ADS 2020 Coursework 2

Nathan Sharp

November 20, 2020

1. Consider the Problem of taking a set of n items with sizes $s_1, ..., s_n$ and values $v_1, ..., v_n$ respectively. We assume $s_i, v_i \in \mathbb{N}$ for all $1 \leq i \leq n$. Suppose we are also given a capacity $C \in \mathbb{N}$.

   The packing problem is the problem of finding a subset $S \subseteq \{1, ..., n\}$ such that $\sum_{i \in S} s_i \leq C$ and such that $\sum_{i \in S} v_i$ is maximised subject to the first constraint.

   We write $P_{n,C}$ to denote the value $\sum_{i \in S} v_i$ of the maximum-value packing on the set of all items. For any $k \leq n$, and any $\hat{C} \leq C, \hat{C} \in \mathbb{N}$, we can consider the same problem on the first k items in regard to capacity $\hat{C}$. We denote the maximum-value packing for such a subproblem by $P_{k,\hat{C}}$.

   The goal is to develop a $\Theta(n \cdot C)$ dynamic programming algorithm to compute the optimal packing wrt. the original n items and capacity C.

   (a) Prove a suitable recurrence for $P_k, \hat{C}$ that holds for all $k \leq n$ and $\hat{C} \leq C$.

   (b) Use your recurrence above to develop a $\Theta(n \cdot C)$ dynamic programming algorithm to compute the optimal packing wrt. the original n items and capacity C. Formally justify the $\Theta(n \cdot C)$ run time of your algorithm.

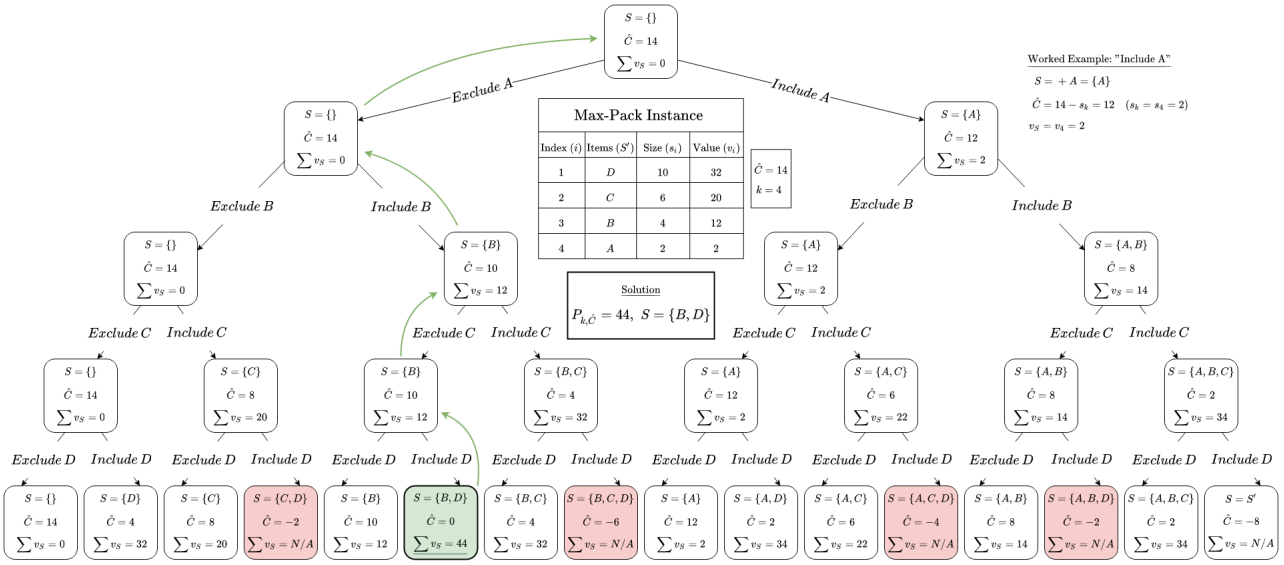
Solution:

(a) A simple brute force strategy for the max-packing problem is to consider all the subsets of items that satisfy the capacity constraint and take the maximum. [1]

We can model this as a binary tree, with root $\emptyset$, where each node at height k has children either– (1) excluding, or (2) excluding and including the k^{th} element in the continuation of the local subtree dependant on whether adding the size s_k exceeds the remaining capacity $\hat{C}$– (1), or not– (2).

