Dynamic Maintenance of Monotone Dynamic Programs and Applications

Monika Henzinger
Institute of Science and Technology Austria, Klosterneuburg, Austria

Stefan Neumann
KTH Royal Institute of Technology, Stockholm, Sweden

Harald Räcke
TU München, Germany

Stefan Schmid
TU Berlin, Germany
Fraunhofer SIT, Darmstadt, Germany

Abstract

Dynamic programming (DP) is one of the fundamental paradigms in algorithm design. However, many DP algorithms have to fill in large DP tables, represented by two-dimensional arrays, which causes at least quadratic running times and space usages. This has led to the development of improved algorithms for special cases when the DPs satisfy additional properties like, e.g., the Monge property or total monotonicity.

In this paper, we consider a new condition which assumes (among some other technical assumptions) that the *rows* of the DP table are *monotone*. Under this assumption, we introduce a novel data structure for computing $(1+\epsilon)$-approximate DP solutions in *near-linear time and space* in the static setting, and with *polylogarithmic update times* when the DP entries change dynamically. To the best of our knowledge, our new condition is incomparable to previous conditions and is the first which allows to derive *dynamic* algorithms based on existing DPs. Instead of using two-dimensional arrays to store the DP tables, we store the rows of the DP tables using monotone piecewise constant functions. This allows us to store length-n DP table rows with entries in $[0, W]$ using only $\text{polylog}(n, W)$ bits, and to perform operations, such as $(\min, +)$-convolution or rounding, on these functions in polylogarithmic time.

We further present several applications of our data structure. For bicriteria versions of *k-balanced graph partitioning* and *simultaneous source location*, we obtain the first dynamic algorithms with subpolynomial update times, as well as the first static algorithms using only near-linear time and space. Additionally, we obtain the currently fastest algorithm for *fully dynamic knapsack*.

2012 ACM Subject Classification Theory of computation → Dynamic programming; Theory of computation → Dynamic graph algorithms; Theory of computation → Packing and covering problems

Keywords and phrases Dynamic programming, dynamic algorithms, data structures

Digital Object Identifier 10.4230/LIPIcs.STACS.2023.36

Related Version *Full Version*: `https://arxiv.org/abs/2301.01744`

Funding *Monika Henzinger*: This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant agreement No. 101019564 "The Design of Modern Fully Dynamic Data Structures (MoDynStruct)" and from the Austrian Science Fund (FWF) project "Fast Algorithms for a Reactive Network Layer (ReactNet)", P 33775-N, with additional funding from the *netidee SCIENCE Stiftung*, 2020–2024.
Stefan Neumann: This research is supported by the the ERC Advanced Grant REBOUND (834862) and the EC H2020 RIA project SoBigData++ (871042).
Stefan Schmid: Research supported by Austrian Science Fund (FWF) project I 5025-N (DELTA), 2020-2024.

1 Introduction

Dynamic programming (DP) is one of the fundamental paradigms in algorithm design. In the DP paradigm, a complex problem is broken up into simpler subproblems and then the original problem is solved by combining the solutions for the subproblems. One of the drawbacks of DP algorithms is that in practice they are often slow and memory-intensive: for inputs of size n their running time is typically $\Omega(n^2)$, and when the DP table is stored using a two-dimensional array they also need space $\Omega(n^2)$.

This motivated researchers to develop more efficient DP algorithms with near-linear time and space. Indeed, such improvements are possible under a wide range of conditions on the DP tables [1, 4, 6, 8, 12, 16, 20–22, 24], such as the Monge property, total monotonicity, certain convexity and concavity properties, or the Knuth–Yao quadrangle-inequality; we discuss these properties in more detail in the full version [17]. When these properties hold, typically one does not have to compute the entire DP table but instead only has to compute $O(n)$ DP entries which reveal the optimal solution.

However, we are not aware of any property for DPs that yields efficient *dynamic* algorithms, i.e., algorithms that provide efficient update operations when the input changes. One might find this somewhat surprising because, from a conceptual point of view, many dynamic algorithms hierarchically partition the input and maintain solutions for subproblems; this is quite similar to how many DP schemes are derived. Indeed, this conceptual similarity is exploited by many "hand-crafted" algorithms (e.g., [9, 18]) which start with a DP scheme and then show how to maintain it dynamically under input changes. However, such algorithms are often quite involved and their analysis often requires sophisticated charging schemes.

Hence, it is natural to ask whether there exists a *general criterion* which, if satisfied, guarantees that a given DP can be updated efficiently under input changes.

Our Contributions. The main contribution of our paper is the introduction of a general criterion which allows to approximate all entries of a DP table up to a factor of $1 + \epsilon$. We show that if our criterion is satisfied by a DP (with suitable parameters) then:

- In the dynamic setting, we can maintain a $(1 + \epsilon)$-approximation of the entire DP table using polylogarithmic update time (see Theorem 10).
- In the static setting, we can compute a $(1+\epsilon)$-approximation of the DP table in near-linear time and space (see Theorem 9).

Our criterion essentially asserts that the *rows* of the DP tables should be *monotone* and that the dependency graph of the DP should be a DAG, where the sets of reachable nodes are small, among some other technical conditions (see Definition 8 for the formal definition). Our criterion is incomparable to the Monge property, total monotonicity or other criteria from the literature (see the full version [17] for a more detailed discussion).

To obtain our results, we introduce a novel data structure for maintaining DPs which satisfy our criterion. Our data structure is based on the idea of storing the DP rows using *monotone piecewise constant* functions. The monotonicity of the DP rows will allow us to ensure that our functions only contain very few pieces. Then we show that we can perform operations on such functions very efficiently, with the running times only depending on the number of pieces. This is crucial because it allows us to compute an *entire* $(1+\delta)$-approximate DP row in time just $\text{polylog}(W)$, even when the DP has $\Omega(n)$ columns, assuming that the DP entries are from $[0, W]$. Note that if $W \leq \text{poly}(n)$ then this decreases the running time for computing an entire row from $\Omega(n)$ to just $\text{polylog}(n)$. Additionally, this also allows us to store each row *using only* $\text{polylog}(W)$ *space* rather than storing it in an array of size $\Omega(n)$. We present our criterion and the details of our data structure in Section 2.

As applications of our data structure, we obtain new static and dynamic algorithms for various problems. We present new algorithms for *k-balanced partitioning*, *simultaneous source location* and for *fully dynamic knapsack*. Next, we describe these results in detail; we discuss more related work in the full version [17].

Our Results for Fully Dynamic 0-1 Knapsack. First, we provide a novel algorithm for fully dynamic 0-1 knapsack. In this problem, the input consists of a knapsack size $B \in \mathbb{R}_+$ and a set of n items, where each item $i \in [n]$ has a weight $w_i \in \mathbb{R}_+$ and a price $p_i \in [1, \infty)$. The goal is to find a set of items I that maximizes $\sum_{i \in I} p_i$ while satisfying the constraint $\sum_{i \in I} w_i \leq B$. In the dynamic version of the problem, items are inserted and deleted. More concretely, we consider the following update operations: *insert*(p_i, w_i), in which the price and weight of item i are set to $p_i \in [1, \infty)$ and $w_i \in \mathbb{R}_+$, respectively, and *delete*(i), where item i is removed from the set of items.

Our main result is a dynamic $(1+\epsilon)$-approximation algorithm with worst-case update time $\epsilon^{-2} \cdot \log^2(nW) \cdot \text{polylog}(1/\epsilon, \log(nW))$, where $W = \sum_i p_i$. Our algorithm improves upon a recent result by Eberle, Megow, Nölke, Simon and Wiese [11] that also maintained a $(1+\epsilon)$-approximate solution with update time $O(\epsilon^{-9} \log^4(nW))$.

▶ **Theorem 1.** *Let $\epsilon > 0$. There exists an algorithm for fully dynamic knapsack that maintains a $(1+\epsilon)$-approximate solution with worst-case update time $\frac{1}{\epsilon^2} \log^2(nW) \,\text{polylog}\left(\frac{1}{\epsilon} \log(nW)\right)$.*

We will also show that we can return the maintained solution I in time $O(|I|)$ and that we can answer queries whether a given item $i \in [n]$ is contained in I in time $O(1)$. This matches the query times of [11].

