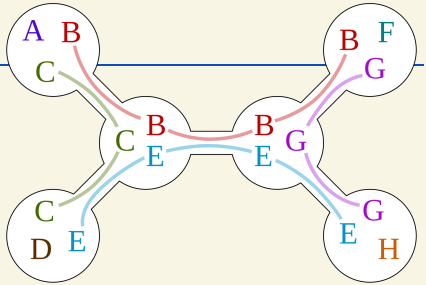
Introduction (cont'd)

DM898: Parameterized Algorithms
Lars Rohwedder



Today's lecture

- Recap of dynamic programming
- FPT algorithms via dynamic programming
- Limits of parameterized algorithms

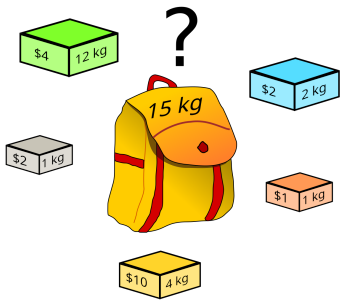
Dynamic programming: 3 views

Dynamic programs can be stated in different (mostly equivalent) ways. We use the Knapsack problem as a simple example.

Knapsack problem

Input: Items $i = 1, 2, \ldots, n$ with profit $p_i \in \mathbb{Z}_{\geq 0}$ and weight $w_i \in \mathbb{Z}_{\geq 0}$; a capacity $C \in \mathbb{Z}_{\geq 0}$.

Output: Subset of items $S \subseteq \{1, 2, \ldots, n\}$ such that $\sum_{i \in C} w_i \leq C$ and the profit $\sum_{i \in S} p_i$ is maximized.



source: https://commons.wikimedia.org/wiki/File:Knapsack.svg

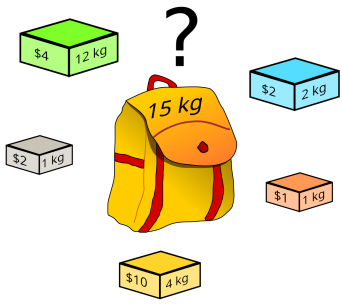
Dynamic programs can be stated in different (mostly equivalent) ways. We use the Knapsack problem as a simple example.

Knapsack problem

Input: Items $i = 1, 2, \ldots, n$ with profit $p_i \in \mathbb{Z}_{\geq 0}$ and weight $w_i \in \mathbb{Z}_{\geq 0}$; a capacity $C \in \mathbb{Z}_{\geq 0}$.

Output: Subset of items $S \subseteq \{1, 2, \ldots, n\}$ such that $\sum_{i \in C} w_i \leq C$ and the profit $\sum_{i \in S} p_i$ is maximized.

- Knapsack is NP-hard, but it can be solved in time polynomial in n and C via dynamic programming
- C can be exponential in the input size, since C can be encoded with $O(\log C)$ bits
- We call such running time **pseudo-polynomial**, which can be seen as a form of FPT running time



source: https://commons.wikimedia.org/wiki/File:Knapsack.svg

View 1: Branching with memory

Idea: branch on whether item n is in solution.

- If n is in solution, then remaining solution should be chosen optimal for capacity $C - w_n$
- If n is not in solution, it can just be ignored

KS($n, (p_i)_{i=1}^n, (w_i)_{i=1}^n, C$)

If $n = 0$

- **return 0;**

else if $w_n > C$

- **return KS(**$n-1, (p_i)_{i=1}^{n-1}, (w_i)_{i=1}^{n-1}, C$**)**

else

- $S1 \leftarrow p_n +$ **KS(**$n-1, (p_i)_{i=1}^{n-1}, (w_i)_{i=1}^{n-1}, C - w_n$**)**
- $S2 \leftarrow$ **KS(**$n-1, (p_i)_{i=1}^{n-1}, (w_i)_{i=1}^{n-1}, C$**)**
- **return** $\max\{S1, S2\}$

View 1: Branching with memory

Idea: branch on whether item n is in solution.