The optimal packing solution, $P_{k,\hat{C}}$, will correspond to maximum leaf nodes packing value and the inclusive set of nodes connecting it to the root.

Figure 1: A Brute Force Max-Pack Binary Tree Instance



Note that $\sum v_S$ is used as shorthand for $\sum_{i \in S} v_i$.

Its self-evident that our tree models the brute force solution [2] and is recursive in nature, each subtree is a legitimately defined problem instance in itself, [3] and the subproblems are combined to produce the answer.

Hence our solution to the max-pack problem can directly be written as a recursion [4]

$$T(k, \hat{C}) = \begin{cases} T(k-1, \hat{C}) & \text{if } s_k > \hat{C} \qquad (1) \\ max(T(k-1, \hat{C}),\, T(k-1, \hat{C} - s_k) + v_k) & \text{if } s_k \leq \hat{C} \qquad (2) \end{cases}$$

In case (1) the size, s_k, of the k^{th} element is greater than the (remaining) capacity $\hat{C}$. Hence it cannot be added to the packing on this subtree and the problem is equal to the (sub)problem with the k^{th} element removed. This is equivalent to a node in the binary-tree with a sole child subtree, excluding the item in question.

In case (2), the size, s_k, of the k^{th} element is less than the (remaining) capacity $\hat{C}$. Hence it might be a member of the maximal packing solution and we take the maximum of the subproblem with the k^{th} element excluded (as in (1)) and the subproblem with it included, subtracting its size s_k from the (remaining) capacity $\hat{C}$ and adding its value, v_k to the output total. This is equivalent

[1] informally bounded by $O(2^n)$, the number of subsets.

[2] the leaf nodes are the 2^n combinations

[3] over the the set of items it's children evaluates, and remaining capacity $\hat{C}$.

[4] Note that this recursion ignores the base (leaf) case when $k = 0$ and an algorithm would return a definite value

to a node on the binary-tree with two child subtrees, one excluding and one including the item in question.

In conclusion the recursion simply a representation the binary tree of the brute force solution to the max packing problem.

(b) The following dynamic programming algorithm calculates the solution to the optimal packing problem in $\Theta(n \cdot C)$. The only additions to a naive implementation of recursion in part (a) is

i. handling of the base (leaf) case where $k = 0$

ii. the creation of an $(n \times C)$ lookup table to store the results of previously solved sub problems

Informally, the naive recursion in the worst case is bounded equivalently to the brute force solution, $\Theta(2^n)$, whereas a lookup table allows us to forgo calculating entire subtrees, every time there has been a problem with a matching element n and capacity C solved before.

The SOLVE-MAX-PACK pseudocode algorithm takes

- S': $\{\mathbb{N}\}$ — a set of *item indices*
- $\{s_1, ...s_n\}$: $\{\mathbb{N}\}$ — a set of *item sizes*
- $\{v_1, ...v_n\}$: $\{\mathbb{N}\}$ — a set of *values*
- C: $\mathbb{N}$ — a *capacity.*

and returns

- $P_{n,C}$: $\mathbb{N}$ — a *maximum packing value*
- S: $\{\mathbb{N}\}$ — the set of *item indices* in this maximum pack

Algorithm 1 SOLVE-MAX-PACK(S', $s_1, ..., s_n$, $v_1, ...v_n$, C): $\rightarrow P_{n,C}$, S

1: $n \leftarrow$ LENGTH(S)

2: $Arr \leftarrow$ a *null* $(n \times C)$ 2d integer array

3: **return** MAX-PACK-REC(S', $s_1, ..., s_n$, $v_1, ...v_n$, C, Arr)

Algorithm 2 MAX-PACK-REC(S', $s_1, ..., s_n$, $v_1, ...v_n$, C, Arr): $\rightarrow P_{n,C}$, S

