Capacitated Dynamic Programming: Faster Knapsack and Graph Algorithms

Kyriakos Axiotis
MIT
kaxiotis@mit.edu

Christos Tzamos
Microsoft Research
chtzamos@microsoft.com

Abstract

One of the most fundamental problems in Theoretical Computer Science is the Knapsack problem. Given a set of n items with different weights and values, it asks to pick the most valuable subset whose total weight is below a capacity threshold T. Despite its wide applicability in various areas in computer science, operations research, and finance, the best known running time for the problem is $O(Tn)$. The main result of our work is an improvement of this algorithm for the special case where the number of distinct weights D is small, running in time $O(TD)$. Previously, better algorithms for knapsack were known only for cases with weights and values bounded by M and V respectively, running in time $O(nMV)$ [13]. In comparison, our algorithm implies a bound of $O(nM^2)$ without any dependence on V. Additionally, for the unbounded knapsack problem, we provide an algorithm running in time $O(M^2 \log T)$.

We also initiate a systematic study of general capacitated dynamic programming, of which knapsack is core problem. This problem asks to compute the maximum weight path of length k in an edge- or node-weighted directed acyclic graph. In a graph with m edges, these problems are solvable by dynamic programming in time $O(km)$, and we explore under which conditions the dependence on k can be eliminated. We identify large classes of graphs where this is possible and apply our results to obtain linear time algorithms for the problem of k-sparse Δ-separated sequences. The main technical innovation behind our results is identifying and exploiting concavity that appears in relaxations and subproblems of the tasks we consider.

1 Introduction

A large number of problems in Computer Science can be formulated as finding the optimal subset of items to pick in order to maximize a given objective subject to capacity constraints.

A core problem in this class is the *Knapsack problem*: In this problem, each of the n items has a value and a weight and the objective is to maximize the total value of the selected items while having total weight at most T.

A standard approach for solving such capacitated problems is to use Dynamic Programming. Specifically, the dynamic programming algorithm keeps a state that tracks how much of the available capacity has already been exhausted. The runtime of these algorithms typically incurs a multiplicative factor equal to the total capacity. In particular, in the case of the Knapsack problem the classical dynamic programming algorithm due to Bellman [5] has a runtime of $O(nT)$.

In contrast, uncapacitated problems do not restrict the number of elements to be selected, but charge an extra cost for each one of them (i.e. they have a *soft* as opposed to a *hard* capacity constraint). The best known algorithms for these problems are usually much faster than the ones for their capacitated counterparts, i.e. for the uncapacitated version of knapsack one would need to pick all items whose value is larger than their cost. Therefore a natural question that arises is whether or when the additional dependence of the runtime on the capacity is really necessary.

In this work, we make progress towards answering this question by exploring when this dependence can be improved or completely eliminated.

Knapsack We first revisit the Knapsack problem and explore under which conditions we can obtain faster algorithms than the standard dynamic programming algorithm.

Despite being a fundamental problem in Computer Science, no better algorithms are known in the general case for over 60 years and it is known to be notoriously hard to improve upon. The best known algorithm for the special case where both the weights and the values of the items are small and bounded by M and V respectively, is a result by Pisinger [13] who presents an algorithm with runtime $O(nMV)$.

Even for the subset sum problem, which is a more restricted special case of knapsack where the value of every item is equal to its weight, the best known algorithm beyond the textbook algorithm by Bellman [5] was also an algorithm by Pisinger [13] which runs in time $O(nM)$ until significant recent progress by Bringmann [6] and Koiliaris and Xu [10] was able to bring its the complexity down to $\widetilde{O}(n + T)$.

However, recent evidence shows that devising a more efficient algorithm for the general Knapsack is much harder. Specifically, [8, 11] reduce the $(\max, +)$-convolution problem to Knapsack, proving that any truly subquadratic algorithm for Knapsack (i.e. $O((n + T)^{2-\varepsilon})$) would imply a truly subquadratic algorithm for the $(\max, +)$-convolution problem. The problem of $(\max, +)$-convolution is a fundamental primitive inherently embedded into a lot of problems and has been used as evidence for hardness for various problems in the last few years (e.g. [8, 11, 3]). However, an important open question remains here: *Can we do better under some reasonable assumption?*

We answer this question affirmatively in the regime where the number of *distinct* weights D is bounded and provide a simple algorithm running in time $O(TD)$. This directly implies a runtime of $O(TM)$ as the number of distinct integer weights is less than or equal to the maximum weight. Our algorithm partitions the items into D sets according to their weights and solves the knapsack problem in each set of the partition for every possible capacity up to T. This can be done efficiently in $O(T)$ time as all items in each set have the same weight and thus knapsack can be greedily solved in those instances. Having a sequence of solutions for every capacity level for each set of items allows us to obtain the overall solution by performing $(\max, +)$-convolutions among them. Even though computing $(\max, +)$-convolutions is generally hard to do faster than quadratic time, we exploit the inherent concavity of the sequences to perform this in linear time. We present our results in Section 3.1.

In addition to our main result for the standard Knapsack setting, we also consider the *Unbounded Knapsack* problem where there are infinite copies of every item. In Section 3.2, we present a novel algorithm for Unbounded Knapsack with running time $O(M^2 \log T)$, where M is the maximum weight of an item. Our algorithm again utilizes $(\max, +)$-convolutions of short sequences to compute the answer in time polynomial in the input representation of the capacity T and is only pseudopolynomial with respect to the maximum weight M.

Capacitated Dynamic Programming In addition to our results on the knapsack problem, we move on to study capacitated problems in a more general setting. Specifically, we consider the problem of computing a path of maximum reward between a pair of nodes in a weighted Directed

Acyclic Graph, where the capacity constraint corresponds to an upper bound on the length of the path.

This model has successfully been used for uncapacitated problems [15, 11], as well as capacitated problems with weighted adjacency matrices that satisfy a specific condition, namely the *Monge property* [2, 14, 4]. In [4], it is shown that under this condition, the maximum weight of a path of length k is *concave* in k. Whenever such a concavity property is true, one can always solve the capacitated problem by replacing the capacity constraint with an "equivalent" cost per edge. This cost can be identified through a binary search procedure that checks whether the solution for the uncapacitated problem with this cost corresponds to a path of length k.

Our second main result, Theorem 4.4, gives a complete characterization of such a concavity property for transitive node-weighted graphs. We show that this holds if and only if the following graph theoretic condition is satisfied:
For every path $a \to b \to c$ of length 2, and every node v, either the edge (a, v) or (v, c) exist.

To illustrate the power of our characterization, we show that linear time algorithms can be obtained for the problem of k-sparse Δ-separated subsequences [9] recovering recent results of [12, 7].

