls to Find and m calls to Union, in some order, using the union-find data structure with union by rank and path compression. Then the last call to Union takes $O(\lg^* m)$ time.

Explanation: The worst case time for a single call to Union might be much larger than its amortized running time.

Problem 2. [Short answer] (18 points)

Answer the following questions, giving a short justification (a sentence or two).

(a) If $P \neq NP$, could there be a polynomial-time algorithm for 3SAT?

Answer: No. 3SAT is NP-complete, so a polynomial-time algorithm for 3SAT would imply a polynomial-time algorithm for every problem in NP—it would imply that $P = NP$.

(b) If $P \neq NP$, could GRAPH 2-COLORING be NP-complete?

Answer: No. There's a polynomial-time algorithm for GRAPH 2-COLORING, so if it were NP-complete, we would have a polynomial-time algorithm for every problem in NP, which would mean that $P = NP$.

(c) If we have a dynamic programming algorithm with n^2 subproblems, is it possible that the running time could be asymptotically strictly more than $\Theta(n^2)$?

Answer: Yes, for instance, if the running time per subproblem is $\Theta(n)$ (or anything larger than $\Theta(1)$).

(d) If we have a dynamic programming algorithm with n^2 subproblems, is it possible that the space usage could be $O(n)$?

Answer: Yes, if we don't need to keep the solution to all smaller subproblems but only the last $O(n)$ of them or so.

For example, Fibonacci is an example where we only need to store the solution to the last few subproblems (it has n subproblems and only needs to store the last 2, so its space usage is $O(1)$), as does the solution to Problem 5.

Comment: We didn't accept "there might be n^2 subproblems but some of them might be solved repeatedly, so memoization would store only the distinct ones." In dynamic programming, when we count the number of subproblems, we count the number of distinct subproblems, without regard to how many times their solution is used when solving some other subproblem.

We didn't accept "some of the n^2 subproblems might never need to be solved." Implicitly, when we're counting the number of subproblems, we only count the ones we need to solve.

We didn't accept "if you use a recursive algorithm but don't memoize and just re-calculate the solution to some subproblems many times, you don't need much space." At that point the algorithm no longer counts as dynamic programming.

(e) Suppose that we implement an append-only log data structure, where the total running time to perform any sequence of n Append operations is at most $3n$. What is the amortized running time of Append?

Answer: $3n/n = O(1)$, as the amortized running time is the total time for many operations divided by the number of operations. (Incidentally, this is Q1 from HW7.)

(f) Suppose we want to find an optimal binary search tree, given the frequency of bunch of words. In other words, the task is:

Input: n words (in sorted order); frequencies of these words: $p_1, p_2, \ldots, p_n$.

Output: The binary search tree of lowest cost (defined as the expected number of comparisons in looking up a word).

Prof. Cerise suggests defining $C(i) =$ the cost of the optimal binary search tree for words $1 \ldots i$, writing a recursive formula for $C(i)$, and then using dynamic programming to find all the $C(i)$.

Prof. Rust suggests defining $R(i, j) =$ the cost of the optimal binary search tree for words $i \ldots j$, writing a recursive formula for $R(i, j)$, and then using dynamic programming to find all the $R(i, j)$.

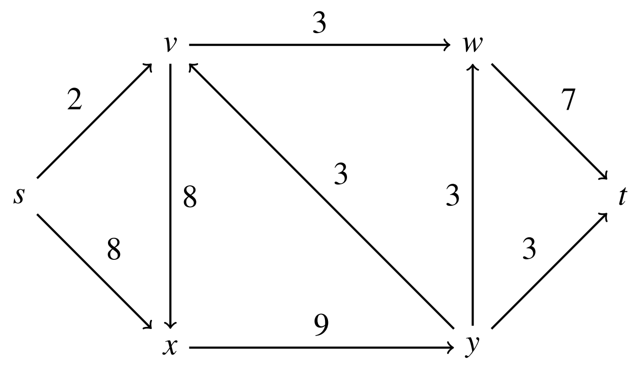
One of the professors has an approach that works, and one has an approach that doesn't work. Which professor's approach can be made to work? In what order should we compute the C values (if you choose Cerise's approach) or R values (if you choose Rust's approach)?