To obtain this result, we first derive a slightly slower algorithm as a simple application of our data structure for maintaining DPs with monotone rows. Then we use this algorithm together with additional ideas to obtain the theorem (see Section 3).

Since our dynamic algorithm is based on a DP, it is possible that the solution changes significantly after each update. However, in the full version [17] we prove a lower bound, showing that every dynamic $(1+\epsilon)$-approximation algorithm for knapsack must either make a lot of changes to the solution after each update or store many (potentially substantially different) solutions between which it can switch after each update. This implies that maintaining a single explicit solution with polylogarithmic update times is not possible and the property of our algorithm cannot be avoided.

Our Results for k-Balanced Partitioning. Our most technically challenging result is for *k-balanced graph partitioning*. In this problem, the input consists of an integer k and an undirected weighted graph $G = (V, E, \text{cap})$ with n vertices, where $\text{cap} : E \to W_\infty$ is a weight function on the edges with weights in $W_\infty := [1, W] \cup \{0, \infty\}$. The goal is to find a partition $V_1, \ldots, V_k$ of the vertices such that $|V_i| \leq \lceil |V| / k \rceil$ for all i and the weight of the edges which are cut by the partition is minimized. More formally, we want to minimize $\mathit{cut}(V_1, \ldots, V_k) := \sum_{i=1}^{k} \sum_{\{u,v\} \in E \cap (V_i \times (V \setminus V_i))} \text{cap}(u, v)$.

We note that this problem is highly relevant in theory [3,13–15] and in practice [5,10,19,23], where algorithms for balanced graph partitioning are often used as a preprocessing step for large scale data analytics. Obtaining practical improvements for this problem is of considerable interest in applied communities [5] and, for instance, the popular METIS heuristic [19] has 1,400+ citations.

Since the above problem is NP-hard to approximate within a factor of $n^{1-\epsilon}$ for any $\epsilon > 0$ even on trees [15], we consider bicriteria approximation algorithms. Given an undirected weighted graph $G = (V, E, \text{cap})$, a partition $V_1, \ldots, V_k$ of V is a *bicriteria* (α, β)*-approximate*

solution if $|V_i| \leq \beta\lceil n/k \rceil$ for all i and $cut(V_1, \ldots, V_k) \leq \alpha \cdot cut(\text{OPT})$, where $\text{OPT} = (V_1^*, \ldots, V_k^*)$ is the optimal solution with $|V_i^*| \leq \lceil n/k \rceil$ for all i. We note that the previously mentioned hardness result implies that any algorithm that computes a bicriteria $(\alpha, 1+\epsilon)$-approximation for any $\alpha \geq 1$ and whose running time depends only polynomially on n, must have a running time depending super-polynomially on $1/\epsilon$, unless $\mathsf{P} = \mathsf{NP}$.[1]

Our main result for the static setting is presented in the following theorem. It gives the first algorithm with polylogarithmic approximation ratio for this problem *with near-linear running time*. More concretely, we compute a bicriteria $(O(\log^4 n), 1+\epsilon)$-approximation in near-linear time for constant k. For comparison, the best approximation ratio achieved by a polynomial-time algorithm [15] is a bicriteria $(O(\log^{1.5} n \log\log n), 1+\epsilon)$-approximation with running time $\Omega(n^4)$.

▶ **Theorem 2.** *Let $\epsilon > 0$ and $k \in \mathbb{N}$. Let $G = (V, E, \text{cap})$ be an undirected weighted graph with n vertices and m edges and edge weights in W_∞. Then for the k-balanced partition problem we can compute:*

- *An $(O(\log^4 n), 1+\epsilon)$-approximation in time $(k/\epsilon)^{O(\log(1/\epsilon)/\epsilon)} \cdot O'(m \cdot \log^2(W)) + (k/\epsilon)^{O(1/\epsilon^2)}$.*[2]
- *A $(1+\epsilon, 1+\epsilon)$-approximation in time $(k/\epsilon)^{O(\log(1/\epsilon)/\epsilon)} \cdot O'(n \cdot h^2 \cdot \log^2(W)) + (k/\epsilon)^{O(1/\epsilon^2)}$ if G is a tree of height h.*
- *A $(1, 1+\epsilon)$-approximation in time $(k/\epsilon)^{O(\log(1/\epsilon)/\epsilon)} \cdot O'(n^4 \cdot \log^2(W)) + (k/\epsilon)^{O(1/\epsilon^2)}$ if G is a tree.*

Furthermore, we extend our results to the dynamic setting in which the graph G is undergoing edge insertions and deletions. In the following theorem, we present *the first dynamic algorithm with subpolynomial update time* for this problem. We again consider bicriteria approximation algorithms with update and query times depending super-polynomially on $1/\epsilon$; this cannot be avoided since if we computed $(\alpha, 1)$-approximations for any $\alpha \geq 1$ or if we had a polynomial dependency on $1/\epsilon$, then the hardness result from above implies that our update and query times must be super-polynomial in n (unless $\mathsf{P} = \mathsf{NP}$).

▶ **Theorem 3.** *Let $\epsilon > 0$ and $k \in \mathbb{N}$. Let $G = (V, E, \text{cap})$ be an undirected weighted graph with n vertices that is undergoing edge insertions and deletions. Then for the k-balanced partition problem we can maintain:*

- *An $(n^{o(1)}, 1+\epsilon)$-approximate solution with amortized update time $(k/\epsilon)^{O(\log(1/\epsilon)/\epsilon)} \cdot n^{o(1)} \cdot O'(\log^2(W))$ and query time $(k/\epsilon)^{O(1/\epsilon^2)}$ if G is unweighted.*
- *A $(1+\epsilon, 1+\epsilon)$-approximate solution with worst-case update time $(k/\epsilon)^{O(\log(1/\epsilon)/\epsilon)} \cdot O'(h^3 \cdot \log^2(W))$ and query time $(k/\epsilon)^{O(1/\epsilon^2)}$ if G is a tree of height h.*

Our approach is inspired by the DP of Feldmann and Foschini [15]. However, the DP rows in the algorithm of [15] are not monotone and, hence, their DP cannot directly be sped up by our approach. Therefore, we first simplify and generalize the exact DP of Feldmann and Foschini to make it monotone. The DP we obtain eventually is still slightly too complex to fit into our black-box framework, but we show that the ideas from our framework can still be used to obtain the result.

Again, it is possible that the solution maintained by our algorithm changes substantially after each update. Similar to above we show in the full version [17] that this cannot be avoided when considering subpolynomial update times.

---

[1] If we had an algorithm that computes a bicriteria $(\alpha, 1+\epsilon)$-approximation in time $\text{poly}(n, 1/\epsilon)$ then we could set $\epsilon = 1/(2n)$ which implies that all partitions have size $\lceil n/k \rceil$. Thus we can compute a bicriteria $(\alpha, 1)$-approximate solution in time $\text{poly}(n)$ which contradicts the hardness result, unless $\mathsf{P} = \mathsf{NP}$.

[2] We use the notation $O'(\cdot)$ to suppress factors in $\text{poly}(\log n, k, \log(1/\epsilon), \log\log(W))$.

Our Results for Simultaneous Source Location. Next, we provide efficient algorithms for the simultaneous source location problem by Andreev, Garrod, Golovin, Maggs and Meyerson [2]. In this problem, the input consists of an undirected graph $G = (V, E, \text{cap}, d)$ with a capacity function $\text{cap}\colon E \to W_\infty$ on the edges and a demand function $d\colon V \to W_\infty$ on the vertices. The goal is to select a minimum set $S \subseteq V$ of *sources* that can simultaneously supply all vertex demands. More concretely, a set of sources S is *feasible* if there exists a flow from the vertices in S that supplies demand $d(v)$ to all vertices $v \in V$ and that does not violate the capacity constraints on the edges. The objective is to find a feasible set of sources of minimum size.