To complement our positive result which allows us to obtain fast algorithms for finding maximum weight paths of length k, we provide strong evidence of hardness for transitive node-weighted graphs which do not satisfy the conditions of our characterization. We base our hardness results on computational assumptions for the $(\max, +)$-convolution problem we described above.

Beyond node-weighted graphs, when there are weights on the edges, no non-trivial algorithms are known other than for Monge graphs. Even in that case, we show that linear time solutions exist only if one is interested in finding the max-weight path of length k between a pair of nodes. If one is interested in computing the solution in Monge graphs for a single source but all possible destinations, we provide an algorithm that computes this in near-linear time in the number of edges in the graph.

2 Preliminaries

We first describe the problems of 0/1 Knapsack and Unbounded Knapsack:

Definition 2.1 (0/1 Knapsack)**.** *Given n items with weights $w_1, \dots, w_n$ and values $v_1, \dots, v_n$, and a parameter T, our goal is to find a set of items $S \subseteq [n]$ of total weight at most T (i.e. $\sum_{i \in S} w_i \leq T$) that maximizes the total value $\sum_{i \in S} v_i$. We will denote the largest item weight by M and the number of distinct weights by D.*

Definition 2.2 (Unbounded Knapsack)**.** *Given n items with weights $w_1, \dots, w_n$ and values $v_1, \dots, v_n$, and a parameter T, our goal is to find a multiset of items $S \subseteq [n]$ of total weight at most T (i.e. $\sum_{i \in S} w_i \leq T$) that maximizes the total value $\sum_{i \in S} v_i$. We will denote the largest item weight by M and the number of distinct weights by D.*

Throughout the paper we make use of the following operation between two sequences called $(\max, +)$-convolution.

Definition 2.3 ($(\max, +)$-convolution)**.** *Given two sequences $a_0, \dots, a_n$ and $b_0, \dots, b_m$, the* $(\max, +)$-

convolution $a \oplus b$ between a and b is a sequence $c_0, \ldots, c_{n+m}$ such that for any i

$$c_i = \max_{0 \le j \le i} \{a_j + b_{i-j}\}$$

This operation is commutative, so it is also true that

$$c_i = \max_{0 \le j \le i} \{a_{i-j} + b_j\}$$

Our algorithms rely on uncovering and exploiting discrete concavity that is inherent in the problems we consider.

Definition 2.4 (Concave, k-step concave)**.** *A sequence $b_0, \ldots, b_n$ is* concave *if for all $i \in \{1, \ldots, n-1\}$ we have $b_i - b_{i-1} \ge b_{i+1} - b_i$. A sequence is called k*-step concave *if its subsequence $b_0, b_k, b_{2k}, \ldots$ is concave and for all i such that $i \mod k \neq 0$, we have that $b_i = b_{i-1}$.*

For the problems defined on graphs with edge weights, we typically assume that their weighted adjacency matrix is given by a Monge matrix.

Definition 2.5 (Monge matrices)**.** *A matrix $A \in \mathbb{R}^{n \times m}$ is called Monge if for any $1 \le i \le n-1$ and $1 \le j \le m-1$*

$$A_{i,j} + A_{i+1,j+1} \ge A_{i+1,j} + A_{i,j+1}$$

Definition 2.6 (Monge weights)**.** *We will say that a Directed Acyclic Graph has* Monge weights *if its adjacency matrix is a Monge matrix.*

In addition to our positive results, we present evidence of computational hardness assuming for $(\max, +)$-convolution problem.

Definition 2.7 ($(\max, +)$-convolution hardness)**.** *The $(\max, +)$-convolution hardness hypothesis states that any algorithm that computes the $(\max, +)$-convolution of two sequences of size n runs in time $\Omega(n^{2-o(1)})$.*

A result of [3] shows that the $(\max, +)$-convolution problem is equivalent to the following problem: Given an integer n and three sequences $a_0, \ldots, a_n$, $b_0, \ldots, b_n$, and $c_0, \ldots, c_n$, compute $\max\limits_{i+j+k=n} \{a_i + b_j + c_k\}$. In our conditional lower bounds, we will be using this equivalent form of the conjecture.

3 Knapsack

In this section we present two novel pseudo-polynomial deterministic algorithms, one for 0/1 Knapsack and one for Unbounded Knapsack. The running times of these algorithms significantly improve upon the best known running times in the small-weight regime. In essence, the main improvements stem from a more principled understanding and systematic use of $(\max, +)$-convolutions. Thus, we show that devising faster algorithms for special cases of $(\max, +)$-convolution lies in the core of improving algorithms for the Knapsack problem. In Theorem 3.1, we present an algorithm for 0/1 Knapsack that runs in time $O(TD)$, where T is the size of the knapsack and D is the number of distinct item weights. Then, in Theorem 3.5, we present an algorithm for Unbounded Knapsack that runs in time $O(M^2 \log T)$, where M is the maximum weight of an item.

3.1 0/1 Knapsack

Given N items with weights $w_1, \dots, w_N$ and values $v_1, \dots, v_N$, and a parameter T, our goal is to find a set of items $S \subseteq [N]$ of total weight at most T (i.e. $\sum_{i \in S} w_i \leq T$) that maximizes the total value $\sum_{i \in S} v_i$. We will denote the largest item weight by M and the number of distinct weights by D.

Algorithm 1 0/1 Knapsack

```
1: Given items with weights in $\{w_1^\#, \dots, w_D^\#\}$
2: Partition items into sets $S_1, \dots, S_D$, so that $S_i = \{j \mid w_j = w_i^\#\}$
3: for $i \in [D]$ and $t \in [T]$ do
4:     $b_t^{(i)} \leftarrow$ solution for $S_i$ with knapsack size $t$
5: $s \leftarrow$ empty sequence
6: for $i \in [D]$ do
7:     $s \leftarrow s \oplus b^{(i)}$ using Lemma 3.3
8:     Truncate $s$ after the $T$-th entry
9: Output $s_T$
```

The following is the main theorem of this section.

Theorem 3.1. *Algorithm 1 solves* 0/1 Knapsack *in time $O(TD)$.*

Overview The main ingredient of this result is an algorithm for fast $(\max, +)$-convolution in the case that one of the two sequences is *k-step concave*. Using the SMAWK algorithm [1] it is not hard to see how to do this in linear time for $k = 1$. For the general case, we show that computing the $(\max, +)$-convolution of the two sequences can be decomposed into $\frac{n}{k}$ subproblems of computing the $(\max, +)$-convolution between two size-k subsequences of the two sequences. Furthermore, the subsequence that came from k-step concave sequence is concave and so each subproblem can be solved in time $O(k)$ and the total time spent in the subproblems will be $O(\frac{n}{k}k) = O(n)$.

