Capacitated Dynamic Programming: Faster Knapsack and Graph Algorithms

Kyriakos Axiotis
MIT, Cambridge, MA, USA
kaxiotis@mit.edu

Christos Tzamos
University of Wisconsin-Madison, USA
tzamos@wisc.edu

──── **Abstract** ────

One of the most fundamental problems in 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 improved algorithm running in time $O(TD)$, where D is the number of distinct weights. Previously, faster runtimes for Knapsack were only possible when both weights and values are bounded by M and V respectively, running in time $O(nMV)$ [17]. In comparison, our algorithm implies a bound of $O(nM^2)$ without any dependence on V, or $O(nV^2)$ without any dependence on M. Additionally, for the unbounded Knapsack problem, we provide an algorithm running in time $O(M^2)$ or $O(V^2)$. Both our algorithms match recent conditional lower bounds shown for the Knapsack problem [10, 15].

We also initiate a systematic study of general *capacitated dynamic programming*, of which Knapsack is a 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

46th International Colloquium on Automata, Languages, and Programming (ICALP 2019).
Editors: Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi;
Article No. 19; pp. 19:1–19:13

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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 [6] has a runtime of $O(Tn)$.

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 [17] 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 [6] was also an algorithm by Pisinger [17] which runs in time $O(nM)$ until significant recent progress by Bringmann [7] and Koiliaris and Xu [14] 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 problem is much harder. Specifically, [10, 15] reduce the $(\max, +)$-convolution problem to Knapsack, proving that any truly subquadratic algorithm for Knapsack (i.e. $O((n + T)^{2-\epsilon})$) 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. [10, 15, 3]). However, an important open question remains here: *Can we get faster algorithms that circumvent this conditional lower bound?*

We answer this question affirmatively by providing an algorithm running in time $O(TD)$, where D is the number of *distinct* weights. Our algorithm is deterministic and computes the optimal Knapsack value for *all* capacities t from 1 to T. Since $D \leq n$, its runtime either matches (for $D = \Theta(n)$) or yields an improvement (for $D = o(n)$) over Bellman's algorithm [6], for all parameter regimes. It also directly implies runtimes of $O(TM)$[1], $O(nM^2)$[2], and $O(nV^2)$, and therefore also yields an improvement over the $O(nMV)$ algorithm of Pisinger [17].

---

[1] Concurrent and independent work by Bateni, Hajiaghayi, Seddighin, and Stein [4] also obtains an algorithm running in time $\widetilde{O}(TM)$, as well as an algorithm running in time $\widetilde{O}(TV)$. In comparison to ours, their $\widetilde{O}(TM)$ algorithm is randomized and computes the answer only for a single capacity T.

[2] Eisenbrand and Weismantel [11] develop fast algorithms for Integer Programming. Concurrently and independently, they also obtain an algorithm for Knapsack that runs in time $O(nM^2)$. They provide a structural property of Knapsack using the Steinitz Lemma that enables us to remove logarithmic factors in T from our results for Unbounded Knapsack (Theorem 13), as they reduce to the case $T = \Theta(M^2)$. Combined with Theorem 8, this also implies an $O(M^3)$ algorithm for Knapsack.

■ **Table 1** Summary of our deterministic pseudopolynomial time results on the Knapsack problem with the corresponding known conditional lower bounds based on $(\min, +)$-convolution.

Setting	Our Results	Conditional Lower bounds
Knapsack		
No bounds on weights or values	$O(TD)$ [**Theorem 8**]	$\Omega((TD)^{1-o(1)})$ [10, 15]
Weights bounded by M	$O(TM)$ [**Corollary 11**]	$\Omega((TM)^{1-o(1)})$ [10, 15]
Values bounded by V	$O(nV^2)$ [**Corollary 12**]	−
Unbounded Knapsack		
Weights bounded by M	$O(M^2)$ [**Corollary 14**]	$\Omega(M^{2-o(1)})$ [10, 15]
Values bounded by V	$O(V^2)$ [**Corollary 15**]	−

Our algorithm can be summarized as follows: First, it 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 it is not known whether computing general $(\max, +)$-convolutions in truly sub-quadratic time is possible, we exploit the inherent concavity of the specific family of sequences produced by our algorithm to perform this in linear time. We present our results in Section 3.1.

In addition to the general Knapsack problem studied above, we also consider the *Unbounded Knapsack* problem where there are infinite copies of every item. In Section 3.2, we present novel algorithms for Unbounded Knapsack with running times $O(M^2)$[3] and $O(V^2)$, where M is the maximum weight and V is the maximum value of any item. Our algorithm again utilizes $(\max, +)$-convolutions of short sequences to compute the answer and interestingly is only pseudo-polynomial with respect to the maximum weight M or the maximum value V and not the capacity T.

