Dynamic Maintenance of Monotone Dynamic Programs and Applications

Monika Henzinger, Stefan Neumann (@StefanResearch), Harald Räcke, Stefan Schmid (@schmiste_ch)
Making Dynamic Programming Dynamic

Monika Henzinger, Stefan Neumann (@StefanResearch), Harald Räcke, Stefan Schmid (@schmiste_ch)
Dynamic Programming (DP)

- Dynamic programming (DP) is a fundamental algorithm design paradigm
  - Complex problem is broken up into simpler subproblems, original problem is solved by combining the solutions
- Often prohibitively costly in practice
  - We often need time $\Omega(n^2)$ to compute solution
  - We even need $\Omega(n^2)$ space to store the DP table!
Speeding Up DPs

- Lots of different conditions that allow solving DPs more quickly
  - Total monotonicity, Monge property, certain convexity and concavity properties, Knuth–Yao quadrangle-inequality, …
  - SMAWK algorithm computes solution for totally monotone DP tables in linear time $O(n + m)$
  - $n = \#\text{rows, } m = \#\text{columns}$
  - Naïve computation would take time $\Omega(mn)$

$T_{i,j} + T_{i',j'} \leq T_{i,j'} + T_{i',j}$
for all $i < i'$ and $j < j'$

Example of totally monotone matrix

This line of research focussed on static algorithms

example from https://courses.engr.illinois.edu/cs473/sp2016/notes/06-sparsedynprog.pdf
Dynamic DPs?

- **Dynamic algorithms:**
  Input is changing over time and we want to maintain a solution

- **Goal:** Get small (polylogarithmic) update time

- Remember what I said about DPs earlier?

  - “Complex problem is broken up into simpler subproblems, original problem is solved by combining the solutions”

  ➡ Quite similar to how many dynamic algorithms are developed, so it should be “easy” to turn DPs into dynamic algorithms?
Question:
Is there a condition that implies that a DP can be made dynamic?
Problem: Arrays Do Not Work for Dynamic DPs

- **Problem:** When a single entry in the DP table changes, we often need to recompute $\Omega(m)$ entries — even in just a single row.

- If we store the DP as a two-dimensional array, this rules out the polylog update times that we hoped for.

→ Can we bypass this limitation by storing the DP table in a smarter way?
Our Results
Answer:
Yes! There is a condition that implies that a DP can be made dynamic!

If the rows are monotonically increasing and we allow approximation*, then we can store the DP table in a smarter way

*... a few more technical conditions apply
Notation

- Consider an $n \times m$ DP table $T$ with entries in $[0, W]$
- The rows are monotone if $T_{i,j} \leq T_{i,j+1}$ for all $i$ and $j$
- We say that a DP table $\tilde{T}$ is an $\alpha$-approximation of $T$ if $T_{i,j} \leq \tilde{T}_{i,j} \leq \alpha \cdot T_{i,j}$ for all $i$ and $j$
- We assume the DP can be computed row-by-row
- Dependency tree for the DP: tree which encodes whether computing row $i$ requires solution for row $j$
General Framework to Make DPs Dynamic

- **Assumption:** DP has $n$ monotone rows and dependency tree of height $O(\log n)$ and rows are “easy to compute”

- **Static result:** We can compute a $(1 + \varepsilon)$-approximation of the DP table in near-linear time and space $\tilde{O}(n)$
  
  - Every entry is correct up to a multiplicative $(1 + \varepsilon)$-factor
  
  - Much more efficient than writing down the entire table as an array in time $\Omega(mn)$

- **Dynamic result:** When entries in the DP table change, we can update the entire table in polylogarithmic time $O(1)$
**Balanced Graph Partitioning**

- **Problem:**
  Given a graph $G = (V, E, w)$ partition the vertices into $k$ groups $V_1, \ldots, V_k$, such that

  each group $V_i$ contains $n/k$ vertices and

  \[
  \text{cut}(V_1, \ldots, V_k) = \sum_{(u,v) \in E: u \in V_i, v \in V_j} w(u,v) \text{ is minimized}
  \]

- **Bicriteria version:** Each group may contain up to $(1 + \varepsilon)n/k$ vertices, we compare cut-value against optimal solution that has to satisfy the constraint exactly