Lemma 3.2. *Given an arbitrary sequence $a_0, \dots, a_m$ and a concave sequence $b_0, \dots, b_n$ we can compute the $(\max, +)$ convolution between a and b in time $O(m + n)$.*

Proof. Consider the matrix A with $A_{ij} = a_j + b_{i-j}$, where we suppose that elements of the sequences with out-of-bounds indices have value ∞. Note now that $(a \oplus b)_i$ is by definition equal to the maximum value of row i of A. Therefore computing $a \oplus b$ corresponds to finding the row maximums of A. Now note that for any $(i, j) \in \{0, 1, \dots, n-1\} \times \{0, 1, \dots, m-1\}$, we have

$$\begin{aligned} A_{i,j} - A_{i,j+1} =& a_j + b_{i-j} - a_{j+1} - b_{i-j-1} \\ \overset{concavity}{\geq}& a_j + b_{i+1-j} - a_{j+1} - b_{i-j} \\ =& A_{i+1,j} - A_{i+1,j+1} \end{aligned}$$

therefore A is Monge. The main result of [1] is that given a Monge matrix $A \in \mathbb{R}^{n \times m}$, one can compute all its row maximums in time $O(m + n)$, which implies the Lemma. □

Lemma 3.3. *Given an arbitrary sequence $a_0, \dots, a_m$ and a k-step concave sequence $b_0, \dots, b_n$ we can compute the $(\max, +)$ convolution of a and b in time $O(m + n)$.*

Proof. We use the fact that we can compute the $(\max, +)$ convolutions of an arbitrary sequence with a concave sequence in linear time (Lemma 3.2). Since b is a k-step concave sequence, taking every k-th term of it one gets a concave sequence of size $O(n/k)$. Then, we do the same for a, taking k subsequences of size m/k each. Therefore we can compute the convolution between the concave sequence and all of these subsequences of a in linear time. The results of these convolutions can be used to compute the final sequence. We now describe this in detail.

For ease of notation, we will again assume that our sequences take value $+\infty$ in out-of-bounds indices. Let $x^{(i)} := (a_i, a_{k+i}, a_{2k+i}, \dots)$ denote the subsequence of a with indices whose remainder is i when divided by k, and $y := (b_0, b_k, b_{2k}, \dots)$. Furthermore, define

$$f_i = \max_{q=0}^{\infty}\{b_{qk} + a_{i-qk}\}$$

Now, for any j we have

$$\begin{aligned}\max_{i=j-k+1}^{j} f_i &= \max_{i=j-k+1}^{j} \max_{q=0}^{\infty}\{b_{qk} + a_{i-qk}\}\\ &= \max_{i=j-k+1}^{j} \max_{q=0}^{\infty}\{b_{qk+j-i} + a_{i-qk}\}\\ &= \max_{z=0}^{\infty}\{b_z + a_{j-z}\}\end{aligned}$$

where the second equality follows from the fact that $b_{qk+t} = b_{qk}$ for any $t \in [k-1]$ and the third from the fact that $z = qk + j - i$ can take any value in $[0, \infty)$.

This is the j-th element of the $(\max, +)$-convolution between a and b, so the elements of this convolution are exactly the minimums of size-k segments of f.

In order to compute f, note that for some p, the convolution between $x^{(p)}$ and y gives us all values of f of the form f_{qk+p}, for any q. This is because from the definition of f,

$$\begin{aligned}f_{qk+p} &= \min_{z=0}^{\infty}\{b_{zk} + a_{qk+p-zk}\}\\ &= \min_{z=0}^{\infty}\{y_z + x^{(p)}_{q-z}\}\\ &= (x^{(p)} \oplus y)_q\end{aligned}$$

Furthermore, y is a concave sequence and by Lemma 3.2 we can compute such a convolution in time $O((m+n)/k)$. Doing this for all $0 \le p < k$, we can compute all values of f in time $O(m+n)$.

Now, in order to compute the target sequence, we have to compute the minimums of all size-k segments of f. We can do that using a simple sliding window technique. Specifically, suppose that for some segment $[i, i+k-1]$ we have an increasing subsequence of $f_{[i,\dots,i+k-1]}$, containing all the potentially useful elements. The first element of this subsequence is the minimum value of f in the segment $[i, i+k-1]$. Now, to move to $[i+1, i+k]$, we remove f_i if it is in the subsequence, and then we compare f_{i+k} with the last element in the subsequence. Note that if that last element has value $\geq f_{i+k}$, it will never be the minimum element in any segment. Therefore we can remove it and repeat until the last element has value less than f_{i+k}, at which point we just insert f_{i+k} in the end of the subsequence. Note that by construction, this subsequence will always be increasing, and the first element will be the minimum of the respective segment. The total runtime is linear if we implement it with a standard queue. $\square$

Now that we have these tools we can use them to prove the main result of this section:

Proof of Theorem 3.1. Consider any knapsack instance where D is the number of distinct item weights $w_1^{\#}, \dots, w_D^{\#}$. Now for each $i \in [D]$ let c_i be the number of items with weight $w_i^{\#}$ and $v_1^{(i)} \geq v_2^{(i)} \cdots \geq v_{c_i}^{(i)}$ their respective values.

If we only consider items with weights $w_i^{\#}$, the knapsack problem is easy to solve, since we will just greedily pick the most valuable items until the knapsack fills up. More specifically, if b_s is the maximum value obtainable with a knapsack of size s, we have that $b_0 = 0, b_{w_i} = v_1^{(i)}, b_{2w_i} = v_1^{(i)} + v_2^{(i)}, \dots$, and also $b_j = b_{j-1}$ for any j not divisible by $w_i^{\#}$. Therefore b is a $w_i^{\#}$-step concave sequence.

In order to compute the full solution, we have to compute the $(\max, +)$ convolution of D such sequences. Since by Lemma 3.3 each convolution takes linear time and we only care about the first T values of the resulting sequence (i.e. we will only ever keep the first T values of the result of a convolution), the total runtime is $O(TD)$, where T is the size of the knapsack. $\square$

Corollary 3.4. *0-1 knapsack can be solved in $O(TM)$ time.*

3.2 Unbounded Knapsack

Given N items with weights $w_1, \dots, w_N$ and values $v_1, \dots, v_N$, and a parameter T, our goal is to find a multiset of items $S \subseteq [N]$ of total weight at most T (i.e. $\sum_{i \in S} w_i \leq T$) that maximizes the total value $\sum_{i \in S} v_i$. We will denote the largest item weight by M.