- If n is in solution, then remaining solution should be chosen optimal for capacity $C - w_n$
- If n is not in solution, it can just be ignored
- **Problem:** running time exponential in n

KS($n, (p_i)_{i=1}^n, (w_i)_{i=1}^n, C$)

If $n = 0$

- **return 0;**

else if $w_n > C$

- **return KS(**$n-1, (p_i)_{i=1}^{n-1}, (w_i)_{i=1}^{n-1}, C$**)**

else

- $S1 \leftarrow p_n +$ **KS(**$n-1, (p_i)_{i=1}^{n-1}, (w_i)_{i=1}^{n-1}, C - w_n$**)**
- $S2 \leftarrow$ **KS(**$n-1, (p_i)_{i=1}^{n-1}, (w_i)_{i=1}^{n-1}, C$**)**
- **return** $\max\{S1, S2\}$

View 1: Branching with memory

Idea: branch on whether item n is in solution.

- If n is in solution, then remaining solution should be chosen optimal for capacity $C - w_n$
- If n is not in solution, it can just be ignored
- **Problem:** running time exponential in n
- **Solution:** remember solved subproblems in a cache, which we think of as a hash map (details omitted)

KS($n, (p_i)_{i=1}^n, (w_i)_{i=1}^n, C$)

If cache[$n, (p_i)_{i=1}^n, (w_i)_{i=1}^n, C$**]** $= \perp$ // entry not found

- **If** $n = 0$
  - cache[$n, (p_i)_{i=1}^n, (w_i)_{i=1}^n, C$] $\leftarrow 0$;
- **else if** $w_n > C$
  - cache[$n, (p_i)_{i=1}^n, (w_i)_{i=1}^n, C$] $\leftarrow$ KS($n-1, (p_i)_{i=1}^{n-1}, (w_i)_{i=1}^{n-1}, C$)
- **else**
  - $S1 \leftarrow p_n +$ KS($n-1, (p_i)_{i=1}^{n-1}, (w_i)_{i=1}^{n-1}, C - w_n$)
  - $S2 \leftarrow$ KS($n-1, (p_i)_{i=1}^{n-1}, (w_i)_{i=1}^{n-1}, C$)
  - cache[$n, (p_i)_{i=1}^n, (w_i)_{i=1}^n, C$] $\leftarrow \max\{S1, S2\}$

return cache[$n, (p_i)_{i=1}^n, (w_i)_{i=1}^n, C$**]**

View 1: Branching with memory

Idea: branch on whether item n is in solution.

- If n is in solution, then remaining solution should be chosen optimal for capacity $C - w_n$
- If n is not in solution, it can just be ignored
- **Problem:** running time exponential in n
- **Solution:** remember solved subproblems in a cache, which we think of as a hash map (details omitted)

KS($n, (p_i)_{i=1}^n, (w_i)_{i=1}^n, C$)

If cache[$n, (p_i)_{i=1}^n, (w_i)_{i=1}^n, C$**]** $= \perp$ // entry not found

- **If** $n = 0$
  - cache[$n, (p_i)_{i=1}^n, (w_i)_{i=1}^n, C$] $\leftarrow 0$;
- **else if** $w_n > C$
  - cache[$n, (p_i)_{i=1}^n, (w_i)_{i=1}^n, C$] $\leftarrow$ KS($n-1, (p_i)_{i=1}^{n-1}, (w_i)_{i=1}^{n-1}, C$)
- **else**
  - $S1 \leftarrow p_n +$ KS($n-1, (p_i)_{i=1}^{n-1}, (w_i)_{i=1}^{n-1}, C - w_n$)
  - $S2 \leftarrow$ KS($n-1, (p_i)_{i=1}^{n-1}, (w_i)_{i=1}^{n-1}, C$)
  - cache[$n, (p_i)_{i=1}^n, (w_i)_{i=1}^n, C$] $\leftarrow \max\{S1, S2\}$

return cache[$n, (p_i)_{i=1}^n, (w_i)_{i=1}^n, C$**]**

Running time analysis

We count the number of subproblems that might appear throughout the execution.

