COMP 360 Algorithm Design Techniques

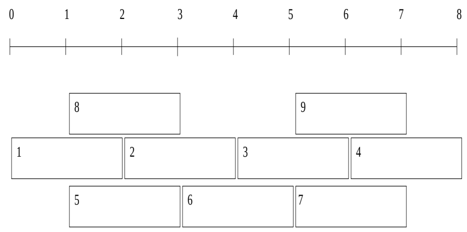
Assignment 2: Solutions

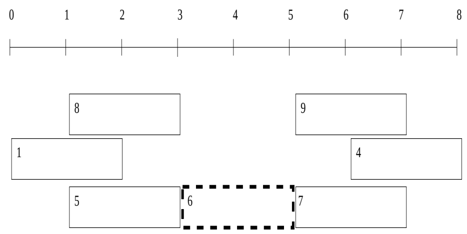
Please send any suspected errors or typos to sue@cs.mcgill.ca .

(1) *Counter-examples for Greediness.*

(a) Consider a ruler with $n = 2^2 = 4$ links, where $l_1 = 2$, $l_2 = 1$, $l_3 = 3$, and $l_4 = 3$. The optimum folding length is 3, which happens to be the length of the longest link. Clearly we can't do better than that! However, the proposed greedy strategy considers the first two links, and folds link 2 against link 1, achieving a folded length of 2 for the subruler consisting of just the first two links. Similarly, for the last two links, the greedy strategy folds link 4 against link 3, achieving a folded length of 3 for the subruler consisting of the last two links. Now, no matter how we put the two subrulers together at the joint between link 2 and link 3 (either folding this joint or making it straight), the total folded length of the entire ruler is 4, not 3. Amazing.

(b) Here is an example where the optimal solution is 4 activities, $\{1, 2, 3, 4\}$. However, the greedy algorithm first chooses item 6; using the remaining feasible activities the best it can achieve is a schedule of three activities.





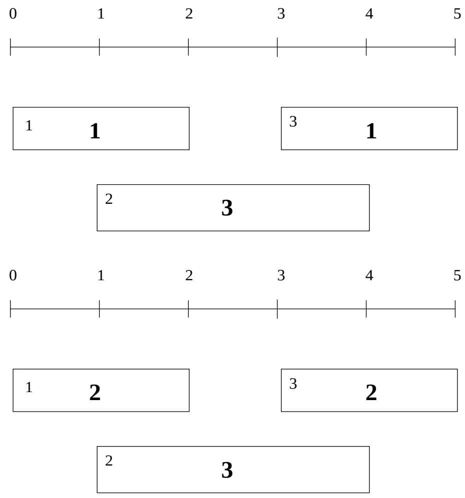
(2) *Weighted Activity Selection.*

(a) Items 1 and 3 are chosen but the optimal solution is item 2.

(b) Item 2 chosen but the optimal solution is $\{1, 3\}$.

(3) *Two-Colourability*

(a) Suppose we have a 2-colouring of G. Take any odd cycle C in G. Let C contain $2k + 1$ vertices. Clearly either at least $k + 1$ of the vertices are coloured blue



or at least $k+1$ of the vertices are coloured red. Without loss of generality we may assume that at least $k+1$ of the vertices are coloured blue. So we have $k+1$ blue vertices on a cycle of length $2k+1$. This implies that two of the blue vertices must be adjacent to each other on C. This contradicts the assumption that we had a 2-colouring of G.

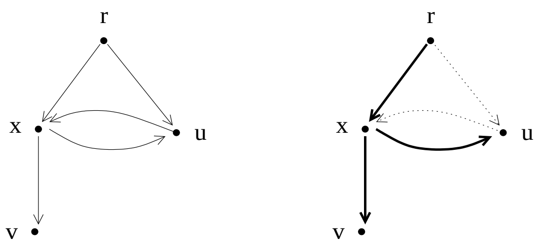
(b) Grow a breadth first search tree T in G rooted at some vertex r. Colour the vertices at odd distance from r red along the tree T; colour those at an even distance blue (note that r itself is blue). This gives a valid 2-colouring. To see this, recall that for any edge u, v in G the vertices are either at the same distance from r in T or their distances from r differ by exactly one. In the former case, let x be the least common ancestor of u and v in T. Let P_1 be the path from x to u in T and P_2 be the path from x to v in T. Observe that these two paths contain the same number of edges. It follows that $P_1 \cup (u, v) \cup P_2$ is a cycle of odd length. This is a contradiction and so, for any edge u, v in G, the distances of the vertices u and v from r in T differ by exactly one. In this case u and v are given different colours as required.