Note that this problem is identical to 0/1 Knapsack except for the fact that there is no limit on the number of times each item can be picked. This means that we can assume that there are no two items with the same weight, since we would only ever pick the most valuable of the two.

Algorithm 2 Unbounded Knapsack

```
1: Let $v^{(0)}$ be a sequence where $v_x^{(0)}$ is the value of the element with weight $x$ or $-\infty$ if no such
   element exists
2: for $z = 1, \dots, \lceil \log M \rceil$ do
3:     $v^{(z)} \leftarrow v^{(z-1)} \oplus v^{(z-1)}$
4: $a_{[0,M]} \leftarrow v^{(\lceil \log M \rceil)}$
5: for $i = \lceil \log \frac{T}{M} \rceil, \dots, 1$ do
6:     $a_{\left[\frac{T}{2^i}-M, \frac{T}{2^i}+M\right]} \leftarrow a_{\left[\frac{T}{2^i}-M, \frac{T}{2^i}\right]} \oplus a_{[0,M]}$
7:     $a_{\left[\frac{T}{2^{i-1}}-M, \frac{T}{2^{i-1}}\right]} \leftarrow a_{\left[\frac{T}{2^i}-M, \frac{T}{2^i}+M\right]} \oplus a_{\left[\frac{T}{2^i}-M, \frac{T}{2^i}+M\right]}$
8: Output $a_T$
```

The following is the main theorem of this section:

Theorem 3.5. *Algorithm 2 solves* Unbounded knapsack *in time $O(M^2 \log T)$.*

Overview As in the algorithm for 0/1 Knapsack our algorithm utilizes $(\max, +)$-convolutions, but with a different strategy. We aren't using any concavity arguments here, but in fact we will use the straightforward quadratic-time algorithm for computing $(\max, +)$-convolutions. The main argument here is that if all the weights are relatively small, one can always partition any solution in two, so that the weights of the two parts are relatively close to each other. Therefore, for any knapsack size we only have to compute the optimal values for a few knapsack sizes around its half, and not for all possible knapsack sizes.

We can now proceed to the proof of this result.

Proof of Theorem 3.5. Consider any valid solution to the unbounded knapsack instance. Since every item has weight at most M, we can partition the items of that solution into two multisets with respective weights W_1 and W_2, so that $|W_1 - W_2| < M$ (one can obtain this by repeatedly moving any item from the larger part to the smaller one). This implies the following, which is the main fact used in our algorithm: If a_s is the maximum value obtainable with a knapsack of size s, then we have that

$$a_s = \left[(a_{s/2-M/2}, \dots, a_{s/2+M/2})^{\oplus 2}\right]_s$$

where ${}^{\oplus 2}$ denotes $(\max, +)$-convolution squaring, i.e. applying $(\max, +)$-convolution between a sequence and itself.

First, we compute the values $a_1, \dots, a_M$ in $O(M^2 \log M)$ time as follows: We start with the sequence $v^{(0)}$, where $v^{(0)}_x$ is the value of the element with weight x, or $-\infty$ if such an item does not exist. Now define $v^{(i+1)} = (v^{(i)} \oplus v^{(i)})_{[0,M]}$. This convolution can be applied in time $O(M^2)$ for any i, since we are always only keeping the first M entries. By induction, it is immediate that $v^{(i)}$ contains the optimal values achievable for all knapsack sizes in $[M]$ using at most 2^i items. Therefore $a_{0,\dots,M} \equiv v^{(\lceil \log M \rceil)}_{0,\dots,M}$, which as we argued can be computed in time $O(M^2 \log M)$.

Now, suppose that we have computed the values $a_{\frac{T}{2^i}-M}, \dots, a_{\frac{T}{2^i}}$ for some i. By convolving this sequence with $a_0, \dots, a_M$ we can compute in time $O(M^2)$ the values $a_{\frac{T}{2^i}+1}, \dots, a_{\frac{T}{2^i}+M}$. Now, convolving the sequence $a_{\frac{T}{2^i}-M}, \dots, a_{\frac{T}{2^i}+M}$ with itself gives us $a_{\frac{T}{2^{i-1}}-M}, \dots, a_{\frac{T}{2^{i-1}}}$ (here we used the fact that to compute a_{2j} we only need the values $a_{j-M/2}, \dots, a_{j+M/2}$). Doing this for $i = \lceil \log T \rceil, \dots, 2, 1$, we are able to compute the values $a_{T-M}, \dots, a_T$ in total time $O(M^2 \log T)$. The answer to the problem, i.e. the maximum value achievable, is $\max\{a_{T-M}, \dots, a_T\}$. $\square$

4 Capacitated Dynamic Programming

We now move on to study more general capacitated dynamic programming settings, described by computing the maximum reward path of length k in a directed acyclic graph. This setting can capture a lot of natural problems, either directly or indirectly. In the following theorem, we show that the Knapsack problem is a special case of this model and thus a better understanding of Knapsack can lead to improved algorithms for other capacitated problems.

Lemma 4.1 (Knapsack)**.** *The Knapsack problem can be modeled as finding a maximum reward path with at most k edges in a node-weighted transitive DAG.*

Proof. Let the item weights and values be $w_1, \dots, w_n$ and $v_1, \dots, v_n$ respectively. We will create a DAG for each item and then join all these DAGs in series. Specifically, for item i, its DAG G_i will consist of two parallel paths Y_i and N_i between a pair of nodes s_i and t_i. Y_i will correspond to taking item i, and N_i to not taking it.

- N_i will be a path of length 2 from s_i to t_i, where the intermediate vertex has reward $b = n w_{max} v_{max}$.

- Y_i will be a path of length $w_i + 2$ from s_i to t_i, all of which vertices other than s_i, t_i have reward $\frac{b+v_i}{w_i+1}$.

Finally, we just join all G_i in series, i.e. for all $i \in [n-1]$, identify t_i with s_{i+1}, and we ask for the maximum reward path with at most $n+T$ edges from s_1 to t_n.

Note that we couldn't have just set the reward of path N_i to 0, because that would potentially allow one to pick a path that is a subset of Y_i, which corresponds to picking a fraction of an item and is invalid. However, note that with our current construction any optimal path will either use the whole path Y_i, or it will not use it at all. This is because the reward of picking 1 vertex from N_i is b, while the reward of picking at most w_i vertices from Y_i is at most $b + v_i - \frac{b+v_i}{w_i+1} = b - \frac{b-w_iv_i}{w_i+1} = b - \frac{nw_{max}v_{max}-w_iv_i}{w_i+1} < b$.

Now, for each i, the optimal path will definitely contain either Y_i or N_i. If this were not the case, the reward of the solution would be at most

$$\sum_i (b+v_i) - \min_i\{b+v_i\} < nb + \sum_i v_i - b \le nb + nv_{max} - nw_{max}v_{max} < nb$$

while by simply picking all N_i's one gets reward nb.