Number of subproblems: every subproblem has the form $(n', (p_i)_i^{n'}, (w_i)_i^{n'}, C')$ for some $n' \leq n$, $C' \leq C$

$\rightsquigarrow nC$ **subproblems**

Running time per subproblem: $O(1)$ (if hash operations are implemented in constant time)

Correctness analysis

Lemma

Let $\mathrm{OPT}(n, (p_i)_{i=1}^n, (w_i)_{i=1}^n, C)$ be the value of the optimal solution for the given input. Then:

- for $n = 0$,
$$\mathrm{OPT}(n, (p_i)_{i=1}^n, (w_i)_{i=1}^n, C) = 0$$
- for $n \geq 1$ and $w_n > C$,
$$\mathrm{OPT}(n, (p_i)_{i=1}^n, (w_i)_{i=1}^n, C) = \mathrm{OPT}(n-1, (p_i)_{i=1}^{n-1}, (w_i)_{i=1}^{n-1}, C)$$
- and for $n \geq 1$ and $w_n \leq C$,
$$\mathrm{OPT}(n, (p_i)_{i=1}^n, (w_i)_{i=1}^n, C) = \max\{p_n + \mathrm{OPT}(n-1, (p_i)_{i=1}^{n-1}, (w_i)_{i=1}^{n-1}, C - w_n), \mathrm{OPT}(n-1, (p_i)_{i=1}^{n-1}, (w_i)_{i=1}^{n-1}, C)\}$$

Proof omitted.

Given this lemma, it follows by induction that the output of $\mathrm{KS}(n, (p_i)_{i=1}^n, (w_i)_{i=1}^n, C)$ is equal to $\mathrm{OPT}(n, (p_i)_{i=1}^n, (w_i)_{i=1}^n, C)$.

View 2: Iterative table filling

- A more "clean" way of implementing the same idea is by iteratively filling out a table over all relevant subproblems.
- The most common way of writing a dynamic program for theoreticians
- Running time more obvious here

KS-iter$(n, (p_i)_{i=1}^n, (w_i)_{i=1}^n, C)$

- Initialize table $D[i, W]$ over $i \in \{0, 1, 2, \ldots, n\}$, $W \in \{0, 1, \ldots, C\}$ // Intuitively $D[i, W]$ contains the value of the optimal solution for items $1, \ldots, i$ and capacity W.
- for $W \in \{0, 1, \ldots, C\}$
  - $D[0, W] \leftarrow 0$
- for $i \in \{1, 2, \ldots, n\}$
  - for $W \in \{0, 1, \ldots, w_i - 1\}$
    - $D[i, W] \leftarrow D[i-1, W]$
  - for $W \in \{w_i, \ldots, C\}$
    - $D[i, W] \leftarrow \max\{D[i-1, W], p_i + D[i-1, W - w_i]\}$
- return $D[n, C]$

View 3: Eliminating dominated solutions

- A third way of writing dynamic programs is to iteratively generate sets of solutions $\mathcal{T}$
- In order to limit the number of solutions we eliminate solutions that are dominated (i.e., worse in all aspects) by others

KS-dom$(n, (p_i)_{i=1}^n, (w_i)_{i=1}^n, C)$

- $\mathcal{T} \leftarrow \{(0,0)\}$ // set of (weight, profit) pairs that can be obtained
- for $i \in \{1, 2, \ldots, n\}$:
  - $\mathcal{T} \leftarrow \mathcal{T} \cup \{(W + w_i, P + p_i) \mid (W, P) \in \mathcal{T}, W + w_i \leq C\}$
  - for $(W, P), (W', P') \in \mathcal{T}$ with $P > P', W \leq W'$
    - $\mathcal{T} \leftarrow \mathcal{T} \setminus (W', P')$
- return $\max\{P \mid (W, P) \in \mathcal{T}\}$

View 3: Eliminating dominated solutions

- A third way of writing dynamic programs is to iteratively generate sets of solutions $\mathcal{T}$
- In order to limit the number of solutions we eliminate solutions that are dominated (i.e., worse in all aspects) by others

KS-dom$(n, (p_i)_{i=1}^n, (w_i)_{i=1}^n, C)$

- $\mathcal{T} \leftarrow \{(0,0)\}$ // set of (weight, profit) pairs that can be obtained
- for $i \in \{1, 2, \ldots, n\}$:
  - $\mathcal{T} \leftarrow \mathcal{T} \cup \{(W + w_i, P + p_i) \mid (W, P) \in \mathcal{T}, W + w_i \leq C\}$
  - for $(W, P), (W', P') \in \mathcal{T}$ with $P > P', W \leq W'$
    - $\mathcal{T} \leftarrow \mathcal{T} \setminus (W', P')$
- return $\max\{P \mid (W, P) \in \mathcal{T}\}$