- Important pre-processing step in many distributed graph algorithms, popular heuristics METIS have thousands of citations

---

Picture taken from Rais et al., https://doi.org/10.20965/jaciii.2019.p0005
Balanced Graph Partitioning

- **Our results:**
  - First static near-linear time algorithm computing a bicriteria $O(\lg^4 n)$-approximation
    - Best polynomial-time algorithm computes a bicriteria $O(\lg^{1.5} n \lg \lg n)$-approximation in time $\Omega(n^4)$ (Feldmann and Foschini, 2015)
  - First dynamic algorithm with subpolynomial update time for unweighted graphs which maintains a bicriteria $n^{o(1)}$-approximation under edge insertions and deletions (update time $O_{k,\epsilon}(1) \cdot n^{o(1)}$)
  - We simplify and generalize the DP by Feldmann and Foschini

Picture taken from Rais et al., https://doi.org/10.20965/jaciii.2019.p0005
Fully Dynamic Knapsack

- **Problem:**
  Given a budget $B$ and $n$ items with profits $v_1, \ldots, v_n$ and weights $w_1, \ldots, w_n$, maximize $\sum_{i \in I} v_i$ such that $\sum_{i \in I} w_i \leq B$

- **Dynamic version:** Items are inserted and deleted

- **Our result:** Maintain a $(1 + \epsilon)$-approximation with update time $O(\epsilon^{-2} \log^2(nW))$
  - Improves upon Eberle et al. (2021) who obtained update time $O(\epsilon^{-9} \log^4(nW))$
  - Can you improve this result?

Picture from Wikipedia
https://commons.wikimedia.org/wiki/File:Knapsack.svg
Further Results

• Two tricks to make rows of DPs monotone

• **Lower bounds** for Dynamic $k$-Balanced Graph Partitioning and Dynamic Knapsack showing that if an algorithm only stores a single solution then update time is high
  
  • Suggests that DP-style implicit solutions are inevitable

• For **simultaneous source location**: First near-linear time static algorithm and first dynamic algorithm with subpolynomial update time

• First dynamic algorithm for $\ell_\infty$-Necklace with additive approximation $\pm \varepsilon$ and update time $O(\varepsilon^{-2})$
How do we get these results?
Store the DP Table’s Monotone Rows using Piecewise Constant Functions
What is a Piecewise Constant Function?

• A piecewise constant function \( f: [0, t] \rightarrow [1, W] \) looks like this:

\[
\begin{align*}
  f(x) &= \begin{cases}
    y_1 & \text{if } x_1 \leq x < x_2 \\
    y_2 & \text{if } x_2 \leq x < x_3 \\
    \vdots \\
    y_p & \text{if } x_p \leq x < x_{p+1}
  \end{cases}
\end{align*}
\]

• The set of tuples \((x_1, y_1), \ldots, (x_p, y_p)\) encodes the \(x\)-coordinates and \(y\)-coordinates

• New complexity measure: number of pieces \(p\) of the function

• Interpretation: For each row \(i\) of the DP table \(T\), we will have a function \(f_i\) such that \(f_i(j) = T_{i,j}\)
  
  ➡ Looking at the function \(f_i\) reveals the entire \(i\)'th row
Efficient Operations

- Let $g, h: [0, t] \rightarrow [0, W]$ be piecewise constant functions with at most $p$ pieces. Then we can compute:
  - $f_{\min}(x) := \min\{g(x), h(x)\}$ in time $\tilde{O}(p)$ and at most $2p$ pieces
  - $f_{\text{add}}(x) := g(x) + h(x)$ in time $\tilde{O}(p)$ and at most $2p$ pieces
    - Running times are fast if $p$ is small, no dependency on the size of the domain $[0, t]$!
  - The $(\max, +)$-convolution $f = g \oplus h$ of $g$ and $h$ in time $\tilde{O}(p^2)$, i.e.,
    $$f(x) = \max_{x' \in [0,x]} g(x') + h(x - x')$$
    - But this function might have $\Theta(p^2)$ pieces, which might cause high running times if we use it multiple times.
Ensuring Few Pieces