Answer: Rust (there's no easy way to compute $C(n)$ from $C(1), \ldots, C(n-1)$, but as we saw in the homework, Rust's approach does work). By increasing value of $j - i$.

(Incidentally, this is Q4 from HW8.)

Problem 3. [Max flow] (10 points)

Consider the following graph G. The numbers on the edges represent the capacities of the edges.

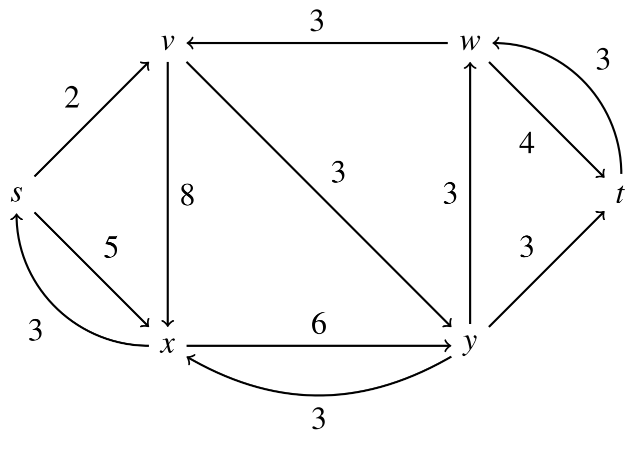


(a) How much flow can we send along the path $s \to x \to y \to v \to w \to t$?

Answer: 3

(b) Draw the resulting residual graph after sending as much flow as possible along the path $s \to x \to y \to v \to w \to t$, by filling in the skeleton below with the edges of the residual graph. Label each edge of the residual graph with its capacity.

Answer:



(c) Find a maximum (s,t)-flow for G. Label each edge below with the amount of flow sent along that edge, in your flow. (You can use the blank space on the next page for scratch space if you like.)

Answer: There are multiple valid answers. Here is one maximum flow:

$$\begin{tikzcd}
 & v \arrow[r, "3"] \arrow[dd, "0"] & w \arrow[dr, "6"] & \\
s \arrow[ur, "1"] \arrow[dr, "8"'] & & & t \\
 & x \arrow[r, "8"] & y \arrow[uu, "3"] \arrow[uul, "2"'] \arrow[ur, "3"'] &
\end{tikzcd}$$

Another maximum flow:

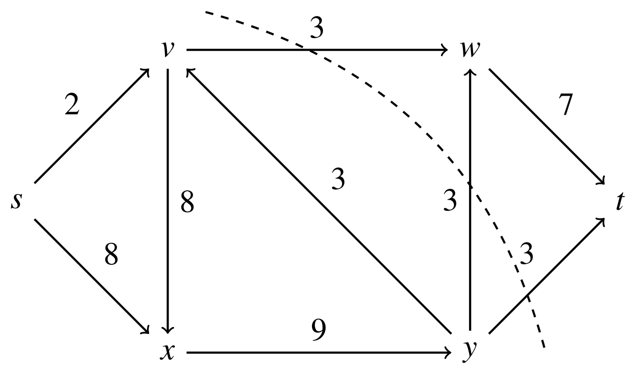
$$\begin{tikzcd}
 & v \arrow[r, "3"] \arrow[dd, "1"] & w \arrow[dr, "6"] & \\
s \arrow[ur, "2"] \arrow[dr, "7"'] & & & t \\
 & x \arrow[r, "8"] & y \arrow[uu, "3"] \arrow[uul, "2"'] \arrow[ur, "3"'] &
\end{tikzcd}$$

Yet another valid maximum flow:

$$\begin{tikzcd}
 & v \arrow[r, "3"] \arrow[dd, "0"] & w \arrow[dr, "6"] & \\
s \arrow[ur, "2"] \arrow[dr, "7"'] & & & t \\
 & x \arrow[r, "7"] & y \arrow[uu, "3"] \arrow[uul, "1"'] \arrow[ur, "3"'] &
\end{tikzcd}$$

(d) Draw a minimum (s,t)-cut for the graph G (shown below again).

Answer: $S = \{s,v,x,y\}$.