Our results are summarized in Table 1.

It follows from the results of [10, 15] that, under the $(\min, +)$-convolution hardness assumption, it is not possible to obtain faster runtimes for Knapsack under most of the parameterizations that we consider. This is because, even though the lower bound claimed in these results is $\Omega((n + T)^{2-o(1)})$, the hardness construction uses a Knapsack instance where T, M, and D are $\Theta(n)$.

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 [19, 15], as well as capacitated problems with weighted adjacency matrices that satisfy a specific condition, namely the *Monge property* [2, 18, 5]. In [5], it is shown that under this condition, the

---

[3] Jansen and Rohwedder [13] extend the results of [11] for Integer Programming and also concurrently and independently obtain an algorithm for Unbounded Knapsack running in time $O(M^2)$.

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 18, 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, at least one of the edges (a, v) and (v, c) exists.

To illustrate the power of our characterization, we show that a linear algorithm can be easily obtained for the problem of k-sparse Δ-separated subsequences [12] recovering recent results of [8, 16].

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 only one 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 Knapsack and Unbounded Knapsack:

▶ **Definition 1** (Knapsack). *Given N items with weights $w_1, \ldots, w_N \in [M]$ and values $v_1, \ldots, v_N \in [V]$, 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 number of distinct weights by D.*

▶ **Definition 2** (Unbounded Knapsack). *Given N items with weights $w_1, \ldots, w_N \in [M]$ and values $v_1, \ldots, v_N \in [V]$, 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 number of distinct weights by D.*

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

▶ **Definition 3** ($(\max, +)$-convolution). *Given two sequences $a_0, \ldots, a_n$ and $b_0, \ldots, b_m$, the $(\max, +)$-convolution $a \oplus b$ between a and b is a sequence $c_0, \ldots, c_{n+m}$ such that for any i*

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

This operation is commutative, so it is also true that

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

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

▶ **Definition 4** (Concave, k-step concave). *A sequence $b_0, \ldots, b_n$ is* concave *if for all $i \in [n-1]$ we have $b_i - b_{i-1} \geq b_{i+1} - b_i$. 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 5** (Monge matrices). *A matrix $A \in \mathbb{R}^{n \times m}$ is called* Monge *if for any $i \in [n-1]$ and $j \in [m-1]$*

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

▶ **Definition 6** (Monge weights). *We will say that a Directed Acyclic Graph has* Monge weights *if its weighted adjacency matrix is a Monge matrix.*

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

▶ **Definition 7** ($(\max, +)$-convolution hardness). *The $(\max, +)$-convolution hardness hypothesis states that any algorithm that computes the $(\max, +)$-convolution of two sequences of size n requires 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 $\max_{i+j+k=n} \{a_i + b_j + c_k\}$. 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 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 8, we present an algorithm for 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 13, we present algorithms for Unbounded Knapsack with runtimes $O(M^2)$ and $O(V^2)$, where M is the maximum weight and V the maximum value of some item.

3.1 Knapsack

Given N items with weights $w_1, \ldots, w_N \in [M]$ and values $v_1, \ldots, v_N \in [V]$, 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 number of distinct weights by D.

The following is the main theorem of this section.

▶ **Theorem 8.** *Algorithm 1 solves* Knapsack *in time $O(TD)$.*

Algorithm 1 Knapsack.

1: Given items with weights in $\{w_1^{\#}, \ldots, w_D^{\#}\}$
2: Partition items into sets $S_1, \ldots, S_D$, 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 10
8:     Truncate s after the T-th entry
9: Output s_T

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 the 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 9.** *Given an arbitrary sequence $a_0, \ldots, a_m$ and a concave sequence $b_0, \ldots, b_n$ we can compute the $(\max, +)$ convolution between a and b in time $O(m + n)$.*

Proof. Consider a (zero-indexed) $(n + 1) \times (m + 1)$ matrix A with $A_{ij} = a_j + b_{i-j}$ for $(i, j) \in \{0, \ldots, n\} \times \{0, \ldots, m\}$, where we suppose that elements of the sequences with out-of-bounds indices have value $-\infty$. 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 maxima of A. Now note that for any $(i, j) \in \{0, 1, \ldots, n-1\} \times \{0, 1, \ldots, m-1\}$, 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 an $(n+1) \times (m+1)$ Monge matrix A, one can compute all its row maxima in time $O(m + n)$, which implies the Lemma. ◀

▶ **Lemma 10.** *Given an arbitrary sequence $a_0, \ldots, a_m$ and a k-step concave sequence $b_0, \ldots, b_n$ 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 9). 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 $x^{(i)} := (a_i, a_{k+i}, a_{2k+i}, \ldots)$ denote the subsequence of a with indices whose remainder is i when divided by k, and $y := (b_0, b_k, b_{2k}, \ldots)$. Furthermore, define