- How do we ensure that the number of pieces stays small?
- We round \( f(x) \) to powers of \( 1 + \delta \): 
  \[
  [f(x)]_{1+\delta} = \min\{(1 + \delta)^i: (1 + \delta)^i \geq f(x), i \in \mathbb{N}\}
  \]
  - If \( f(x) \in [1, W] \) for all \( x \), then \([f(x)]_{1+\delta}\) only takes \( O(\log_{1+\delta}(W)) \) values
  - If \( f \) is monotone, we have \( \leq 1 \) piece for each value and thus \([f]_{1+\delta}\) has at most \( O(\log_{1+\delta}(W)) \) pieces
- We can perform all operations from before in time \( \log^{O(1)}_{1+\delta}(W) \)
Why is this Useful in DPs?

- We store the rows $i$ of the DP table as piecewise constant functions $f_i$
- We compute entire rows in polylogarithmic time, using operations for our piecewise constant functions (rather than computing them entry-by-entry)
- Recall our assumption: DP has $n$ monotone rows and dependency tree of height $O(\log n)$ and rows are “easy to compute”
  - “easy to compute”: to compute a row, we use only $O(1)$ operations of type $\min\{g, h\}$ and $g + h$ and at most one $(\max, +)$-convolution
- After computing a row, we perform a rounding step $[f]_{1+\delta}$
  - Bounds the number of pieces, allows us to compute each row in time $\tilde{O}(1)$
  - Since the dependency tree has small height, error does not compound too much
Often DPs Have Monotone Rows

- For many optimization problems, the columns correspond to budget constraints (e.g., Knapsack)
  - $k$’th column = “Maximum objective function with budget at most $k$”
  - Monotone rows appear automatically
- Sometimes we have exact budget constraints (“with budget exactly $k$”)
  - Often the DP can be adapted such that it works in the “budget at most $k$”-setting
  - In the paper, we do this for $k$-Balanced Graph Partitioning
- Sometimes other tricks can help
  - For simultaneous source location, a DP by Andreev et al. did not fit our framework (e.g., used negative values)
  - In the paper, we consider the “inverse” of this DP — takes only positive values, fits into our framework
Conclusion
Making Dynamic Programming Dynamic
Monika Henzinger, Stefan Neumann (@StefanResearch), Harald Räcke, Stefan Schmid (@schmiste_ch)

• We provide a general framework such that if
  a DP has monotone rows, the dependency tree is of small height and rows are “easy to compute”,
then we can compute a \((1 + \varepsilon)\)-approximate solution in near-linear time and dynamically with polylog update times

• First near-linear time and dynamic algorithms for \(k\)-Balanced GraphPartitioning
• Fastest fully dynamic algorithm for Knapsack
  • Can you improve it?
• We believe there will be many applications in the future

\[
\begin{array}{cccccc}
0 & 4 & 5 & 10 & 17 \\
0 & 0 & 14 & 29 & 60 \\
0 & 1 & 1 & 2 & 5 \\
0 & 16 & 29 & 29 & 29 \\
0 & 8 & 12 & 20 & 22 \\
\end{array}
\]
Appendix
Our main technical contribution in the paper is for $k$-Balanced Graph Partitioning. But the result for fully dynamic Knapsack is very illustrative for our approach.
Fully Dynamic Knapsack

• **Problem:**
  Given a budget $B$ and $n$ items with profit $v_1, \ldots, v_n$ and weights $w_1, \ldots, w_n$, maximize $\sum_{i \in I} v_i$ such that $\sum_{i \in I} w_i \leq B$

• **Dynamic version:** Items are inserted and deleted

• **Our result:** Maintain a $(1 + \varepsilon)$-approximation with update time $O(\varepsilon^{-2} \log^2(nW))$
  - Improves upon Eberle et al. (2021) who obtained update time $O(\varepsilon^{-9} \log^4(nW))$
Warm-Up: Existing Algorithm

• How can we solve Knapsack using piecewise constant functions?

• Consider an item $i$ with profit $v_i$ and weight $w_i$

• Set $f_{\{i\}}(w) =$ maximum profit if we can spend weight $w$ and can only use items in the set $\{i\}$
Two Items

- Suppose now we have two items $i$ and $j$ such that $v_i \leq v_j$ and $w_i \leq w_j$.