Problem 4. [Max flow] (12 points)

We would like an efficient algorithm for the following task:

Input: A directed graph $G = (V, E)$, where each edge has capacity 1; vertices $s, t \in V$; a number $k \in \mathbb{N}$

Goal: find k edges that, when deleted, reduce the maximum $s - t$ flow in the graph by as much as possible

Consider the following approach:

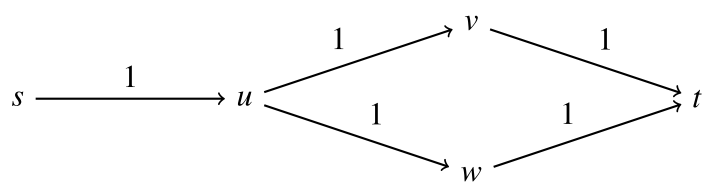
1. Compute a maximum (s,t)-flow f for G.
2. Let G^f be the residual graph for G with flow f.
3. Define a set S of vertices by (something).
4. Define a set T of edges by (something).
5. Return any k edges in T.

(a) How could we define the sets S and T in steps 3–4, to get an efficient algorithm for this problem?

Answer: $S = \{v \in V : v \text{ is reachable from } s \text{ in } G^f\}$.
$T = \{(v,w) \in E : v \in S, w \notin S\}$. (or: T = the edges that cross the cut $(S, V - S)$.)

Explanation: $(S, V - S)$ is a minimum (s,t)-cut, and by the min-cut max-flow theorem, its capacity is value(f). If we remove any k edges from T, then the capacity of the cut $(S, V - S)$ will drop to value$(f) - k$, and it'll be a minimum (s,t)-cut in the modified graph. Therefore, by the min-cut max-flow theorem, the value of the maximum in the modified graph will be value$(f) - k$. (Deleting k edges can't reduce the value of the maximum flow more than that, so this is optimal.)

Comment 1: T = edges that are saturated (i.e., where the flow on that edge is equal to the capacity) is not correct. There can be edges that are saturated but whose removal does not reduce the size of the maximum flow, e.g., because there is an alternate route flow could take. Consider, e.g., the following graph; we might have one unit of flow going via $s \to u \to v \to t$, but deleting the saturated edge $v \to t$ does not reduce the value of the maximum flow.



Comment 2: We expect definitions of the sets S and T that are mathematically correct. We graded the answers to Problem 4(a) strictly. It was not enough to have an intuitive understanding; you needed to be able to express your definition in terms that are mathematically accurate. We did not give any credit if your definition of S and/or T was imprecise, vague, or contains mistakes. (For instance, note that S should be a set of *vertices* and T should be a set of *edges*; if you defined T to be a set of vertices, you did not receive any credit for T.) Also, we do not give any partial credit for sloppy definitions like:

$S =$ vertices of a minimum $s-t$ cut

$T =$ edges of a minimum $s-t$ cut

An (s,t)-cut consists of two disjoint sets of vertices. These sets should be appropriately defined if used in the definitions of S and/or T.

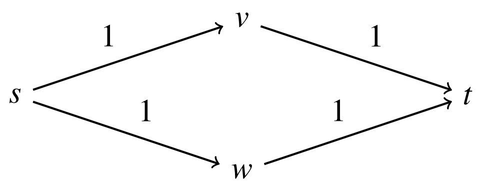
Also, the direction of the edges that are chosen to be added in T should be explicitly specified, and must be specified correctly to receive any credit. Note that edges $(w,v) \in E$ such that $w \notin S$ and $v \in S$ should not be included in T.

As a special case, we did give partial credit for definitions of S like the following, if they had no errors:

$S =$ one of the two disjoint sets of vertices of a minimum $s-t$ cut

This definition does not explain how to generate the set S and thus it does not give directly an algorithm, so it did not receive full credit. We deducted one point from such definitions.

Comment 3: Note that the definition: $S = \{v \in V : v \text{ is reachable from } t \text{ in } G^f\}$ is not correct. For example, consider the following graph:



Such a definition of S would imply $S = \{s,v,w,t\}$ which is not one of the two disjoint sets of vertices of a minimum (s,t)-cut.

(b) Is there an algorithm to implement step 1 in $O(|V||E|)$ time or less? If yes, what algorithm should we use? If no, why not? Either way, justify your answer in a sentence or two.

Answer: Yes. Ford-Fulkerson: the value of the max flow is at most $|V|-1$ (since there are at most $|V|-1$ edges out of s, each of which can carry at most 1 unit of flow), so Ford-Fulkerson does at most $|V|-1$ iterations, and each iteration takes $O(|E|)$ time.