We will again consider bicriteria approximation algorithms. Let S^* be the optimal solution for the simultaneous source location problem. Then we say that S is a *bicriteria* (α, β)*-approximate solution* if $|S| \leq \alpha\,|S^*|$ and if S is a feasible set of sources when all edge capacities are increased by a factor β.

The following theorem summarizes our main results. It presents *the first near-linear time algorithm* for simultaneous source location that computes a $(1+\epsilon)$-approximate solution while only exceeding the edge capacities by a $O(\log^4 n)$ factor. In comparison, the best algorithm with arbitrary polynomial processing time computes a bicriteria $(1, O(\log^2 n \log\log n))$-approximate solution in time $\Omega(n^3)$ [2].

▶ **Theorem 4.** *Let $\epsilon > 0$. Let $G = (V, E, \text{cap}, d)$ be an undirected weighted graph with n vertices and m edges. Then for the simultaneous source location problem we can compute:*

- *A $(1+\epsilon, O(\log^4(n)))$-approximation in time*[3] $\tilde{O}(\frac{1}{\epsilon^2}m)$.
- *A $(1+\epsilon, 1)$-approximation in time $\tilde{O}(\frac{1}{\epsilon^2}h^2 \cdot n)$ if G is a tree of height h.*

Next, we turn to dynamic versions of the problem. We consider the following update operations: *SetDemand*(v, d): updates the demand of vertex v to $d(v) = d$, *SetCapacity*$((u, v), c)$: updates the capacity of the edge (u, v) to $\text{cap}(u, v) = c$, *Remove*(u, v): removes the edge (u, v), *Insert*$((u, v), c)$: inserts the edge (u, v) with capacity $\text{cap}(u, v) = c$.

We obtain *the first dynamic algorithms with subpolynomial update times* for this problem, which exceed the edge capacities only by a small subpolynomial factor.

▶ **Theorem 5.** *Let $\epsilon > 0$. Let $G = (V, E, \text{cap}, d)$ be a graph with n vertices and m edges that is undergoing the update operations given above. Then for the simultaneous source location problem we can maintain:*

- *A $(1+\epsilon, n^{o(1)})$-approximation with amortized update time $n^{o(1)}/\epsilon^2$ and preprocessing time $O(n^2/\epsilon^2)$ if all edge capacities are 1.*
- *A $(1+\epsilon, O(\log^4(n)))$-approximation with worst-case update time $\tilde{O}(1/\epsilon^2)$ and preprocessing time $\tilde{O}(m)$ if we only allow the update operation SetDemand(v, d).*
- *A $(1+\epsilon, O(\log^2(n)\log\log(n)))$-approximation with worst-case update time $\tilde{O}(1/\epsilon^2)$ and preprocessing time $\text{poly}(n)$ if we only allow the update operation SetDemand(v, d).*
- *A $(1+\epsilon, 1)$-approximate solution with worst-case update time $\tilde{O}(h^3/\epsilon^2)$ and preprocessing time $O(n^2/\epsilon^2)$ if G is a tree of height h.*

To obtain these results, we use a similar DP approach as the one used by Andreev et al. [2]. Interestingly, the DP function that we use essentially computes *the inverse function* of the one used by Andreev et al. After making these changes, the theorems become straightforward applications of our data structure for maintaining DPs with monotone rows.

[3] We write $\tilde{O}(f(n, \epsilon, W))$ to denote running times of the form $f(n, \epsilon, W) \cdot \text{polylog}(n, \epsilon, \log W)$.

Organization of Our Paper. In Section 2 we provide the details of our condition for DPs with monotone rows. In Section 3 we present our results for 0-1 Knapsack which nicely illustrate the applicability of our black-box framework from Section 2. All other results, including full proofs and a technical overview of our more involved results for k-Balanced Graph Partitioning and for Simultaneous Source Location are presented in the full version [17].

Open Problems and Future Work. In the future, it will be interesting to use our framework to obtain more dynamic algorithms based on existing DPs. We believe that this is interesting both in theory and in practice. Furthermore, it is intriguing to ask whether our criterion from Definition 8 can be generalized. Indeed, our approach was built around approximating monotone functions using piecewise constant functions, which can be viewed as piecewiese degree-0 polynomials. An interesting question is whether we can obtain a more general criterion if we approximate DP rows using pieces of higher-degree polynomials, such as splines. Results in this direction might be possible; for example, in the full version [17] we give a side result for the case when the functions contain a small number of non-monotonicities and derive a dynamic algorithm for the ℓ_∞-necklace problem.

2 Maintaining Monotone Dynamic Programming Tables

In this section, we introduce our notion of DP tables with *monotone rows* and the additional technical assumptions that we are making. Then we present our data structure for efficiently maintaining DP tables that satisfy our assumptions. In our data structure, we will store the rows of the DP using piecewise constant functions, which we will introduce first.

List Representation of Piecewise Constant Functions. Let $t \in \mathbb{R}$, $W \in [1, \infty)$ and set $W_\infty := \{0\} \cup [1, W] \cup \{+\infty\}$. A function $f\colon [0, t] \to W_\infty$ is *piecewise constant with p pieces* if there exist real numbers $0 = x_0 < x_1 < x_2 < \cdots < x_p = t$ and numbers $y_1, \ldots, y_p \in W_\infty$ such that on each interval $[x_{i-1}, x_i)$, f is constant and has value y_i. More formally, for all $i \in \{1, \ldots, p\}$ we have $f(x) = y_i$ for all real numbers $x \in [x_{i-1}, x_i)$ and $f(x_p) = y_p$. Note that we need the condition $f(x_p) = y_p$ such that f is defined on the whole domain.

In the *list representation* of a piecewise constant function f, we use a doubly linked list to store the pairs $(x_1, y_1), \ldots, (x_p, y_p)$. We also store the pairs (x_i, y_i) in a binary search tree that is sorted by the x_i-values, which allows us to compute a function value $f(x)$ in time $O(\log p)$ for all $x \in [0, t]$. In the following, we assume that all piecewise constant functions we consider are stored in the list representation with an additional binary search tree.

One of the main observations we use is that many operations on piecewise constant functions are efficient if there are only few pieces. The following lemma shows that several operations can be computed in time almost linear in the number of pieces of the function, rather than in time depending on the size of the domain of f.[4] For $\delta > 0$ and $y \in W_\infty$, we write $\lceil y \rceil_{1+\delta}$ to denote the smallest power of $1+\delta$ that is at least y, i.e., $\lceil y \rceil_{1+\delta} = \min\{(1+\delta)^i : (1+\delta)^i \geq y,\ i \in \mathbb{N}\}$; we follow the convention that $\lceil 0 \rceil_{1+\delta} = 0$ and $\lceil \infty \rceil_{1+\delta} = \infty$.

▶ **Lemma 6.** *Let $t \in \mathbb{R}$ and $c \in \mathbb{R}_+$. Let $g, h : [0, t] \to W_\infty$ be monotone and piecewise constant functions with p_g and p_h pieces, resp. Then we can compute the following functions:*

- *$f_{\min}(x) := \min\{g(x), h(x)\}$ with at most $p_g + p_h$ pieces in time $O((p_g + p_h)\log(p_g + p_h))$;*

[4] We note that computing the operations themselves can be done in linear time. However, since we also store the pairs (x_i, y_i) of the list representations in a binary search tree, the running times in the lemma include an additional logarithmic factor.

- $f_{\text{shift}}(x) := g(x-c)$ *for* $x \geq c$, $f_{\text{shift}}(x) = g(0)$ *for* $x < c$ *with at most* p_g *pieces in time* $O(p_g \log(p_g))$;
- $f_{\text{add}}(x) := g(x) + h(x)$, *with at most* $p_g + p_h$ *pieces in time* $O((p_g + p_h)\log(p_g + p_h))$;
- $f_{\text{round}}(x) := \lceil g(x) \rceil_{1+\delta}$ *for* $\delta > 0$ *with at most* $2 + \lceil \log_{1+\delta}(W) \rceil$ *pieces in time* $O(p_g \log(p_g))$.