Therefore we have shown that the total reward of the optimal solution will be

$$\sum_{i\in S}(b+v_i) + b(n-|S|) = nb + \sum_{i\in S} v_i$$

for some set S such that $n+T \ge \sum_{i\in S}(w_i+1) + (n-|S|) = n + \sum_{i\in S} w_i$, or equivalently $\sum_{i\in S} w_i \le T$. Therefore this set S is the optimal set of items to be picked in the knapsack.

□

4.1 Node-weighted Graphs

In this section, we study the problem of finding maximum-reward paths in node-weighted transitive DAGs. In Lemma 4.2, we show that in general this problem is hard, by reducing $(\max,+)$-convolution to it. We then proceed to show our second main result, which provides a family of graphs for which the problem can be efficiently solved.

Lemma 4.2 ($(\max,+)$-hardness of Node-weighted graphs). *Given a transitive DAG, a pair of vertices s and t, and an integer k, the problem of computing a maximum reward path from s to t with at most k edges is $(\max,+)$-convolution hard, i.e. requires $\Omega((mk)^{1-o(1)})$ time assuming $(\max,+)$-convolution hardness.*

Proof. Given a sequence $x_0,\dots,x_k$, we construct the following node-weighted DAG, on nodes $a_0,\dots,a_k,a'_0,\dots,a'_k$. For all $i \in [k]$, we add edge (a_{i-1},a_i) and for all $0 \le i \le k$, we add edge (a_i,a'_i). Let $M = 3\max\{|x_0|,\dots,|x_k|\}$ and $T = 10Mk$. If we denote the value of node z as $val(z)$, we define $val(a_i) = M$ and $val(a'_i) = T + x_i - Mi$ for all i.

Now consider three such DAGs, one for each subsequence $x_0,\dots,x_k$, $y_0,\dots,y_k$, and $z_0,\dots,z_k$, with the node sets being $a_\star$ and $a'_\star$, $b_\star$ and $b'_\star$, and $c_\star$ and $c'_\star$ respectively. We connect them in series, i.e. each node of the first DAG has an edge to b_0, and each node of the second DAG has an edge to c_0. Then we take the transitive closure of the resulting DAG.

Note that any maximum reward path with $k+5$ hops on this DAG will necessarily use some $a'_\star$, $b'_\star$, and $c'_\star$. If this were not the case, the value to be obtained would be less than

$$M(k+4)+2(T+M) \leq 2.5T$$

However, a path containing some a'_i, b'_j, c'_l for some i, j, l will have weight at least

$$3T - 3Mk \geq 2.5T$$

Furthermore, the first nodes of the path will be of the form $a_0, a_1, \ldots, a_i, a'_i$. To see this, suppose otherwise, i.e. that the path contains $j+1 < i+1$ nodes of the form $a_\star$. Then the total value of the part of the path up to a'_i will be

$$\begin{aligned} M(j+1)+T+x_i-Mi &\leq M(j+1)+T+x_i-M-Mj \\ &< M(j+1)+T+x_j-Mj = a_0+a_1+\cdots+a_j+a'_j \end{aligned}$$

so the path $a_0, a_1, \ldots, a_j, a'_j$ is always better and has the same number of edges. A similar argument can be applied to the rest of the path, to show that the total maximum length $(k+5)$-hop path will be of the form $a_0, \ldots, a_{k_1}, a'_{k_1}, b_0, \ldots, b_{k_2}, b'_{k_2}, c_0, \ldots, c_{k_3}, c'_{k_3}$, with $k_1+k_2+k_3=k$. The only remaining case is that of the path containing some edge (a'_{k_1}, b'_j), for some j (and similarly for (b'_{k_2}, c_j)). However in this case we can find a better path. Suppose that $k_1 \geq 1$ (otherwise we can do it for k_3). Replace edges (a_0, a_1) and (a_1, u) of the path with edge (a_0, u), essentially skipping over a_1, and also replace edge (a'_{k_1}, b'_j) by edges (a'_{k_1}, b_0) and (b_0, b'_j). Note that both the value and the length remained the same, but we use less than k_1+1 $a_\star$ nodes, so this path is not optimal, as seen by the argument we stated before.

In light of the above, a maximum-weight $(k+5)$-hop path in this graph will be of the form

$$a_0, \ldots, a_{k_1}, a'_{k_1}, b_0, \ldots, b_{k_2}, b'_{k_2}, c_0, \ldots, c_{k_3}, c'_{k_3}$$

where $k_1+k_2+k_3=k$, and have value equal to $3T+x_{k_1}+y_{k_2}+z_{k_3}$. Therefore computing $\max_{k_1+k_2+k_3=k}\{x_{k_1}+y_{k_2}+z_{k_3}\}$ is equivalent to finding the maximum value $(k+5)$-hop path on this DAG. Note that if given in succinct form the number of edges is linear in k, so any $o(mk)$ algorithm for this problem yields an $o(k^2)$ algorithm for $(\max,+)$-convolution. ☐

As we saw in the introduction, one can solve the problem if the optimal value as a function of the capacity is concave. This is made formal in the following lemma:

Lemma 4.3 (Concave functions). *Let G be a node-weighted transitive DAG with n vertices and m edges, whose weights' absolute values are bounded by M, and let $f(x)$ be the maximum reward obtainable in a path of length x. If f is a concave function, then one can reduce the capacitated problem (i.e. computing $f(k)$ for some k) to solving $O(\log(nM))$ uncapacitated problems with some fixed extra cost per item. Since each one of these problems can be solved in $O(m)$ time, the total runtime is $O(m\log(nM))$.*

In Theorem 4.4 we give a complete graph-theoretic characterization of the graphs that have this concavity property and therefore can be solved efficiently.