Comment: Karp-Edmonds' running time is $O(|V||E|^2)$ in general, too slow. (There are apparently algorithms that can compute the max flow in an arbitrary graph in $O(|V||E|)$ time, but they are super-complex methods: e.g., Orlin's algorithm plus the King-Rao-Tarjan algorithm.)

Problem 5. [Dynamic programming] (12 points)

Let $A[1..n]$ be a list of integers, possibly negative. I play a game, where in each turn, I can choose between two possible moves: (a) delete the first integer from the list, leaving my score unchanged, or (b) add the sum of the first two integers to my score and then delete the first three integers from the list. (If I reach a point where only one or two integers remain, I'm forced to choose move (a).)

I want to maximize my score. Design a dynamic programming algorithm for this task. Formally:

Input: $A[1..n]$

Output: the maximum score attainable, by some sequence of legal moves

For instance, if the list is $A = [2,5,7,3,10,10,1]$, the best solution is $5+7+10+10 = 32$.

You do not need to explain or justify your answer on any of the parts of this question.

(a) Define $f(j)$ = the maximum score attainable by some sequence of legal moves, if we start with the list $A[j..n]$. Fill in the following base cases:

Answer:
$f(n) = 0$
$f(n-1) = 0$
$f(n-2) = \max(0, A[n-2] + A[n-1])$

(b) Write a recursive formula for $f(j)$. You can assume $1 \leq j \leq n-3$.

Answer: $f(j) = \max(f(j+1),\ A[j] + A[j+1] + f(j+3))$

(c) Suppose we use the formulas from parts (a) and (b) to solve this problem with dynamic programming. What will the asymptotic running time of the resulting algorithm be?

Answer: $\Theta(n)$.

Explanation: We have to solve n subproblems, and each one takes $\Theta(1)$ time to solve (since that is the time to evaluate the formula in part (b) for a single value of j).

(d) Suppose we want to minimize the amount of space (memory) used by the algorithm to store intermediate values. Asymptotically, how much space will be needed, as a function of n? (Don't count the amount of space to store the input.) Use $\Theta(\cdot)$ notation.

Answer: $\Theta(1)$.

Explanation: As we solve the subproblems in the order $f(n), f(n-1), \ldots, f(1)$, we don't need to store all of the solutions; we only need to store the three last values of $f(\cdot)$.

Problem 6. [Greedy cards] (12 points)

Ning and Evan are playing a game, where there are n cards in a line. The cards are all face-up (so they can both see all cards in the line) and numbered 2–9. Ning and Evan take turns. Whoever's turn it is can take one card from either the right end or the left end of the line. The goal for each player is to maximize the sum of the cards they've collected.

(a) Ning decides to use a greedy strategy: "on my turn, I will take the larger of the two cards available to me". Show a small counterexample ($n \leq 5$) where Ning will lose if he plays this greedy strategy, assuming Ning goes first and Evan plays optimally, but he could have won if he had played optimally.

Answer: $[2,2,9,3]$.

Explanation: Ning first greedily takes the 3 from the right end, and then Evan snatches the 9, so Evan gets 11 and Ning gets a miserly 5. If Ning had started by craftily taking the 2 from the left end, he'd guarantee that he would get 11 and poor Evan would be stuck with 5.

There are many other counterexamples. They're all of length at least 4.

(b) Evan decides to use dynamic programming to find an algorithm to maximize his score, assuming he is playing against Ning and Ning is using the greedy strategy from part (a). Let $A[1..n]$ denote the n cards in the line. Evan defines $v(i,j)$ to be the highest score he can achieve if it's his turn and the line contains cards $A[i..j]$.

Evan needs a recursive formula for $v(i,j)$. Fill in a formula he could use, below.

Evan suggests you simplify your expression by expressing $v(i,j)$ as a function of $\ell(i,j)$ and $r(i,j)$, where $\ell(i,j)$ is defined as the highest score Evan can achieve if it's his turn and the line contains cards $A[i..j]$, if he takes $A[i]$; also, $r(i,j)$ is defined to be the highest score Evan can achieve if it's his turn and the line contains cards $A[i..j]$, if he takes $A[j]$. Write an expression for all three that Evan could use in his dynamic programming algorithm. You can assume $1 \leq i < j \leq n$ and $j - i \geq 2$. Don't worry about base cases.