Note that if we set $\tilde{f} = \lceil f \rceil_{1+\delta}$ then $\tilde{f}$ is a $(1+\delta)$-approximation of f in the following sense. For $\alpha > 1$, we say that a function $\tilde{f} \colon [0,t] \to W_\infty$ *α-approximates* a function $f \colon [0,t] \to W_\infty$ if for all $x \in [0,t]$,

$$f(x) \leq \tilde{f}(x) \leq \alpha \cdot f(x). \tag{1}$$

Furthermore, if f is monotone then the rounded function $\tilde{f}$ contains at most $O(\log_{1+\delta}(W))$ pieces. This will be crucial later because this ensures that, if we perform a single rounding operation for each row of our DP table, the resulting function will have few pieces and operations on the function can be performed efficiently.

Next, consider functions $f_1, f_2 : [0,t] \to W_\infty$. A function $f : [0,t] \to W_\infty$ is the $(\min, +)$-*convolution* $f_1 \oplus f_2$ if for all $x \in [0,t]$, $f(x) = (f_1 \oplus f_2)(x) := \min_{\bar{x} \in [0,x]} f_1(\bar{x}) + f_2(x - \bar{x})$. Such convolutions are highly useful for the computation of many DPs. The following lemma shows that we can efficiently compute the convolution of piecewise constant functions.

▶ **Lemma 7.** *Let* $f_1, f_2 : [0,t] \to W_\infty$ *be piecewise constant functions with at most* p *pieces and assume that one of them is monotonically decreasing. Then we can compute the function* $f : [0,t] \to W_\infty$ *with* $f = f_1 \oplus f_2$ *in time* $O(p^2 \log p)$ *and* f *is a piecewise constant function with* $O(p^2)$ *pieces. Furthermore, after computing* f, *for any* $x \in [0,t]$ *we can return a value* $\bar{x}^* \in [0,t]$ *such that* $f(x) = f_1(\bar{x}^*) + f_2(x - \bar{x}^*)$ *in time* $O(\log p)$.

Now observe that Lemma 7 has a drawback for our approach: The number of pieces (i.e., the complexity of the functions) grows quadratically with every application. An important property which can be used to mitigate this issue is that the result of the convolution is still a monotone function, as we show in the full version [17]. Later, to keep the number of pieces in our functions small, after each convolution that we perform via Lemma 7 (and that might grow the number of pieces quadratically), we perform a rounding operation $\lceil \cdot \rceil_{1+\delta}$ (see Lemma 6). This loses a factor $1+\delta$ in approximation but guarantees that the resulting function has $O(\log_{1+\delta}(W))$ pieces. This will be crucial to ensure that our functions have only few pieces.

Maintaining DPs With Monotone Rows. Next, we introduce our DP scheme formally. We consider DP tables with a finite set of rows $\mathcal{I}$ and a set of columns $\mathcal{J}$, with entries taking values in W_∞. We will consider DP tables as functions $\mathsf{DP} \colon \mathcal{I} \times \mathcal{J} \to W_\infty$.[5] Further, we will associate the i'th row of the DP with a function $\mathsf{DP}(i, \cdot) \colon \mathcal{J} \to W_\infty$, and we store each such function $\mathsf{DP}(i, \cdot)$ using piecewise constant functions from above.

Next, we introduce the dependency graph for the *rows* of our DP. More concretely, the *dependency graph* $D = (\mathcal{I}, E_D)$ is a directed graph that has the rows $\mathcal{I}$ as vertices and a directed edge (i', i) between two rows if for some columns $j, j' \in \mathcal{J}$ the entry $\mathsf{DP}(i', j')$ is required to compute $\mathsf{DP}(i, j)$. We write $\text{In}(i) = \{i' \in \mathcal{I} \colon (i', i) \in E_D\}$ to denote the set of rows i' that are required to compute row i. For the rest of the paper we will assume that the dependency graph is a DAG, which is the case for all applications that we study. We will also write $\text{Reach}(i)$ to denote the set of vertices that are reachable from row i in D.

[5] Even though our definition may suggest that we only consider two-dimensional DP tables, we do not require an order on $\mathcal{I}$ and we allow $\mathcal{I}$ to be any finite set. For example, in our DP for balanced graph partitioning we will set $\mathcal{I}$ to 3-tuples corresponding to the parameters of a four-dimensional DP.

Since we assume that the dependency graph is a DAG, we can compute the i'th DP row as soon as we have computed the solutions for the DP rows in $\mathrm{In}(i)$. We assume that this is done via a *procedure* $\mathcal{P}_i$ that takes as input the DP rows $\mathsf{DP}(i', \cdot)$ for all $i' \in \mathrm{In}(i)$ and returns the row $\mathsf{DP}(i, \cdot) = \mathcal{P}_i(\{\mathsf{DP}(i', \cdot) \colon i' \in \mathrm{In}(i)\})$.

Next, we come to our condition which encodes when our scheme applies. In the definition and for the rest of the paper, we write ADP to refer to an approximate DP table, which approximates the exact DP table DP. Let $\beta > 1$. We say that ADP *β-approximates* DP if $\mathsf{DP}(i,j) \leq \mathsf{ADP}(i,j) \leq \beta \mathsf{DP}(i,j)$ for all $i \in \mathcal{I}, j \in \mathcal{J}$.

▶ **Definition 8.** *A DP table is* (h, α, p)-well-behaved *if it satisfies the following conditions:*

1. *(Monotonicity:) For all $i \in \mathcal{I}$, the function $\mathsf{DP}(i, \cdot)$ is monotone.*
2. *(Dependency graph:) The dependency graph is a DAG and $|\mathrm{Reach}(i)| \leq h$ for all $i \in \mathcal{I}$.*
3. *(Sensitivity:) Suppose $\beta > 1$ and for all $i' \in \mathrm{In}(i)$, we obtain a β-approximation $\mathsf{ADP}(i', \cdot)$ of $\mathsf{DP}(i', \cdot)$. Then applying $\mathcal{P}_i$ on the $\mathsf{ADP}(i', \cdot)$ yields a β-approximation of $\mathsf{DP}(i, \cdot)$, i.e.,*

$$\mathsf{DP}(i, \cdot) \leq \mathcal{P}_i(\{\mathsf{ADP}(i', \cdot) \colon i' \in \mathrm{In}(i)\}) \leq \beta \cdot \mathsf{DP}(i, \cdot).$$

4. *(Pieces:) For each procedure $\mathcal{P}_i$ there exists an approximate procedure $\tilde{\mathcal{P}}_i$ such that:*
   (a) $\tilde{\mathcal{P}}_i(\{\mathsf{ADP}(i', \cdot) \colon i' \in \mathrm{In}(i)\})$ is an α-approximation of $\mathcal{P}_i(\{\mathsf{ADP}(i', \cdot) \colon i' \in \mathrm{In}(i)\})$,
   (b) $\tilde{\mathcal{P}}_i$ can be computed as the composition of a constant number of operations from Lemma 6 and and at most one application of Lemma 7, and
   (c) $\tilde{\mathcal{P}}_i$ returns a monotone piecewise constant function with at most p pieces.

The definition is motivated in the following way: our operations on the piecewise constant functions have efficient running times when the functions are monotone and have few pieces. This is ensured by Properties (1), 4(b), and 4(c). Next, rounding errors cannot compound too much if each row can only reach h other rows and the sensitivity condition is satisfied. This is ensured by Properties (2), (3), and 4(a).

Even though the definition might look slightly technical at first glance, it applies in many settings. In particular, Property (2) is satisfied when the dependency graph is a rooted tree of height h in which all edges point towards the root; this is the case in all of our applications. The other conditions are immediately satisfied by our DP for 0-1 Knapsack in Section 3 and the DP for simultaneous source location (see [17]). However, our DP for balanced graph partitioning violates Property (4b) of Definition 8. Hence, in the full version [17] we will also consider a weaker assumption which, however, will not allow for nice black-box results, such as Theorems 9 and 10 below.

Next, we state our main results. They imply that we obtain static $(1+\epsilon)$-approximation algorithms running in near-linear time and space for $(\tilde{O}(1), \ln(1+\epsilon)/\tilde{O}(1), \tilde{O}(1))$-well-behaved DPs. They also show that under this assumption, we can dynamically maintain $(1+\epsilon)$-approximate DP solutions with polylogarithmic update times.