Theorem 4.4 (Concavity characterization). *The problem of finding a maximum reward path with at most k edges in a transitive DAG is concave for all choices of node weights if and only if for any path $u_1 \to u_2 \to u_3$ and any node v either $u_1 \to v$ or $v \to u_3$ (Property $\mathcal{P}$).*

Proof. Let $f(k)$ be the maximum reward obtainable with a path of exactly k edges.
$\Rightarrow$: Let G be a DAG for which property $\mathcal{P}$ doesn't hold. Let $u_1 \to u_2 \to u_3$ be the path of length 2 and v be the vertex that has no edge to or from any of u_1, u_2, u_3. We set the node values as $val(u_1) = val(u_2) = val(u_3) = 1$, $val(v) = 1 + \varepsilon$, and $-\infty$ for all other vertices. Then, $f(1) = 1 + \varepsilon$, $f(3) = 3$, but $f(2) = 2 < \frac{f(1)+f(3)}{2}$, therefore f is not concave.
$\Leftarrow$: Suppose that property $\mathcal{P}$ is true. Now, let $P = (s, p_1, p_2, \ldots, p_{k-1}, t)$ be a path of length k such that $val(P) = f(k)$ and $Q = (s, q_1, q_2, \ldots, q_{k+1}, t)$ be a path of length $k+2$ such that $val(Q) = f(k+2)$, where P and Q can potentially have common vertices other than s and t. Since property $\mathcal{P}$ is true, we know that for any $i \in [k-1]$, there is either an edge from one of q_i, q_{i+1}, q_{i+2} to p_i, or from p_i to one of q_i, q_{i+1}, q_{i+2}. By transitivity, this implies that either $q_i \to p_i$, or $p_i \to q_{i+2}$. We distinguish three cases. In all three cases we will be able to find paths P' and Q' with $k+1$ edges each, that contain all vertices of the form p_i and q_i.

Case 1: $q_1 \to p_1$
We pick $P' = (s, q_1, p_1, \ldots, p_{k-1}, t)$ and $Q' = (s, q_2, \ldots, q_{k+1}, t)$.

Case 2: $\forall i : p_i \to q_{i+2}$
We pick $P' = (s, p_1, \ldots, p_{k-1}, q_{k+1}, t)$ and $Q' = (s, q_1, \ldots, q_k, t)$

Case 3: $\exists i : p_i \to q_{i+2}$, $q_{i+1} \to p_{i+1}$
We pick $P' = (s, p_1, \ldots, p_i, q_{i+2}, \ldots, q_{k+1}, t)$ and $Q' = (s, q_1, \ldots, q_{i+1}, p_{i+1}, \ldots, p_{k-1}, t)$.

Therefore we established that in any case there exist such paths P' and Q'. Now note that

$$\max\left\{val(P'), val(Q')\right\} \geq \frac{1}{2}\left(val(P') + val(Q')\right) = \frac{1}{2}\left(val(P) + val(Q)\right)$$

and therefore f is a concave function. □

As mentioned before, even very simple special cases of the model capture a lot of important problems. In the following lemma, we show that we can solve the k-sparse Δ-separated subsequence problem [9] in near-linear time using the main result of this section, thus recovering recent results of [12, 7].

Lemma 4.5 (Maximum-weight k-sparse Δ-separated subsequence)**.** *Given a sequence $a_1, \ldots, a_n$, find indices $i_1, i_2, \ldots, i_k$ such that for all $j \in [k-1]$, $i_{j+1} \geq i_j + \Delta$ and the sum $\sum_{j \in [k]} a_{i_j}$ is maximized. This problem can be solved in $O(n \log(n \max_i |a_i|))$ time.*

Proof. Let's define a simple node-weighted DAG for this problem. We define a sequence of vertices $u_1, \ldots, u_n$ each one of which corresponds to picking an element from the sequence. Then, we add an edge $u_i \to u_j$ iff $j - i \geq \Delta$. Furthermore, for all i, $val(u_i) = a_i$. It remains to prove that it satisfies the property of Theorem 4.4. Consider any length-2 path $u_i \to u_j \to u_k$. We know that both $k - j$ and $j - i$ are at least Δ. Now, for any u_p we have that

$$\max\{|u_p - u_k|, |u_p - u_i|\} \geq \frac{1}{2}\left(|u_p - u_k| + |u_p - u_i|\right) \geq \frac{1}{2}\left(|u_k - u_i|\right) \geq \frac{1}{2}2\Delta = \Delta$$

so there is an edge between u_p and either u_i or u_k. Therefore by Lemma 5.1 the problem can be solved in time $O(m \log(n \max_i a_i)) = O(n^2 \log(n \max_i a_i))$.

The quadratic runtime stems from the fact that the DAG we constructed is dense. In fact, we can do better by defining some auxiliary vertices $v_1, \ldots, v_n$. The values of these extra vertices will be set to $-\infty$ to ensure that they aren't used in any solution and thus don't break the concavity. Instead of edges between vertices u_i, we only add the following edges

- $u_i \to v_i$ for all i
- $v_i \to u_{i+\Delta}$ for all $i + \Delta \leq n$
- $v_i \to v_{i+1}$ for all $i + 1 \leq n$

Now, the number of edges is $O(n)$ and so the runtime becomes $O(n \log(n \max_i |a_i|))$. $\square$

As another example of a problem that can be expressed as capacitated maximum-reward path in a DAG, we consider the Max-Weight Increasing Subsequence of length k problem. In contrast to its uncapacitated counterpart, which is solvable in linear time, the capacitated version requires quadratic time, assuming $(\max, +)$-convolution hardness.

Lemma 4.6 (Max-Weight Increasing Subsequence of length k)**.** *Given a sequence $a_1, \ldots, a_n$ with respective weights $w_1, \ldots, w_n$, find indices $i_1 < i_2 < \cdots < i_k$ such that for all $j \in [k-1]$, $a_{i_j} \leq a_{i_{j+1}}$ and the sum $\sum\limits_{j \in [k]} w_{i_j}$ is maximized. This problem is $(\max, +)$-convolution hard, i.e. requires $\Omega((nk)^{1-o(1)})$ time assuming $(\max, +)$-convolution hardness.*

Proof. Consider the construction used in Lemma 4.2. We define an instance of the Max-Weight Increasing Subsequence of length k problem which contains an element for each node of the DAG. Specifically, let's define our sequence to be

$$x_0, \ldots, x_k, x'_0, \ldots, x'_k, y_0, \ldots, y_k, y'_0, \ldots, y'_k, z_0, \ldots, z_k, z'_0, \ldots, z'_k$$

$$\begin{aligned} \text{with} \quad & x_i = i & x'_i &= 2k + 1 - i \\ & y_i = 2k + 2 + i & y'_i &= 4k + 3 - i \\ & z_i = 4k + 4 + i & z'_i &= 6k + 5 - i \end{aligned}$$

where the weight of each element is equal to the weight of the corresponding node in the DAG (i.e. $x_\star \leftrightarrow a_\star$, $x'_\star \leftrightarrow a'_\star$, $y_\star \leftrightarrow b_\star$, $y'_\star \leftrightarrow b'_\star$, $z_\star \leftrightarrow c_\star$, $z'_\star \leftrightarrow c'_\star$)

By definition of the sequence, the fact that we are looking for increasing subsequences implies that there is a $1{-}1$ correspondence between length-k increasing subsequences and $(k{-}1)$-hop paths of the original DAG. Therefore any $o((nk)^{1-\varepsilon})$ algorithm for the Max-Weight Increasing Subsequence of length k problem implies a truly subquadratic algorithm for the $(\min, +)$-convolution problem. $\square$

5 Graphs with Monge Weights

In this section we study the problem of computing maximum-reward paths with at most k edges in a DAG with edge weights satisfying the Monge property. Using the elegant algorithm of [4], one can compute a single such path in $\widetilde{O}(n)$ time.