Answer:
$v(i,j) = \max(\ell(i,j), r(i,j))$

$$\text{where } \ell(i,j) = \begin{cases} A[i] + v(i+1, j-1) & \text{if } A[j] > A[i+1] \\ A[i] + v(i+2, j) & \text{otherwise.} \end{cases}$$

$$r(i,j) = \begin{cases} A[j] + v(i+1, j-1) & \text{if } A[i] \geq A[j-1] \\ A[j] + v(i, j-2) & \text{otherwise.} \end{cases}$$

Comment: We didn't specify how Ning resolves ties, if the card on the left end has the same value as the card on the right end. Therefore, your recursive formula can resolve ties in any manner you want: any correct formula will be accepted as correct regardless of how you decided to resolve ties. (For instance, the formula above assumes that if there is a tie, Ning takes the card on the left end.)

(c) What will the running time of the dynamic programming algorithm be, if we use your formula from part (b)? You don't need to justify your answer.

Answer: $\Theta(n^2)$.

Answer: There are $n(n+1)/2$ subproblems and each one can be solved in $\Theta(1)$ time (that's the time to evaluate the recursive formula in part (b) for a single value of i, j).

Problem 7. [Tile David's walkway] (14 points)

David is going to lay tile for a long walkway leading up to his house, and he wants an algorithm to figure out which patterns of tile are achievable. The walkway is a long strip, n meters long and 1 meter wide. Each 1×1 meter square can be colored either white or black. The input to the algorithm is a pattern $P[1..n]$ that specifies the sequence of colors that should appear on the walkway. There are three kinds of tiles available from the local tile store: a 1×1 white tile (W), a 2×1 all-black tile (BB), and a 3×1 black-white-black tile (BWB). Unfortunately, there is a limited supply of each: the tile store only has p W tiles, q BB's, and r BWB's in stock.

Devise an efficient algorithm to determine whether a given pattern can be tiled, using the tiles in stock. In other words, we want an efficient algorithm for the following task:

Input: A pattern $P[1..n]$, integers $p, q, r \in \mathbb{N}$

Question: Is there a way to tile the walkway with pattern P, using at most p W's, q BB's, and r BWB's?

For example, if the pattern P is WBWBWBBWWBWBW and $p = 5$, $q = 2$, and $r = 2$, the answer is yes: it can be tiled as follows: $\boxed{\text{W}}\boxed{\text{BWB}}\boxed{\text{W}}\boxed{\text{BB}}\boxed{\text{W}}\boxed{\text{W}}\boxed{\text{BWB}}\boxed{\text{W}}$.

For this problem, we suggest you use dynamic programming. Define

$$f(j,p,q,r) = \begin{cases} \text{True} & \text{if there's a way to tile } P[j..n] \text{ using at most } p \text{ W's, } q \text{ BB's, and } r \text{ BWB's} \\ \text{False} & \text{otherwise.} \end{cases}$$

You don't need to justify or explain your answer to any of the following parts. You can assume someone else has taken care of the base cases ($j = n-2, n-1, n$), e.g., $f(n,p,q,r) = (P[n] = W \wedge p \geq 1)$ and so on; you don't need to worry about them.

(a) Write a recursive formula for f. You can assume $1 \leq j \leq n-3$.

Answer:

$$\begin{aligned} f(j,p,q,r) = &(P[j] = W \wedge p > 0 \wedge f(j+1, p-1, q, r)) \vee \\ &(P[j..j+1] = BB \wedge q > 0 \wedge f(j+2, p, q-1, r)) \vee \\ &(P[j..j+2] = BWB \wedge r > 0 \wedge f(j+3, p, q, r-1)). \end{aligned}$$

(b) If we use your formula from part (a) to create a dynamic programming algorithm for this problem, what will its asymptotic running time be?

Answer: $\Theta(npqr)$ or $\Theta(n(p+1)(q+1)(r+1))$.

(There are $n(p+1)(q+1)(r+1)$ subproblems. Each subproblem can be solved in $\Theta(1)$ time using the formula in part (a).)

$\Theta(n^4)$ is also OK (since without loss of generality we can assume $p < n$, $q < n$, and $r < n$; there's no way to use more than n tiles).