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

Now, for any j we have

$$\max_{i=j-k+1}^{j} f_i = \max_{i=j-k+1}^{j} \max_{q=0}^{\infty}\{b_{qk} + a_{i-qk}\}$$

$$= \max_{i=j-k+1}^{j} \max_{q=0}^{\infty}\{b_{qk+j-i} + a_{i-qk}\}$$

$$= \max_{z=0}^{\infty}\{b_z + a_{j-z}\}$$

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 maxima 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,

$$f_{qk+p} = \max_{z=0}^{\infty}\{b_{zk} + a_{qk+p-zk}\}$$

$$= \max_{z=0}^{\infty}\{y_z + x^{(p)}_{q-z}\}$$

$$= (x^{(p)} \oplus y)_q$$

Furthermore, y is a concave sequence and by Lemma 9 we can compute such a convolution in time $O((m+n)/k)$. Doing this for all $p \in \{0, \ldots, k-1\}$, 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 maxima 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 $f_{[i,\ldots,i+k-1]}$, containing all the potentially useful elements. The first element of this subsequence is the maximum 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 $\leq f_{i+k}$, it will never be the maximum element in any segment. Therefore we can remove it and repeat until the last element has value greater 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 decreasing, and the first element will be the maximum of the respective segment. The total runtime is linear if implemented with a standard queue. ◄

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

Proof of Theorem 8. Consider any knapsack instance where D is the number of distinct item weights $w_1^\#, \ldots, w_D^\#$. 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 $b_0 = 0, b_{w_i} = v_1^{(i)}, b_{2w_i} = v_1^{(i)} + v_2^{(i)}, \ldots$, 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 10 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. ◄

▶ **Corollary 11.** *Knapsack can be solved in $O(TM)$ time.*

▶ **Corollary 12.** *Knapsack can be solved in time $O(nM^2)$ or $O(nV^2)$.*

Proof. The first bound directly follows by Corollary 11 and the fact that $T \leq nM$. For the second bound, note that by swapping the role of the weights and the values in Algorithm 1, replacing all $(\max, +)$-convolutions by $(\min, +)$-convolutions, and setting the knapsack capacity to nV as opposed to T, this algorithm runs in time $O(nV^2)$ and outputs for every possible value, the minimum weight of items that can achieve this value. The answer can then be recovered by finding the minimum value that gives a corresponding weight of at most T. ◀

3.2   Unbounded Knapsack

Given N items with weights $w_1, \ldots, w_N$ and values $v_1, \ldots, v_N$, 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 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** $z = 1, \ldots, \lceil \log M \rceil$ **do**
3:     $v^{(z)} \leftarrow v^{(z-1)} \oplus v^{(z-1)}$
4: $a_{[0,M]} \leftarrow v^{(\lceil \log M \rceil)}$
5: **for** $i = \lceil \log \frac{T}{M} \rceil, \ldots, 1$ **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 13.** *Algorithm 2 solves* Unbounded knapsack *in time $O(M^2 \log T)$.*

Overview. As in the algorithm for 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 13. 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

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

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

First, we compute the values $a_1, \ldots, a_M$ in $O(M^2 \log M)$ time as follows: We start with the sequence $v^{(0)}$, where $v_x^{(0)}$ 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 $a_{0,\ldots,M} \equiv v_{0,\ldots,M}^{(\lceil \log M \rceil)}$, which as we argued can be computed in time $O(M^2 \log M)$.

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

Recent work of [11] shows, using the *Steinitz lemma*, that an optimal Knapsack solution for a capacity in $[T - M, T]$ can be turned into an optimal solution for capacity T by inserting or removing at most M elements, where M is a bound on weight of the items. In the case of Unbounded Knapsack, a solution that only uses the best item until it exceeds capacity $T - M^2$ can always be extended into an optimal solution with capacity T. Therefore the capacity can be assumed to be $O(M^2)$. Combining this with the $\frac{M^2}{2^{\Omega\left(\sqrt{\log M}\right)}}$-time randomized algorithm of [20] (or the deterministic algorithm of [9]) for $(\max, +)$-convolution implies an algorithm that runs in time $\frac{M^2}{2^{\Omega\left(\sqrt{\log M}\right)}} O(\log M) = \frac{M^2}{2^{\Omega\left(\sqrt{\log M}\right)}}$.