Lemma 5.1 (From [4])**.** *Given a DAG with Monge weights, with n vertices, a pair of vertices s and t, and a positive integer k, we can compute a maximum reward path from s to t that uses at most k edges, in time $\widetilde{O}(n)$.*

Given the adjacency matrix A of the DAG, one can see this equivalently as computing one element of the matrix power A^k in the tropical semiring (i.e. we replace $(+, \cdot)$ with $(\max, +)$). Therefore, an important question is whether a whole row or column of A^k can be computed efficiently rather. This corresponds to finding maximum reward paths with k edges from some vertex s to *all* other vertices, or finding maximum reward paths with k edges from some vertex s to some vertex t for all k. In Lemma 5.2 we show that one needs $\Omega(n^{3/2})$ time to compute a column of A^k in general.

Lemma 5.2. *Given a DAG with Monge weights, with n vertices, computing the maximum weight path of length k from a given s to all other nodes t requires $\Omega(n^{1.5})$ time.*

On the positive side, by further exploiting the Monge property, in Lemma 5.3 we present an algorithm that can compute any row or column of A^k in $\widetilde{O}(nnz(A)) = \widetilde{O}(m)$ time.

Lemma 5.3. *Let G be a DAG of n vertices and m edges equipped with Monge weights that are integers of absolute value at most M. Given a vertex s, and a positive integer k, we can compute a maximum reward path from s to t that uses at most k edges, for all t, in time $O(m \log n \log(nM))$. Furthermore, if we are given a pair of vertices s and t, we can compute as maximum reward path from s to t that uses at most k edges, for all $k \in [n]$, in time $O(m \log n \log(nM))$.*

Acknowledgements. We are grateful to Arturs Backurs for insightful discussions that helped us improve this work.

References

[1] Alok Aggarwal, Maria M Klawe, Shlomo Moran, Peter Shor, and Robert Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2(1-4):195–208, 1987.

[2] Alok Aggarwal, Baruch Schieber, and Takeshi Tokuyama. Finding a minimum-weightk-link path in graphs with the concave monge property and applications. *Discrete & Computational Geometry*, 12(3):263–280, 1994.

[3] Arturs Backurs, Piotr Indyk, and Ludwig Schmidt. Better approximations for tree sparsity in nearly-linear time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2215–2229. SIAM, 2017.

[4] WW Bein, LL Larmore, and JK Park. The d-edge shortest-path problem for a monge graph. Technical report, Sandia National Labs., Albuquerque, NM (United States), 1992.

[5] Richard Bellman. Dynamic programming (dp). 1957.

[6] Karl Bringmann. A near-linear pseudopolynomial time algorithm for subset sum. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1073–1084. Society for Industrial and Applied Mathematics, 2017.

[7] Henning Bruhn and Oliver Schaudt. Fast algorithms for delta-separated sparsity projection. *arXiv preprint arXiv:1712.06706*, 2017.

[8] Marek Cygan, Marcin Mucha, Karol Wegrzycki, and Michal Wlodarczyk. On problems equivalent to (min, +)-convolution. In *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, pages 22:1–22:15, 2017.

[9] Chinmay Hegde, Marco F Duarte, and Volkan Cevher. Compressive sensing recovery of spike trains using a structured sparsity model. In *SPARS'09-Signal Processing with Adaptive Sparse Structured Representations*, 2009.

[10] Konstantinos Koiliaris and Chao Xu. A faster pseudopolynomial time algorithm for subset sum. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1062–1072. SIAM, 2017.

[11] Marvin Künnemann, Ramamohan Paturi, and Stefan Schneider. On the fine-grained complexity of one-dimensional dynamic programming. In *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, pages 21:1–21:15, 2017.

[12] Aleksander Mądry, Slobodan Mitrović, and Ludwig Schmidt. A fast algorithm for separated sparsity via perturbed lagrangians. *arXiv preprint arXiv:1712.08130*, 2017.

[13] David Pisinger. Linear time algorithms for knapsack problems with bounded weights. *Journal of Algorithms*, 33(1):1–14, 1999.

[14] Baruch Schieber. Computing a minimum weightk-link path in graphs with the concave monge property. *Journal of Algorithms*, 29(2):204–222, 1998.

[15] Robert Wilber. The concave least-weight subsequence problem revisited. *Journal of Algorithms*, 9(3):418–425, 1988.

A Omitted Proofs

Proof of Lemma 4.3. Instead of computing $f(k)$, we will solve the Lagrangian relaxation $\max_k \{f(k) - \lambda k\}$, where $\lambda \geq 0$ is some parameter. Note that usually this problem is much easier to solve, since one can incorporate an extra cost of λ to the value of each item. This can be solved by subtracting λ from all node weights and computing the maximum-reward path and so can be done in linear time. Because of the concavity of f, this corresponds to finding the uppermost intersection point of the function $f(k)$ with a line with slope λ. Let this point be $(k', f(k'))$. Therefore we have computed $f(k')$. If $k > k'$, then we should in fact be looking for smaller λ, and if $k < k'$ we should be looking for larger λ. This yields a binary search algorithm that eventually finds $(k, f(k))$ as the intersection of f with a line with slope λ. Therefore the total runtime will be $O(m \log(nM))$. □

Proof of Lemma 5.2. Consider a complete DAG in which for any $i < j$, $w(i,j) = (i-j)^2$, and we wish to minimize the total weight. To see that these weights are Monge, note that

$$\begin{aligned}(i-j)^2 + (i+1-(j+1))^2 - (i-(j+1))^2 - (i+1-j)^2 \\ = -2(ij + (i+1)(j+1) - (i+1)j - i(j+1)) \\ = -4 < 0\end{aligned}$$

We will prove that the set of edges used by the solutions has size $\Omega(n^{1.5})$. Note that the optimal path from 0 to n with k edges will only contain edges of lengths $\lfloor \frac{n}{k} \rfloor$ and $\lceil \frac{n}{k} \rceil$. To see this, note that otherwise the path must consist of two edges whose lengths differ by at least 2. Suppose these lengths are a and $b \geq a+2$. But since the cost of an edge depends just on its length, we can replace them by the edges $a+1$ and $b-1$, thus decreasing the cost by $a^2+b^2-(a+1)^2-(b-1)^2 = 2(b-a-1) > 0$. We will suppose wlog that all the larger edges are in the beginning of the path, i.e. closer to 0 than the smaller ones.