Our main theorem for static algorithms is as follows.

▶ **Theorem 9.** *Consider an (h, α, p)-well-behaved DP. Then we can compute an approximate DP table ADP which α^{h+1}-approximates DP in time and space $O(|\mathcal{I}| \cdot p^2 \log(p))$.*

Later, we will apply the theorem to DPs with dependency trees of logarithmic heights $h = O(\log n)$, we will set the approximation ratio to $\alpha = \ln(1+\epsilon)/(h+1)$, and the number of pieces to $p = \mathrm{polylog}(W)$. This will yield our desired algorithms with near-linear running time $\tilde{O}(|\mathcal{I}|)$ and space usage. Note that this is a big improvement upon the brute-force running times and space usages of $\Omega(|\mathcal{I}| \cdot |\mathcal{J}|)$.

The proof of the theorem follows from observing that when moving from one vertex to another in the dependency graph, we lose a multiplicative α-factor in the approximation ratio; as each vertex can only reach h other vertices, this will compound to at most α^{h+1}. Combining the assumptions on the functions $\tilde{\mathcal{P}}_i$ and the results from Lemmas 6 and 7, we get that each row $\mathsf{ADP}(i,\cdot)$ can be computed in time $O(p^2\log(p))$ which gives $O(|I|\cdot p^2\log(p))$ total time.

We also give the following extension to the dynamic setting which shows that if one of the DP rows changes, we can update *the entire table* efficiently.

▶ **Theorem 10.** *Consider an (h,α,p)-well-behaved DP and suppose that row i is changed. Then we can update our approximate DP table* ADP *such that after time $O(h\cdot p^2\log(p))$ it is an α^{h+1}-approximation of* DP*.*

As before, we will typically use the theorem with $h=O(\log n)$, $\alpha=\ln(1+\epsilon)/(h+1)$ and $p=\operatorname{polylog}(W)$. This will then result in our desired polylogarithmic update times. Note that this is a significant speedup compared to storing the DP tables using two-dimensional arrays: in that case even updating *a single row* would take time $\Omega(|\mathcal{J}|)$, which in many applications would already be linear in the size of the input.

The theorem follows from observing that after a row i changes, we only have to update those rows which can be reached from i in the dependency graph. But these can be at most h and each of them can be updated in time $O(p^2\log(p))$ by Lemmas 6 and 7.

3 Fully Dynamic Knapsack

In *0-1 knapsack*, the input consists of a knapsack size $B\in\mathbb{R}_+$ and a set of n items, where each item $i\in[n]$ has a weight $w_i\in\mathbb{R}_+$ and a price $p_i\in[1,\infty)$. The goal is to find a set of items I that maximizes $\sum_{i\in I}p_i$ while satisfying the constraint $\sum_{i\in I}w_i\le B$. For a set of items $I\subseteq[n]$, we refer to the sum $\sum_{i\in I}w_i$ as *the weight of I*.

For the rest of this section we set $W=\sum_i p_i$ and $t=\sum_{i\in[n]}w_i$.

Next, we first derive a dynamic algorithm with update time $\tilde{O}(\log^3(n)\log^2(W)/\epsilon^2)$ which is based on our framework for DPs with monotone rows. Then we will use this algorithm as a subroutine to obtain a faster algorithm with update time $\tilde{O}(\log^2(nW)/\epsilon^2)$ in Section 3.2; this will prove Theorem 1.

▶ **Theorem 1.** *Let $\epsilon>0$. There exists an algorithm for fully dynamic knapsack that maintains a $(1+\epsilon)$-approximate solution with worst-case update time $\frac{1}{\epsilon^2}\log^2(nW)\operatorname{polylog}\left(\frac{1}{\epsilon}\log(nW)\right)$.*

Below we will also show that we can return the maintained solution I in time $O(|I|)$ and that we answer queries whether a given item $i\in[n]$ is contained in I in time $O(1)$. This matches the query times of [11].

3.1 Knapsack via Convolution of Monotone Functions

First, we give a brief recap of the knapsack approach by Chan [7]. We consider the more general problem of approximating the function $f_J\colon[0,t]\to\mathbb{R}_+$, where $J\subseteq[n]$ is a set of items and

$$f_J(x)=\max\left\{\sum_{i\in I}p_i\colon\ \sum_{i\in I}w_i\le x,\ \ I\subseteq J\right\}. \tag{2}$$

Intuitively, the value $f_J(x)$ corresponds to the best possible knapsack solution if we can only pick items which are contained in J and if the weight of the solution can be *at most* x. Therefore, $f_{[n]}(B)$ corresponds to the optimum solution of the global knapsack instance.

Note that for each $J \subseteq [n]$, $f_J(x)$ is a monotonically increasing piecewise constant function: Indeed, consider $x' \leq x$. Any solution $I \subseteq J$ that is feasible for x' (i.e., the weight of I is at most x') is also a feasible solution for x. Thus, $f_J(x') \leq f_J(x)$ and, therefore, f_J is monotonically increasing. Furthermore, f_J is piecewise constant since each function value $f_J(x)$ corresponds to a solution $I \subseteq J$ and the number of choices for $I \subseteq J$ is finite.

Next, note that if we have two disjoint subsets $J_1, J_2 \subseteq [n]$ then it holds that $f_{J_1 \cup J_2}$ is the $(\max, +)$-convolution of f_{J_1} and f_{J_2}, i.e., for all x it holds that

$$f_{J_1 \cup J_2}(x) = \max_{\bar{x}} f_{J_1}(\bar{x}) + f_{J_2}(x - \bar{x}).$$

This can be seen by observing that for each x, the optimum solution I for the instance $J_1 \cup J_2$ with weight at most x can be split into two disjoint solutions $I_1 \subseteq J_1$ and $I_2 \subseteq J_2$ such that I_1 has weight $\bar{x}$ and I_2 has knapsack weight at most $x - \bar{x}$ (for suitable choice of $\bar{x} \in [0, x]$). We conclude that if we have two knapsack instances over disjoint sets of items J_1 and J_2, then we compute the solution for the knapsack instance with items $J_1 \cup J_2$ by computing the $(\max, +)$-convolution of f_{J_1} and f_{J_2}.

The Exact DP. The previous paragraphs imply a simple way of computing the exact solution of a knapsack instance: For each item $i \in [n]$, compute the function $f_{\{i\}}$ and then recursively merge the solutions for sets of size 2^j, $j = 1, \ldots, \lceil \log n \rceil$, by computing $(\max, +)$-convolutions until we have computed the global solution $f_{[n]}$. We perform the recursive merging of the solutions using a balanced binary tree, resulting in a tree of height $O(\log n)$.

More concretely, we build a rooted balanced binary tree T with n leaf nodes, where all edges point towards the root. We have one leaf $f_{\{i\}}$ for each item i. Each internal node u in T is associated with a function f_{J_u} as per Equation (2), where J_u is the set of all items in the subtree rooted at u. To simplify notation, we will also refer to f_{J_u} as f_u.

Now we consider the exact computation of the DP. This will reveal the procedures $\mathcal{P}_i$ from Definition 8. As base case, for each $i \in [n]$, the i'th leaf of T contains the function $f_{\{i\}}$, which is a piecewise constant function that has value 0 on the interval $[0, w_i)$ and value p_i on the interval $[w_i, t]$.

Next, in each internal node u of T with children u_1 and u_2, we set f_u to the $(\max, +)$-convolution of f_{u_1} and f_{u_2}. By induction it can be seen that for every node u in T, it holds that $J_u = J_{u_1} \cup J_{u_2}$ and thus J_u is the set of all items whose corresponding leaf is contained in the subtree T_u. Hence, for the root r of T it holds that $f_r = f_{[n]}$ and $f_r(B)$ is the optimal solution for the global knapsack instance.

In the following, we check that our DP satisfies Properties (1–3) of Definition 8.