(4) *Depth First Search Trees*

There is a path from u to v via x. A valid DFS tree is shown in bold (we assume that from x we visit u before v and thus $start(u) < start(v)$). Observe that when we reach u all its neighbours are marked and we are forced to reach v by returning to x first. Thus, u is not ancestor of v in T.

(5) *The Knapsack Problem*

This problem can be solved by dynamic programming as follows: Create an $n \times W$ table whose j^{th} column ($j = 0, \ldots, W$) represents a knapsack of capacity j and whose i^{th} row ($i = 0, 1, 2 \ldots, n$) corresponds to the situation in which only the first i items are allowed to be considered when selecting items to put in the knapsack (if $i = 0$ then no items may be selected).. The entry $t(i, j)$ in row i, column j will give the



best value that can be obtained by selecting from items $1, ..., i$, to fill a knapsack of capacity j. Hence the answer to the original problem will appear in row n, column W (there, we are filling a knapsack of full capacity m, selecting from unlimited quantities of all n items).

How can this table be filled? We must find a formula that expresses $t(i, j)$ in terms of t's for smaller i's and/or j's. What is $t(i, j)$? There are two possibilities:

Case 1. The best value that can be obtained for a knapsack of capacity j selecting from items $1, \ldots, i$ is actually obtained by not taking advantage of the availability of item i – in other words, by just optimally selecting various quantities (possibly choosing the quantity to be 0) for items $1, \ldots, i-1$. In this case, we have $t(i, j) = t(i-1, j)$.

Case 2. The best value is obtained by using at least one (maybe more) of item i. In this case, we can imagine removing exactly one of item i from the knapsack. Then we would be left with a knapsack of capacity $j - w_i$, and the remaining contents would still have been chosen optimally from items $1, \ldots, i$. Hence in this case,

$$t(i, j) = t(i, j - w_i) + v_i$$

The first term in the above expression is the value of the knapsack with one unit of item i removed, and the second term adds back the value of this item.

Of course, we don't know in advance when we will be in Case 1 and when we will be in Case 2, but this doesn't matter: we simply compute *both* values, and then select the better one, breaking ties arbitrarily i.e.

$$t(i, j) = \max[t(i-1, j), t(i, j - w_i) + v_i]$$

We have one caveat: if $j < w_i$, there is not enough room in the knapsack to obtain *any* units of item i. In this case we have no choice, we set $t(i, j) = t(i-1, j)$.

Base Cases. What remains to be done is to specify enough table entries so that the rest of the table can be filled out by using the formulas above. It is good enough to specify the first row and column. For entry $(0, j)$, since no items are available for selection, the knapsack must be empty and thus has value 0. For entry $(i, 0)$, since the knapsack has zero capacity, again no items may be selected and the value is 0..

Dynamic Programming Solution. Puting this together: we have that $t(i, j)$ is the optimal solution for a knapsack with capacity of weight j then considering only the first i items. We may recursively define $t(i, j)$ by

$$\begin{aligned} t(i,j) &= 0 && \text{if } i = 0 \text{ or } j = 0 \\ &= t(i-1, j) && \text{if } w_i > j \\ &= \max[\, t(i-1, j), t(i, j - w_i) + v_i \,] && \text{otherwise} \end{aligned}$$

Recovering an Optimal Solution. As table entries are determined using the formula, a pointer to the table entry that gave rise to the *max* can be placed in entry (i, j) along with $t(i, j)$. You should try working out the details of this to make sure you understand. Invent your own example.

Running time. The number of entries in the table is nW. As long as the values stay small (do not grow with n), the time to compute each table entry is bounded by a constant that works for the whole table. Hence the running time is in $\Theta(nW)$. Of course, W can be a large function of n (e.g. $W = n!$) so the algorithm is not necessarily fast in terms of n.