1: $n \leftarrow$ LENGTH(S)
2: **if** $C \leq 0$ **or** $S' = \emptyset$ **then**
3: **return** $0, \emptyset$
4: **end if**
5: **if** $Arr[n][C] \neq null$ **then**
6: $P_{n,C} \leftarrow Arr[n][C]$
7: **return** $P_{n,C}$, S'
8: **end if**
9: **if** $s_n > C$ **then**
10: $Arr[n][C]$, $S \leftarrow$ MAX-PACK-REC($S - S_n$, $s_1, ..., s_{n-1}$, $v_1..., v_{n-1}$, C, Arr)
11: **return** $Arr[n][C]$, S
12: **else**
13: $MaxPack_{excl} \leftarrow$ MAX-PACK-REC($S - S_n$, $s_1..s_{n-1}$, $v_1..v_{n-1}$, C, Arr)
14: $MaxPack_{incl} \leftarrow$ MAX-PACK-REC($S - S_n$, $s_1..s_{n-1}$, $v_1..v_{n-1}$, $C - s_n$, Arr)
15: $MaxPack_{incl}[0] \leftarrow MaxPack_{incl}[0] + v_n$
16: $MaxPack_{incl}[1] \leftarrow MaxPack_{incl}[1] + S_n$
17: **if** $MaxPack_{incl}[0] \geq MaxPack_{excl}[0]$ **then**
18: $Arr[n][C] \leftarrow MaxPack_{incl}[0]$
19: **return** $MaxPack_{incl}$
20: **else**
21: $Arr[n][C] \leftarrow MaxPack_{excl}[0]$
22: **return** $MaxPack_{excl}$
23: **end if**
24: **end if**

Note that Arr is assumed to be global.

Analysis of the algorithms run time is as follows–

- SOLVE-MAX-PACK line 1 finding the length of S is $\Theta(1)$ [5]
- line 2 takes $\Theta(n \cdot C)$ time to initialise Arr
- line 3 calls MAX-PACK-REC
- MAX-PACK-REC line 1 is, as above, $\Theta(1)$
- line 2's comparisons are $\Theta(1)$
- lines 3-4 only occur in the base (leaf) case so is trivial

[5] potentially $\Theta(n)$ in some implementations, but this makes to difference to the overall runtime

- line 5-8's lookup in Arr is trivial in the sense of running in $\Theta(1)$ but critical in saving us from recomputing previously solved problems in later lines.
- line 9's comparison is $\Theta(1)$
- lines 10 & 11 make a single recursive call. However a worse case is the 2 recursive calls in lines 13-14. As this worst case of $s_n \leq C \, \forall s_n$ can occur every time [6], lines 10 & 11 can be safely ignored in our analysis.
- lines 13 & 14 makes two recursive calls at each level. We know there are exactly a maximum of $n \cdot C$ unique recursive calls that will not trigger the $\Theta(1)$ lookup table in line 5. Hence this bounds the total number of recursive calls that can be made by any instance of the algorithm.
- lines 15 & 16 are trivial
- lines 16-22's if-else block is all $\Theta(1)$

It can be seen that all relevant lines are individually $\Theta(1)$ except for line 2 of Solve-Max-Pack and lines 13 & 14 of Max-Pack-Rec, which are both $\Theta(n \cdot C)$. Adding these we can remove the constant term and hence the total running time for our dynamic-programming algorithm for the max packing problem is

$$\underline{T(n, C) \in \Theta(n \cdot C)}$$

2. King Arthur has problems to administer his realm. His court contains n knights and he rules over m counties. The knights differ in their abilities and local popularity: Each knight i can oversee at most q_i counties, and each county j will revolt unless it is overseen by some knight in a given subset $S_j \subseteq \{1, ..., n\}$ of the knights. Only 1 knight can oversee a county, to prevent conflicts between the knights.

   The king discusses the problem with his court magician Merlin.

   (a) Show how Merlin can use the Max-Flow algorithm to efficiently compute an assignment of the counties to the knights that prevents a revolt, provided that one exists.

   How can he determine whether the algorithm was successful?

   Prove the complexity of this algorithm, *in terms of some function* $F(v, e)$, where $F(v, e)$ denotes the running time of a Max-Flow algorithm on a graph with v vertices and e edges.

   (b) Suppose that Merlin runs the Max-Flow algorithm on the encoded instance, but it does not produce a solution that fits the constraints and prevents all counties from revolting.

[6] take the case where the sum of all the sizes is less than C

King Arthur demands an explanation. While he believes that the algorithm/encoding (and proof) is correct, he doubts that Merlin has executed the algorithm correctly on the given large instance. Merlin needs to convince the king that no suitable assignment is possible under the given constraints. Re-running the algorithm step-by-step in front of the king is not an option, because the king does not have that much time.