First, note that the tree T from above is also the dependency graph of our DP. Hence, our DP has a row for every vertex of T and thus $O(n)$ rows in total. Furthermore, since T has height $O(\log n)$ and all edges point towards the root, every vertex can reach at most $h = O(\log n)$ vertices. Hence, Property (2) of Definition 8 is satisfied.

Second, we observe that in both cases above, the function $f_{\{i\}}$ and f_u which correspond to the rows of our DP table are monotonically increasing (we argued this above for all functions f_J). Thus, Property (1) is satisfied.

Third, observe that Property (3) is also satisfied since $(\max, +)$-convolution satisfies our sensitivity condition.

We conclude that the first three properties of Definition 8 are satisfied. Unfortunately, this does not yet imply that we can obtain efficient algorithms: Note that if we compute the exact DP bottom-up then we compute one convolution per node and thus the total running time of this approach is $O(n \cdot t(p))$, where p is an upper bound on the number of pieces in our functions and $t(p)$ is the time it takes to compute a $(\max, +)$-convolution of two functions with p pieces. However, observe that computing the convolutions can potentially take a large amount of time because the number of pieces of the functions might grow quadratically after each convolution (see Lemma 7). We will resolve this issue below using rounding.

The Approximate DP. Next, we consider approximations which will reveal the functions $\tilde{\mathcal{P}}_i$ from Definition 8.

First, note that we need to compute $(\max, +)$-convolutions of monotonically increasing functions efficiently. We observe that this can be done efficiently using our subroutine from Lemma 7 for the $(\min, +)$-convolution of monotonically decreasing functions: Indeed, suppose that f is the $(\max, +)$-convolution of two monotonically increasing functions g and h, then for all x it holds that

$$f(x) = \max_{\bar{x}}\{g(\bar{x}) + h(x - \bar{x})\} = -\min_{\bar{x}}\{-g(\bar{x}) + (-h(x - \bar{x}))\}.$$

Now observe that $-g$ and $-h$ are monotonically decreasing functions and, therefore, $f = -((-g) \oplus (-h))$, where $\oplus$ denotes the $(\min, +)$-convolution. Thus, we can use the efficient routine for $(\min, +)$-convolution from Lemma 7 with the same running time.[6]

Now we can define the subroutines $\tilde{\mathcal{P}}_i$. Let $\delta > 0$ be a parameter that we set later. Whenever we compute a function f_u via a $(\max, +)$-convolution, we use the efficient subroutine from Lemma 7. After computing the convolution, we set $f_u = \lceil f_u \rceil_{1+\delta}$ via the subroutine from Lemma 6.

Observe that this approach satisfies Property (4a) of Definition 8 with $\alpha = 1 + \delta$. Furthermore, Property (4b) is satisfied since we only use a single convolution and a single rounding step. Finally, Property (4c) is also satisfied because the resulting function is monotone and has $p = O(\log_{1+\delta}(W))$ after the rounding.

The above arguments show that our DP is (h, α, p)-well-behaved for $h = \lceil \log n \rceil$, $\alpha = 1+\delta$, $\delta = \ln(1+\epsilon)/\lceil \log n \rceil$ and $p = O(\log_{1+\delta}(W)) = O(\log(W)/\delta)$. Hence, Theorem 10 immediately implies the following lemma.

▶ **Lemma 11.** *Let $\epsilon > 0$. There exists an algorithm that computes a $(1 + \epsilon)$-approximate solution for 0-1 knapsack in time $n \cdot \frac{1}{\epsilon^2} \log^2(n) \log^2(W) \cdot \text{polylog}(\frac{1}{\epsilon} \log(nW))$.*

We note that we can return our solution I in time $|I| \log(n) \cdot \text{polylog}(\frac{1}{\epsilon} \log(nW))$ as follows. Recall that our global objective function value is achieved by $f_r(B)$ and that $f_r(B) = f_{u_1}(\bar{x}^*) + f_{u_2}(B - \bar{x}^*)$, where u_1 and u_2 are the nodes below the root node r of the dependency tree. Now using the second part of Lemma 7 we can get the value of $\bar{x}^*$ in time $O(\log p)$. If $f_{u_1}(\bar{x}^*) > 0$ we recurse on $f_{u_1}(\bar{x}^*)$ and if $f_{u_2}(B - \bar{x}^*) > 0$ we also recurse on $f_{u_2}(B - \bar{x}^*)$. At some point we will reach a leaf node i and we include i in the solution iff $f_{\{i\}}(x) > 0$. Note that since we only recurse for function values which are strictly larger than zero, for each item that we include into the solution we have to follow a single path in the dependency tree of height $O(\log n)$ and our work in each internal node is bounded by $O(\log p)$. This gives the total time of $O(|I| \log(n) \log(p))$ and our claim follows from our choice of p above.

[6] We note that, formally, Lemma 7 can only be applied on functions with non-negative values. However, this can be achieved by adding a number C to $-g$ and $-h$, which is an upper bound on the values taken by g and h, and at the end we subtract the constant function $2C$, i.e., we set $f = -((-g+C) \oplus (-h+C)) - 2C$.

Extension to the Dynamic Setting. Next, we extend our result to the dynamic setting. For the sake of simplicity, we assume that n is an upper bound on the maximum number of available items (items in S) and given to our algorithm in the beginning.[7] We consider update operations that insert and delete items from the set. More concretely, we consider the following update operations:

- *insert*(p_i, w_i), in which i is added to S by setting the price and weight of item i to $p_i \in W_\infty$ and $w_i \in \mathbb{R}_+$, respectively, and
- *delete*(i), where item i is removed from the set of items.

Our implementation is as follows. In the preprocessing phase, we build the same tree T as above and use the subroutine from above to compute the function $f_{\{i\}}$. For the operation *delete*(i), we set $p_i = 0$ and $w_i = 0$, which changes exactly one row of our DP table. For the operation *insert*(p_i, w_i), we set the price and weight of item i to p_i and w_i, resp., which again changes a single row in our DP table. After changing such a row, we recompute the global DP solution via Theorem 10. Since the DP is (h, α, p)-well-behaved with the same parameters as above, the theorem implies the following proposition.

▶ **Proposition 12.** *Let $\epsilon > 0$. There exists an algorithm for the fully dynamic knapsack problem that maintains a $(1+\epsilon)$-approximate solution with worst-case update time $\frac{1}{\epsilon^2}\log^3(n)\log^2(W) \cdot \text{polylog}\left(\frac{1}{\epsilon}\log(nW)\right)$.*

Observe that with the same procedure as for the static algorithm, we can return our solution I in time $|I|\log(n) \cdot \text{polylog}(\frac{1}{\epsilon}\log(nW))$. Furthermore, given an item $i \in [n]$, we can return whether $i \in I$ in time $\log(n) \cdot \text{polylog}(\frac{1}{\epsilon}\log(nW))$. This can be done by using the same query procedure as in the static setting, where we only recurse on the unique subtree in the depedency tree that contains the node for item i.

We note that the above proposition already improves upon the update time in the result of Eberle et al. [11] in terms of the dependency on $\frac{1}{\epsilon}$ but it has a worse dependency on $\log(nW)$. However, our query time is slower than the $O(1)$-time query operation in [11]. We will resolve these issues in the next subsection, where we will use the algorithm from Proposition 12 as a subroutine.

3.2 Dynamically Maintaining a Small Instance

Next, we we obtain a faster dynamic algorithm with update time $\tilde{O}(\frac{1}{\epsilon^2}\log^2(nW))$ by combining the algorithm from Proposition 12 and with ideas from Eberle et al. [11]. Our high-level approach is as follows. First, we partition the items into a small number of *price classes*. Then we take a few items of small weight from each price class. This will give a very small knapsack instance X for which we maintain an almost optimal solution using the subroutine from Proposition 12; since this instance is very small (i.e., $|X| \ll n$), the update time for maintaining this instance essentially becomes $O(\frac{1}{\epsilon^2}\log^2(W))$, i.e., we lose the $O(\log^3 n)$ term that made the update time in the proposition too costly. For the rest of the items which are not contained in X, we show that we can compute a good solution using fractional knapsack, which can be easily solved using a set of binary search trees. Then it remains to show that the combination of the two solutions is a $(1+\epsilon)$-approximation.