- Set $f_{\{i,j\}}(w) = \text{maximum profit if we can spend weight } w \text{ and can only use items in the set } \{i,j\}$.

- Then $f_{\{i,j\}}(w)$ looks like this:
How to Compute $f_{i,j}(w)$?

- Observe that $f_{i,j}(w)$ can be computed via a $(\max, +)$-convolution:

$$f_{i,j}(w) = \max_{w_i \in [0, w]} \left( f_{i,i}(w_i) + f_{j,j}(w - w_i) \right)$$
General Case of More Than Two Items

• More generally, set \( f_J(w) = \) maximum profit if we can spend weight \( w \) and can only use items in the set \( J \)

  • \( f_{[n]}(B) \) is the optimal solution for the global problem

• If \( J = J_1 \cup J_2 \) then \( f_J(w) = \max_{0 \leq w' \leq w} f_{J_1}(w') + f_{J_2}(w - w') \)

• Algorithm:
  • Compute the DP bottom-up
  • In each internal node, we compute \( f_J \) as \( (\max, +) \)-convolution of \( f_{J_1} \) and \( f_{J_2} \)
  • Then we set \( f_J = \lceil f_J \rceil_{1+\delta} \)
Analysis

• Algorithm:
  - In each internal node we compute $f_j$ as $(\max, +)$-convolution of $f_{j_1}$ and $f_{j_2}$
  - Then we set $f_j = \lceil f_j \rceil_{1+\delta}$

• Approximation ratio:
  - At each level, we lose approximation factor $1 + \delta$ because of rounding
  - Since the dependency tree has height $O(\log n)$, our approximation ratio is $(1 + \delta)^{O(\log n)} \leq \exp(\delta \cdot O(\log n)) \leq 1 + \epsilon$ for $\delta = \log(1 + \epsilon)/O(\log n)$

• Running time:
  - Each function has at most $O(\log_{1+\delta}(W))$ pieces, thus convolution takes time $O(\log_{1+\delta}(W))$
  - Using $\delta$ as above, total time is $O(n \cdot \epsilon^{-2} \log^2(W) \log^2(n))$
Dynamic Knapsack

- **Dynamic version:**
  - Suppose we can change item profits and weights
  - \( \text{Update}(i, v, w): \) set \( v_i = v \) and \( w_i = w \)
  - After update for item \( i \), recomputate leaf–root path from node \( i \) to root
  - Takes update time \( O(\epsilon^{-2} \log^2(W) \log^3(n)) \)
  - But we can be even faster: update time \( O(\epsilon^{-2} \log^2(nW)) \)
Faster Dynamic Knapsack

- Partition the items into weight classes $V_\ell = \{i: (1 + \epsilon)\ell \leq v_i < (1 + \epsilon)^{\ell+1}\}$
- Set $V_{\ell/\epsilon}$ to the $1/\epsilon$ items from $V_\ell$ of smallest weight, $V'_\ell = V_\ell \setminus V_{\ell/\epsilon}$
- Consider the $1/\epsilon$ items $X = \bigcup_{\ell \geq 0} V_{\ell/\epsilon}$ of smallest weight from each class, and the other items $Y = \bigcup_{\ell \geq 0} V'_\ell$
- Note that $|X| = \ell \cdot 1/\epsilon = O(\epsilon^{-2} \log(W))$
  - Maintain $X$ using our data structure from before, since $|X|$ is small, update time is $O(\epsilon^{-2} \log^2(nW))$
- Maintain $Y$ in binary search trees, sorted by density $v_i/w_i$
Answering Queries

• We maintain $X$ using our data structure from before
  • Solution is stored as a piecewise constant function with pieces $(x_1, y_1), \ldots, (x_p, y_p)$
  • Every time a new piece starts, objective function value increases

• **Returning a solution.** For each $i = 1, \ldots, p$ do:
  • Spend budget $x_i$ on solution from $X$ and budget $B - x_i$ on solution from $Y$ using fractional knapsack
  • Fractional knapsack solution can be queried from binary search trees
    • In the analysis, we prove that removing the item which is fractionally cut in the fractional knapsack solution is not a problem

\[ X(x_1, y_1), \ldots, (x_p, y_p) \]