How can Merlin *quickly* convince the king that there is no solution, based on the structure of the Max-Flow problem? (Note that the argument must work for every instance where there is no solution, not just a particular instance.)

Solution:

(a) The following shows the problem represented as a Network-Flow diagram. Note that all unlabeled capacities (edges between nodes) have a value of 1.

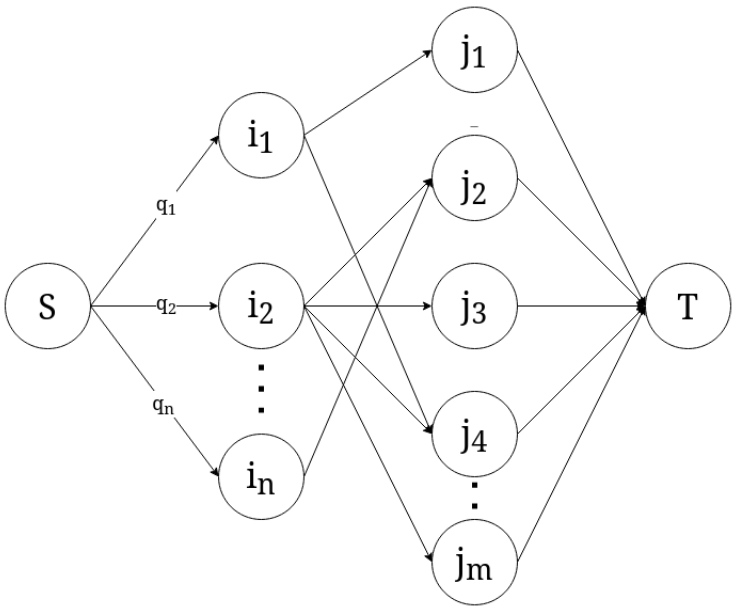


Figure 2: King Arthur's Flow-Network

If not obvious by inspection, the *flow network* [7] in Figure 2 maps to the constraints of King Arthur's problem in the following way–

- S is the source edge
- the vertical '$\cdots$"s, represent arbitrarily many additional knights, i and counties, j.

[7] as defined in lecture 10

- the 'i' column of vertices represents the $i_1, ..., i_n$ knights each of which have an input capacity $q_1, ..., q_n$, representing integer value of the maximum number of counties they can oversee.
- The 'j' column represents the $j_1, ..., j_m$ counties, each of which has an input edge connecting it from the S_m subsets of knights that can oversee it and a single output capacity of 1, connecting it to the target, T, as it can only be ruled by one knight.
- T the target edge

Its clear that this construction generalises to problem instances of arbitrary (n, m) size and that it satisfies the input constraints of a max-flow problem– (i) a connected directed graph with capacities for each edge and two distinguished edges and, (ii) a source, S, and target, T. Note, the only way this is not the case is if (i) is violated in the input– the input instance has counties with an empty subset of possible ruling knights, in which case the problem is intractable, or, if there is a knight no county approves of, in which case this knight should be excluded from the flow network construction.

The solution to King Arthur's problem will be obtained by running a suitable max-flow algorithm and then assigning a county–knight ruling arrangement if the resulting flow network has a flow (of 1) in the edge connecting them.

We can determine the algorithm successful if all counties have a knight assigned. This can be translated in a number of ways, you could say the solution is successful if the flow $T_{in} = n$.

To analyse the complexity we will be using the *Ford-Fulkerson* algorithm as described in the lectures and shown in pseudocode below.

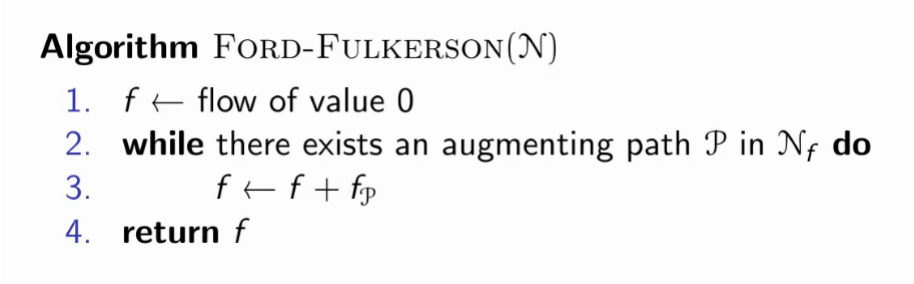


Figure 3: Ford-Fulkerson Max-Flow Algorithm from Lecture 10

where $\mathcal{N}$ is a flow network, $\mathcal{N}_f$ is the residual network and $\mathcal{P}$, the augmenting path, all as defined in lecture 10.