[7] It is possible to drop this assumption using an amortization argument. More concretely, every time the number of items is less than $n/2$ or more than n, we rebuild the data structure with a new value of n. Each rebuild can be done in time $O(nt(n))$, where $t(n)$ is our update time. Since this only happens after $\Omega(n)$ updates occured, we can amortize this cost over the updates that appeared since the last rebuild.

The main differences of our algorithm and the one by Eberle et al. [11] are as follows. Eberle et al. also partition the items into a small number of price classes. They also combine solutions for a small set of heavy items X and solutions based on fractional knapsack for the other items. However, they have to enumerate many different sets X and they also guess the approximate price of the fractional knapsack solution; more concretely, they enumerate $\Theta(\frac{1}{\epsilon^2}\log(W))$ choices for X and the number of guesses they have to make for the fractional knapsack solution is $\Theta(\frac{1}{\epsilon}\log(W))$. Thus they have to consider $\Theta(\frac{1}{\epsilon^3}\log^2(W))$ guesses and for each of them they have to compute approximate solutions, which takes time $\Theta(\frac{1}{\epsilon^4})$ for each X since they have to run a static algorithm from scratch. In our approach, we only have to consider a single set X which we maintain in our data structure from Proposition 12, which saves us a lot of time. Furthermore, the piecewise constant function, in which we store the solution for X, essentially "guides" our $\Theta(\frac{1}{\epsilon}\log(W))$ guesses for the weight of fractional knapsack solution. In our analysis we have to be slightly more careful to ensure that our guesses for the weight of the fractional knapsack solution guarantee the correct approximation ratio.

Definitions. We assume that $\epsilon < 1$ and that $1/\epsilon$ is an integer. More concretely, we run the algorithm with $\epsilon' = \max\{\frac{1}{i} : \frac{1}{i} \leq \epsilon, i \in \mathbb{N}\}$. Set $L = \lceil \log_{1+\epsilon}(W) \rceil$ and recall that we set $W = \sum_i p_i$.

We define the *price classes* $V_\ell = \{i : (1+\epsilon)^\ell \leq p_i < (1+\epsilon)^{\ell+1}\}$. In the following, we assume that all items from price class V_ℓ have price exactly $(1+\epsilon)^{\ell+1}$. We only lose a factor of $1+\epsilon$ by making this assumption. Furthermore, we set $V_\ell^{1/\epsilon}$ to the set of $1/\epsilon$ items from V_ℓ with smallest weights w_i (breaking ties arbitrarily). We also define $V'_\ell = V_\ell \setminus V_\ell^{1/\epsilon}$.

Next, we set $X = \bigcup_{\ell \geq 0} V_\ell^{1/\epsilon}$ and $Y = \bigcup_{\ell \geq 0} V'_\ell$ for all $\ell \geq 0$. Note that X and Y partition the set of items and $|X| = \frac{1}{\epsilon} \cdot L = O(\epsilon^{-2}\log(W))$.

Now our strategy is to use our algorithm from Proposition 12 to maintain a solution for the items in X. Then we show how we can combine the solution for X with a solution for Y that is based on fractional knapsack and a charging argument.

Data Structures. For each $\ell \in [L]$, we maintain V_ℓ sorted non-decreasingly by weight.

We also maintain the set X in a binary search tree, in which we sort the items by their index, and we maintain our data structure from Proposition 12 on the items in X.

Furthermore, let $U_\ell = \bigcup_{\ell' \leq \ell} V'_{\ell'}$ denote the set of all items that are not contained in X and of price class at most ℓ. For each ℓ, we maintain the set U_ℓ in a binary search tree T in which the items are stored as leaves and sorted by their density $\frac{p_i}{w_i}$. In each internal node u of T, we store the total weight of the items in the subtree T_u rooted at u and the total profit of the items in T_u. Observe that this allows us to answer queries of the type: "Given a budget b, what is the value of the optimal fractional[8] knapsack solution in U_ℓ with weight at most b?" in time $O(\log n)$.

Updates. Now consider an item insertion or deletion and suppose that the updated item is of price class V_ℓ. We first update the sets V_ℓ, $U_{\ell'}$ for $\ell' \leq \ell$ and the sets X and Y. Note that for each of these sets at most one item can be removed and inserted. Thus, these steps can be done in time $O(\ell \cdot \log(n)) = O(\epsilon^{-1}\log(W)\log(n))$.

[8] In fractional knapsack, we may use items fractionally. An optimal solution is achieved by sorting the items items by their density and greedily adding items to the solution until we have used up our budget b. This approach uses at most one item fractionally (namely, the one at which we use up our budget).

Next, if X changed in the previous step, then we also perform the corresponding updates in the data structure from Proposition 12. Since $|X| = O(\epsilon^{-2}\log(W))$ holds by construction of X, the update operations for the data structure maintaing the knapsack solution for X take a total time of

$$O\left(\epsilon^{-2}\log^3(|X|)\log^2(W)\cdot\operatorname{polylog}\left(\frac{1}{\epsilon}\log(|X|\,W)\right)\right)$$
$$= O\left(\epsilon^{-2}\log^2(W)\cdot\operatorname{polylog}\left(\frac{1}{\epsilon}\log(nW)\right)\right).$$

Furthermore, we can explicitly write down our solution I_X for the items in X in time $\epsilon^{-2}\log(W)\cdot\operatorname{polylog}(\frac{1}{\epsilon}\log(nW))$ since $|X| = O(\epsilon^{-2}\log(W))$. Also, for each $i \in I_X$, we can set a bit indicating that $i \in I_X$. Note that the time for writing down I_X and setting the bits is subsumed by the update time above.

Queries. *Returning the value of a solution:* We return the value of a global knapsack solution as follows.

Consider the data structure from Proposition 12 which maintains a solution for the items in X. Note that this solution is stored as a piecewise constant function with $p \leq L$ pieces and consider the list representation $(x_1, y_1), \ldots, (x_p, y_p)$ of this function.

Our strategy is as follows: For each $i = 1, \ldots, p$, we consider a solution which spends budget x_i on items in X and budget $B - x_i$ on items in Y. Then we take the maximum over all of the solutions we have considered. More concretely, for given $i = 1, \ldots, p$, we obtain our solution as follows. We pick ℓ_i such that $(1+\epsilon)^{\ell_i} = \lceil \epsilon \cdot y_i \rceil_{1+\epsilon}$ (see Lemma 13 below for a justification of this choice). Now we use the binary search tree for U_{ℓ_i} to find the highest profit that we can obtain from fractional knapsack on items in $U_{\ell_i} \subseteq Y$ if we can spend budget at most $b = B - x_i$. Let y_i' be the value of this query after removing any profit that we gain from the (at most one) fractionally cut item. We also store the density of the final item that is contained in the fractional knapsack solution. Now we return the maximum of $y_i + y_i'$ over all $i = 1, \ldots, p$.

Note that since the solution for X has at most $L = O(\epsilon^{-1}\log(W))$ pieces and for each of them we perform a single query in a binary search tree, the total time for return the solution value is $O(\epsilon^{-1}\log(W)\log(n))$. Note that this time is subsumed by the update time.

Returning the entire solution: Now we can return our global solution I in time $O(|I|)$ as follows. Observe that I is composed of the solution I_X for the items in X and of the items in the fractional knapsack solution. During our updates, we already stored the items in I_X and can write them down in time $O(|I_X|)$. Next, to return the items from the fractional knapsack solution, recall that we stored the density of the final item in the fractional knapsack solution. Thus, we only have to output the items ordered non-decreasingly by their density, while we are above the desired density-threshold. This can be done in time linear in the size of the fractional knapsack solution. This is essentially the same query procedure as in [11].

Returning whether an item is in the solution: Furthermore, observe that the above implies that we can answer whether an item $i \in [n]$ is contained in our solution in time $O(1)$: If $i \in X$ then we already stored a bit whether $i \in I_X$. If $i \notin X$ then we can check whether i is in the fractional knapsack solution by checking whether its density is above or below the threshold given by the final item in the fractional knapsack solution.