$\Theta(pqr)$ is also a reasonable answer: with the recursive formula in part (a), j is determined by the p, q, r arguments to f.

There's also a way to make this run in $\Theta(n)$ time, but not solving all of the subproblems: you look at whether $P[j..n]$ starts with W, BB, or BWB, then evaluate f at the corresponding value. This leads to a linear scan through the P array.

We decided to accept all of $\Theta(n)$, $\Theta(n^2)$, $\Theta(n^3)$, $\Theta(n^4)$, since one can make an argument for each of them, as well as $\Theta(npqr)$ and $\Theta(pqr)$.

Problem 8. [Walkways, with more tiles] (13 points)

Now let's consider SUPERTILE, a generalization of Problem 7. The SUPERTILE problem is

Input: A pattern $P[1..n]$, tiles $t_1, t_2, \dots, t_m$

Output: A way to tile the walkway using a subset of the provided tiles in any order, or "NO" if there's no way to do it

Each tile t_i is provided as a sequence of colors (W or B). Each particular tile t_i can be used at most once, but the same tile-sequence can appear multiple times in the input (e.g., we can have $t_i = t_j$). The tiles can be used in any order.

For example, if the pattern P is WBBBWWB and the tiles are $t_1 =$ BBW, $t_2 =$ B, $t_3 =$ WB, $t_4 =$ WB, then the answer is yes, $t_3 t_1 t_4$ (this corresponds to $\boxed{\text{WB}\,|\,\text{BBW}\,|\,\text{WB}}$).

(a) Is SUPERTILE in NP? Justify your answer in a sentence or two.

Answer: Yes. There is a polynomial-time solution-checking algorithm: check that the tiles used are a subset of $t_1, \dots, t_m$ and no tile is used more times than it appears in $t_1, \dots, t_m$.

(b) Prof. Mauve believes she has found a way to reduce SUPERTILE to SAT. In other words, she believes she has proven that SUPERTILE $\le_P$ SAT. If she is correct, does this imply that SUPERTILE is NP-hard? Justify your answer in a sentence or two.

Answer: No. SUPERTILE could be much easier than SAT. SAT is NP-complete, so everything in NP reduces to SAT.

(c) Prof. Argyle believes he has found a way to reduce SAT to SUPERTILE. In other words, he believes he has proven that SAT $\le_P$ SUPERTILE. If he is correct, does this imply that SUPERTILE is NP-hard? Justify your answer in a sentence or two.

Answer: Yes. This reduction shows that SUPERTILE is at least as hard as SAT, which is NP-complete.

(d) Consider the following variant problem, ORDEREDTILE:

Input: A pattern $P[1..n]$, tiles $t_1, t_2, \dots, t_m$

Output: A way to tile the walkway using a subset of the tiles, in the provided order, or "NO" if there's no way to do it

In this variant, you cannot re-order the tiles. For instance, if P is WBBBWWB and the tiles are $t_1 =$ BBW, $t_2 =$ B, $t_3 =$ WB, $t_4 =$ WB, then the output is "NO"; however, if the tiles are $t_1 =$ WB, $t_2 =$ B, $t_3 =$ BBW, $t_4 =$ WB, then the answer is yes, $t_1 t_3 t_4$.

Rohit's advisor asked him to prove that ORDEREDTILE is NP-complete. Rohit has spent the past few days trying to prove it, but without any success. He's wondering whether he just needs to try harder or if it's hopeless. Based on what you've learned from this class, should you encourage him to keep trying, or should you advise him to give up? Explain why, in 2–3 sentences.

Answer: Give up. There's a polynomial-time algorithm for ORDEREDTILE using dynamic programming, so any proof that ORDEREDTILE is NP-complete would imply that $P = NP$. Given how many other researchers have tried, it's not reasonable to ask Rohit to prove $P = NP$.

Explanation: The dynamic programming algorithm: let $f(i, j)$ be true if there is a way to tile $P[i..n]$ using tiles $t_j, \dots, t_m$. Then

$$f(i,j) = f(i,j+1) \vee (P[i..i+\text{len}(t_j)-1] = t_j \wedge f(i+\text{len}(t_j), j+1)),$$

so we obtain a $\Theta(nm)$ time algorithm for ORDEREDTILE.