Analysing the complexity in terms of the function $F(v, e)$, we will show that the Ford-Fulkerson Algorithm runs in $\Theta(e \cdot f)$ where f is the maximum flow in the network [8].

[8] assuming that all capacities are integrals. , as they are in our model (and could be coerced by denominator multiplication if rational)

The initialisation in line 1 is trivial. In each iteration of the while loop, the flow value is increased by at least one up to a maximum of f, the maximum flow.

For the while loop condition, finding the augmenting path takes $\Theta(e)$ as we have a one off cost of initialising the residual network, taking $\Theta(v+e)$ and for each iteration it requires depth first search over the entire graph $\Theta(v+e)$. For a connected graph there must be at least $v-1$ edges, so we can rewrite $\Theta(v+e)$ in both cases as this as $\Theta(v+e) = \Theta(e+e) = \Theta(e)$. The depth first search $\Theta(e)$ is the only one that counts to our total runtime as it is multiplied by f the maximum number of loops. All other operations are trivial in comparison hence the total running time is

$$\underline{F(v,e) \in \Theta(e \cdot f)}$$

This is the general case, however in our model the maximum flow is bounded by m, the number of counties. So if we wanted we could rewrite as $\underline{F(v,e) \in \Theta(e \cdot m)}$. We could also write this as $O(e \cdot v)$ if we wanted a solution strictly in terms of v a e, as $m \in \emptyset(v)$.

It is also worth noting that a common implementation of the Ford-Fulkerson algorithm uses the *Edmonds-Karp Heuristic*, which instead of doing a depth first search to find the largest bottleneck, does a breadth first search and selects the shortest augmenting path [9]. We will show that that the total number of while loop iterations under this heuristic is $\Theta(v \cdot e)$, making for a total running time of $\Theta(v \cdot e^2)$.

As in depth first search each minimum augmenting path can be found in $\Theta(e)$.

In each iteration at least one edge, the bottleneck in $\mathcal{N}_f$, will become fully saturated and removed from the residual network, $\mathcal{N}_f$.

Edge (u,r) can only be re-appear on a minimum augmenting path in $\mathcal{N}_f$ if the flow is first reduced, i.e. the backwards edge (r,u) has featured in a different augmenting path.

When (u,r) was originally the bottleneck, distance on the augmenting path from source to r was 1 greater than that from source to u, now it is the reverse. For the backwards edge to be the shortest path, there must have been an increase in the augmenting path length of at least 2.

After (u,v) is a bottleneck, if u is reachable from source, its distance is bound by the number of vertices v, hence the total number of times it can be the bottle neck is bound by $v/2$ ($\Theta(v)$) as the path is monotonically increasing [10].

With e edges connecting the vertices in $\mathcal{N}_f$ the total number of bottlenecks (and hence iterations of the while loop) is $\Theta(v \cdot e)$. Combining this with the cost $\Theta(e)$ of our breadth first search gives.

$$\underline{F(v,e) \in \Theta(v \cdot e^2)}$$

[9] proof of the correctness of this strategy is given in lecture 11

[10] proof CLRS 3rd Ed, p749

(b) Merlin can use the *max-flow min-cut* theorem to convince the king. The *max-flow min-cut* theorem [11] states the maximum flow in a network, $\mathcal{N}$, (from source to target) is equal to the minimum capacity of a 'cut' [12] that would disconnect the source from the target.

Intuitively it can be seen that the min-cut partitions the graph in such a way as the flow must 'cross' the boundary which acts as a bottleneck. To find the minimum cut, take the final residual network $\mathcal{N}_f$ and locate the set of vertices reachable from the source in it. All edges $(reachable, unreachable)$ in the graph are minumum cut edges.

If the king accepts the *max-flow–min-cut* theorem to be true and Merlin can show him a minimum cut [13], which is less than the number of counties, m, then the king will be persuaded that there is no such solution to the instance of the problem as a solution requires a max-flow of exactly m (from counties to target).

[11] proved in lecture 11

[12] formally defined in lecture 11

[13] in fact any cut less that m will suffice