Analysis. We start by making some simplifications to OPT. We let OPT' denote the version of OPT in which for each $\ell \in [L]$, we pick the $|\mathrm{OPT} \cap V_\ell|$ items of smallest weight from V_ℓ. This only loses a factor of $1+\epsilon$. Next, define $\mathrm{OPT}'_X = \mathrm{OPT}' \cap X$ and $\mathrm{OPT}'_Y = \mathrm{OPT}' \cap Y$. Observe that by how we picked OPT', it holds that $\mathrm{OPT}'_Y \cap V_\ell \neq \emptyset$ iff $\left|\mathrm{OPT}' \cap V_\ell\right| > 1/\epsilon$.

Let p_X denote the total price of items in OPT'_X and let w_X denote the total weight of the items in OPT'_X. Let f denote the piecewise constant function that stores the solution for the items in X. Observe that by Proposition 12 we have that

$$p_X \leq f(w_X) \leq (1+\epsilon)p_X.$$

Also, the function value $f(w_X)$ is part of a piece (x_{i^*}, y_{i^*}) with $x_{i^*} \leq w_X$ and $y_{i^*} = f(w_X)$.

The next lemma justifies why we set ℓ_i such that $(1+\epsilon)^{\ell_i} = \lceil \epsilon \cdot y_i \rceil_{1+\epsilon}$ in our algorithm. To this end, let ℓ_{i^*} be such that $(1+\epsilon)^{\ell_{i^*}} = \lceil \epsilon \cdot y_{i^*} \rceil_{1+\epsilon}$ and let ℓ_Y be the price class of the most valuable item in OPT'_Y. In the lemma we show that $\ell_{i^*} \geq \ell_Y$. We will use this to show that our solution for X of profit y_{i^*} is valuable enough such that we can charge a fractionally cut item from fractional knapsack onto the solution from X and only lose a factor of $(1+\epsilon)^2$.

▶ **Lemma 13.** *It holds that $\ell_{i^*} \geq \ell_Y$.*

Proof. Since $\text{OPT}'_Y \cap V'_{\ell_Y} \neq \emptyset$, $\left|\text{OPT}' \cap V_{\ell_Y}\right| > 1/\epsilon$ and thus OPT'_X contains all $1/\epsilon$ items from $V^{1/\epsilon}_{\ell_Y}$. Hence, $p_X \geq \frac{1}{\epsilon} \cdot (1+\epsilon)^{\ell_Y}$. From above we get $f(w_X) = y_{i^*}$ and $f(w_X) \geq p_X$. By choice of ℓ_{i^*},

$$(1+\epsilon)^{\ell_{i^*}} = \lceil \epsilon \cdot y_{i^*} \rceil_{1+\epsilon} = \lceil \epsilon \cdot f(w_X) \rceil_{1+\epsilon} \geq \lceil \epsilon \cdot p_X \rceil_{1+\epsilon} \geq \left\lceil \epsilon \cdot \frac{1}{\epsilon}(1+\epsilon)^{\ell_Y} \right\rceil_{1+\epsilon} = (1+\epsilon)^{\ell_Y}.$$

This implies $\ell_{i^*} \geq \ell_Y$. ◀

Next, consider the the fractional knapsack solution that we obtain from our query. Note that this solution has a profit that is at least as large as the profit of OPT'_Y (since fractional knapsack is a relaxation of 0-1 knapsack). Furthermore, the fractional solution uses at most one item fractionally and this item is from $U_{\ell_{i^*}}$ and has value at most $(1+\epsilon)^{\ell_{i^*}} = \lceil \epsilon \cdot y_{i^*} \rceil_{1+\epsilon} \leq (1+\epsilon)\epsilon \cdot y_{i^*}$. Thus, we can charge this item on OPT'_X and lose a factor of at most $(1+\epsilon)^2$.

We conclude that the solution $y_{i^*} + y'_{i^*}$ is a $(1+\epsilon)^{O(1)}$-approximation of OPT. Combining this with our previous running time analysis, we obtain Theorem 1.

References

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

2 Konstantin Andreev, Charles Garrod, Daniel Golovin, Bruce M. Maggs, and Adam Meyerson. Simultaneous source location. *ACM Trans. Algorithms*, 6(1):16:1–16:17, 2009.

3 Konstantin Andreev and Harald Räcke. Balanced graph partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.

4 Wolfgang W. Bein, Mordecai J. Golin, Lawrence L. Larmore, and Yan Zhang. The knuth–yao quadrangle-inequality speedup is a consequence of total monotonicity. *ACM Trans. Algorithms*, 6(1):17:1–17:22, 2009.

5 Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. In *Algorithm Engineering - Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 117–158. Springer, 2016.

6 Rainer E. Burkard, Bettina Klinz, and Rüdiger Rudolf. Perspectives of monge properties in optimization. *Discret. Appl. Math.*, 70(2):95–161, 1996.

7 Timothy M. Chan. Approximation schemes for 0-1 knapsack. In *SOSA*, volume 61, pages 5:1–5:12, 2018.

8 Timothy M. Chan. Near-optimal randomized algorithms for selection in totally monotone matrices. In *SODA*, pages 1483–1495, 2021.

9 Spencer Compton, Slobodan Mitrovic, and Ronitt Rubinfeld. New partitioning techniques and faster algorithms for approximate interval scheduling. *CoRR*, abs/2012.15002, 2020. arXiv:2012.15002.

10 Yihe Dong, Piotr Indyk, Ilya P. Razenshteyn, and Tal Wagner. Learning space partitions for nearest neighbor search. In *ICLR*, 2020.

11 Franziska Eberle, Nicole Megow, Lukas Nölke, Bertrand Simon, and Andreas Wiese. Fully Dynamic Algorithms for Knapsack Problems with Polylogarithmic Update Time. In *FSTTCS*, volume 213, pages 18:1–18:17, 2021.

12 David Eppstein, Zvi Galil, and Raffaele Giancarlo. Speeding up dynamic programming. In *FOCS*, pages 488–496. IEEE Computer Society, 1988.

13 Guy Even, Joseph Naor, Satish Rao, and Baruch Schieber. Fast approximate graph partitioning algorithms. *SIAM J. Comput.*, 28(6):2187–2214, 1999.

14 Uriel Feige and Robert Krauthgamer. A polylogarithmic approximation of the minimum bisection. *SIAM J. Comput.*, 31(4):1090–1118, 2002.

15 Andreas Emil Feldmann and Luca Foschini. Balanced partitions of trees and applications. *Algorithmica*, 71(2):354–376, 2015.

16 Zvi Galil and Kunsoo Park. Dynamic programming with convexity, concavity, and sparsity. *Theor. Comput. Sci.*, 92(1):49–76, 1992.

17 Monika Henzinger, Stefan Neumann, Harald Räcke, and Stefan Schmid. Dynamic maintenance of monotone dynamic programs and applications. *CoRR*, abs/2301.01744, 2024. Full version of this paper. arXiv:2301.01744.

18 Monika Henzinger, Stefan Neumann, and Andreas Wiese. Dynamic approximate maximum independent set of intervals, hypercubes and hyperrectangles. In *SoCG*, volume 164 of *LIPIcs*, pages 51:1–51:14, 2020.

19 George Karypis and Vipin Kumar. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices, 1997.

20 Donald E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14–25, 1971.

21 Gary L. Miller, Richard Peng, Russell Schwartz, and Charalampos E. Tsourakakis. Approximate dynamic programming using halfspace queries and multiscale monge decomposition. In *SODA*, pages 1675–1682, 2011.

22 Gaspard Monge. Mémoire sur la théorie des déblais et des remblais. *Mem. Math. Phys. Acad. Royale Sci.*, pages 666–704, 1781.

23 Peter Sanders and Christian Schulz. Think locally, act globally: Highly balanced graph partitioning. In *SEA*, volume 7933, pages 164–175, 2013.

24 F. Frances Yao. Efficient dynamic programming using quadrangle inequalities. In *STOC*, pages 429–435, 1980.