Now, consider any edge $(i, i+x)$. If the following three conditions are true, this edge is part of the solution.

- $\lfloor \frac{i}{x} \rfloor \geq x-1$
- $k(x+1) \leq n$
- $\lceil \frac{i+x}{x} \rceil \leq k$

Suppose that these three conditions are met. We will create an optimal solution containing edge $(i, i+x)$. By the third condition, the path $a, a+x, a+2x, \ldots, i, i+x$, for some $0 \leq a < x$ contains at most k edges. Furthermore, since $a < x$, by the first condition we can increase the lengths of the first a edges by 1, so that the new path starts at 0. Finally, extend this path to the right by length-x edges so that it is a k-hop path. This can be done because of the second condition. Therefore we have a path that is optimal for some endpoint and contains (i, x).

This means that the total number of edges is at least $\sum_{x=1}^{\lfloor \frac{n}{k} \rfloor -1} [(k-1)x - x(x-1)] \geq \Theta(n^{3/2})$, where we have picked $k = \Theta(\sqrt{n})$ in order to maximize the sum.

Now adding some small amount of arbitrary noise to all the edges ensures that we have to look at all the edges in the solution just to compute the weights of all the solutions.

□

Proof of Lemma 5.3.

The proof is based on the following lemma.

Lemma A.1. *Consider a DAG with Monge weights, nodes indexed by $0, \ldots, n$ in order. Let $P := (P_0 = 0, P_1, \ldots, P_{k_1} = n)$ be a maximum reward path from 0 to n, and $Q := (Q_0 = 0, Q_1, \ldots, Q_{k_2} = n)$ be a maximum reward path from 0 to n but in the DAG where all edge weights are increased by the same positive number (obviously the DAG is still Monge). There exists a choice of Q such that $Q_1 \geq P_1$.*

Proof. It is easy to see that it suffices to show this for $k_1 = k_2 + 1$, so let $k_2 = k$ and $k_1 = k+1$. Suppose that $P_1 > Q_1$. Such a pair of paths P, Q can be equivalently described as follows: Let's visualize the paths in their topological order, and scan with a vertical line from left to right, while keeping a point (x, y) in the integer plane, starting from $(0,0)$. Every time the vertical line meets an endpoint of some edge in P we move from (x, y) to $(x, y+1)$, every time it meets and endpoint of some edge in Q we move to $(x+1, y)$, and if it meets a common endpoint of both we move to $(x+1, y+1)$. It is easy to see that at any time, x (resp. y) is the number of edges of Q (resp. P) already encountered by the vertical line. Therefore, since the size of P is $k+1$ and the size of Q is k, we will end up at $(k, k+1)$. Now, $P_1 > Q_1$ states the fact that after $(0,0)$ we move to $(1,0)$ and this means that eventually we will have to cross the line $x = y$.

Furthermore, if we ever move on that line, say e.g. from some (x, x) to $(x+1, x+1)$, this means that the paths seen so far are interchangeable and so the path $Q' := (P_0, \ldots, P_{x+1}, Q_{x+2}, \ldots, Q_k)$ has the same number of hops and weight as Q, but also $Q'_1 = P_1$.

Otherwise, at some point we have to move from some $(x, x-1)$ to (x, x) and then to $(x, x+1)$. By the Monge property, this means that $Q_x < P_x < P_{x+1} < Q_{x+1}$ and so

$$w(Q_x, Q_{x+1}) + w(P_x, P_{x+1}) \geq w(Q_x, P_{x+1}) + w(P_x, Q_{x+1})$$

So if we define the paths

$$P' := (Q_0, \ldots, Q_x, P_{x+1}, \ldots, P_{k+1})$$

and

$$Q' := (P_0, \ldots, P_x, Q_{x+1}, \ldots, Q_k)$$

by the Monge property and optimality of P, Q we know that P' has the same weight as P and Q' the same weight as Q. Furthermore, Q' also has $Q'_1 = P_1$. □

Suppose that we add the number λ to all the weights and then find the maximum-weight path that ends at node n. This path has $f(\lambda)$ edges for some function $f : \mathbb{R} \to [n]$. It is easy to see that the function f is decreasing and takes all values in $[n]$ for which there exists a path of that length from 0 to n. Furthermore, if $w(\lambda)$ is the maximum weight of a path P after adding λ to all weights, we know that $w(\lambda) = w(P) + \lambda f(\lambda)$. So among all paths with $f(\lambda)$ edges, P maximizes $w(\lambda)$, so it also maximizes $w(P)$. Therefore, that path is the maximum weight path with $f(\lambda)$ hops.

Suppose that $a_\lambda(i)$ is the node after i in this path, and that $l_\lambda(i)$ (resp. $r_\lambda(i)$) is the number of edges of the form $(i, \star)$ that are shorter (resp. longer) than (i, a_i).

By Lemma A.1 we know that when looking at paths with $\lambda' > \lambda$, we will have $a'_i \geq a_i$, and when looking at paths with $\lambda' < \lambda$ we will have $a'_i \leq a_i$. This basically splits our edge set into two subsets, and we can recurse on both of them. Therefore, we would like to pick λ so as to split them as evenly as possible, which we can do by binary search on λ, each time computing a shortest path on the DAG. Note that there will always exist a balanced split, since any edge that is not part of a shortest path for any λ can be discarded. Let $r(m)$ be the amount of time the algorithm takes, given a DAG with m edges, plus n edges (one outgoing edge for each vertex). Note that the n edges define an arborescence A that for some choice of λ was the shortest path tree of the DAG. In order to compute the shortest path tree in such a DAG, we run a Breadth-First search using only the m edges, and each time we encounter a path that is shorter than the respective path in A, we update A by substituting an edge (u, v) with another edge (u', v). This takes $O(m)$ time. Therefore if we denote by $r(m)$ the time the algorithm takes when the number of non-arborescence edges is m, the recursion to (implicitly) compute the shortest path trees for all choices of λ will be

$$r(m) = 2r(m/2) + m \log(nM)$$

so $r(m) = O(m \log n \log(nM))$ and so the total time is $O(m \log n \log(nM))$.

Note that each leaf of this recursion exactly corresponds to an implicit shortest path tree, for a particular value of λ. In order to reconstruct the shortest path from 0 to some node u with exactly k edges, we traverse the recursion tree top-down, moving to the left child (smaller λ) if the number of edges in the current path from 0 to u is less than k, or to the right child if the number of edges is is more than k. This way, we can compute the shortest paths of k hops from 0 to each node, in time $O(m \log n \log(nM))$. □