▶ **Corollary 14.** *Unbounded Knapsack can be solved in time $O(M^2)$.*

A similar argument can be used to get a more efficient algorithm when we have a bound on the values of the items. In particular, using the item j with the highest value-to-weight ratio v_j/w_j, $k = \lfloor \frac{T}{w_j} \rfloor + 1$ times, until we exceed the capacity we get both a lower bound of $(k-1)v_j$ and a upper bound of kv_j on the value of the optimal solution. In addition, again by the Steinitz Lemma, there exists an optimal solution that uses item j at least $k - V$ times. This allows us to start from value $(k - V)v_j$ and compute the minimum weight required to achieve values in $[(k-1)v_j, kv_j]$. This gives an algorithm that runs in $O(V^2 \log V)$ using the naive algorithm for $(\min, +)$-convolutions, and $O\frac{V^2}{2^{\Omega\left(\sqrt{\log V}\right)}}$, again using the improved algorithms in [20] or [9].

▶ **Corollary 15.** *Unbounded Knapsack can be solved in time $O(V^2)$.*

4 k-link path in Node-weighted graphs

We now move on to study more general capacitated dynamic programming settings, described by computing the maximum reward k-link path in a directed acyclic graph. This setting can capture a lot of natural capacitated problems, either directly or indirectly, such as the Knapsack problem, k-sparse Δ-separated sequences, max-weight increasing subsequence of length k, and so on. Therefore a better understanding of these special cases might lead to improved algorithms for other capacitated problems.

More specifically, we study the problem of finding maximum-reward paths in node-weighted transitive DAGs. In Lemma 16, 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 16** ($(\max, +)$-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.*

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 17** (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 Lemma 18 we give a complete graph-theoretic characterization of the graphs that have this concavity property and therefore can be solved efficiently.

▶ **Lemma 18** (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 $val(u_1) = val(u_2) = val(u_3) = 1$, $val(v) = 1 + \epsilon$, and $-\infty$ for all other vertices. Then, $f(1) = 1 + \epsilon$, $f(3) = 3$, 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

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

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 [12] in near-linear time using the main result of this section, thus recovering recent results of [8, 16].

▶ **Lemma 19** (Max-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 $\sum\limits_{j \in [k]} a_{i_j}$ 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 Lemma 18. 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 17 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 $O(n \log(n \max_i |a_i|))$.   ◀

As another example of a problem that can be modeled as a capacitated maximum-reward path problem 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, as witnessed in the following lemma.

▶ **Lemma 20** (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 16. 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

$$x_0, x'_0, x_1, x'_1, \ldots, x_k, x'_k, y_0, y'_0, \ldots, y_k, y'_k, z_0, z'_0, \ldots, z_k, z'_k$$

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

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 $O((nk)^{1-\epsilon})$ algorithm for the Max-Weight Increasing Subsequence of length k problem implies a truly subquadratic algorithm for the $(\max,+)$-convolution problem. ◀

5   *k*-link path in 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 [5], one can compute a single such path in $\widetilde{O}(n)$ time.

▶ **Lemma 21** (From [5]). *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 22 we show that one needs $\Omega(n^{3/2})$ time to compute a column of A^k in general.

▶ **Lemma 22.** *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 23 we present an algorithm that can compute any row or column of A^k in $\widetilde{O}(nnz(A)) = \widetilde{O}(m)$ time.

▶ **Lemma 23.** *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 $O(m \log n \log(nM))$.*

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-weight$k$-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   MohammadHossein Bateni, MohammadTaghi Hajiaghayi, Saeed Seddighin, and Cliff Stein. Fast algorithms for knapsack via convolution and prediction. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 1269–1282, 2018.

5   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.

6   Richard Bellman. Dynamic Programming (DP). *Princeton University Press*, 1957.

7   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.

8   Henning Bruhn and Oliver Schaudt. Fast Algorithms for Delta-Separated Sparsity Projection. *arXiv preprint*, 2017. `arXiv:1712.06706`.

9   Timothy M Chan and Ryan Williams. Deterministic apsp, orthogonal vectors, and more: Quickly derandomizing razborov-smolensky. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 1246–1255. Society for Industrial and Applied Mathematics, 2016.

10   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.

11   Friedrich Eisenbrand and Robert Weismantel. Proximity results and faster algorithms for integer programming using the steinitz lemma. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 808–816. Society for Industrial and Applied Mathematics, 2018.

12   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.

13   Klaus Jansen and Lars Rohwedder. On Integer Programming and Convolution. *arXiv preprint*, 2018. `arXiv:1803.04744`.

14   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.

15   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.

16   Aleksander Mądry, Slobodan Mitrović, and Ludwig Schmidt. A Fast Algorithm for Separated Sparsity via Perturbed Lagrangians. In *International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain*, pages 20–28, 2018.

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

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

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

20   Ryan Williams. Faster all-pairs shortest paths via circuit complexity. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 664–673. ACM, 2014.