- A running time can be derived from bounding the number of undominated solutions, i.e., $|\mathcal{T}|$:
  At the beginning of each iteration, $\mathcal{T}$ does not contain any dominated pair. Thus, for every $W \leq C$ there can only be one pair $(W, P) \in \mathcal{T}$ for some P. It follows that $|\mathcal{T}| \leq C$. The running time of each iteration is $\mathrm{poly}(|\mathcal{T}|)$ and there are n iterations. $\rightsquigarrow$ **running time** $\mathrm{poly}(n, C)$
- In some cases, this variant can be more efficient in practice, since it does not generate solution for states that are impossible to reach. For example, it is also efficient if C is large, but all p_i's are small: it runs in $\mathrm{poly}(n, \min\{C, \sum_{i=1}^n p_i\})$ time

Steiner tree

Motivation

Case: A new pipeline for natural gas should connect the production site at Kårstø to the existing pipeline network at several locations in Norway, Sweden, and Denmark.

Building pipelines is very expensive. How can we minimize the total length of the new pipeline?

Plan for pipeline "Skanled". Source: Gassco

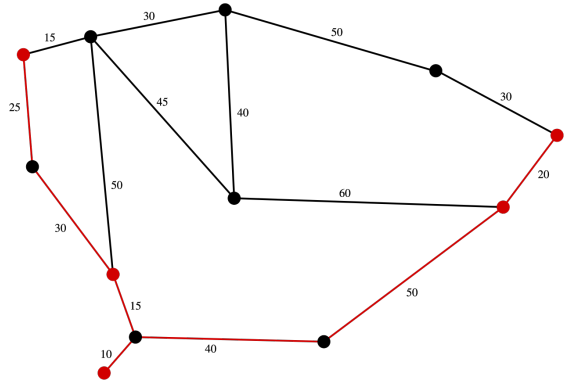
Formal definition

Steiner tree problem

Input: Undirected graph G with edge weights $w : E(G) \to \mathbb{Z}_{\geq 0}$ and terminals $K \subseteq V(G)$

Output: Connected subgraph H with $K \subseteq V(H)$ and minimal weight $w(H) = \sum_{e \in E(H)} w(e)$

- We consider here parameter $k = |K|$
- Minimum Steiner tree $\neq$ minimum spanning tree



K depicted as red dots. Red edges form a solution.

Source: https://commons.wikimedia.org/wiki/File:Steinerbaum_Beispiel_Graph.svg

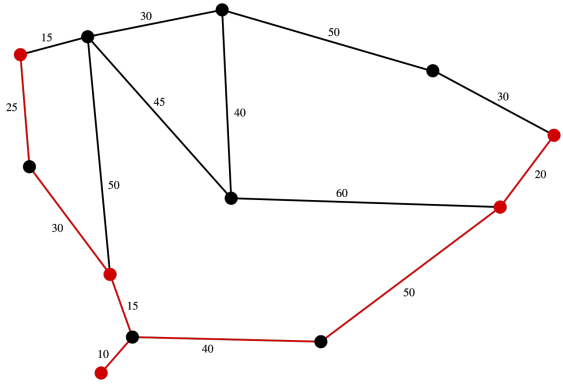
Formal definition

Steiner tree problem

Input: Undirected graph G with edge weights $w : E(G) \to \mathbb{Z}_{\geq 0}$ and terminals $K \subseteq V(G)$

Output: Connected subgraph H with $K \subseteq V(H)$ and minimal weight $w(H) = \sum_{e \in E(H)} w(e)$

- We consider here parameter $k = |K|$
- Minimum Steiner tree $\neq$ minimum spanning tree



K depicted as red dots. Red edges form a solution.

Source: https://commons.wikimedia.org/wiki/File:Steinerbaum_Beispiel_Graph.svg

Simplifications

- There exists an optimal solution, which is a tree
- We can reduce to the case that G is connected, $|K| > 1$ and $\deg(t) = 1$ for all $t \in K$

Dynamic program for Steiner tree

Steiner(G, w, K)

- Assume that G connected, $|K| > 1$, and $\deg(t) = 1$ for all $t \in K$
- Initialize table $D[K', v]$ over all $K' \subseteq K$, $v \in V \setminus K$
  // $D[K', v]$ is the lowest weight among Steiner trees on terminals $K' \cup \{v\}$
- for $t \in K, v \in V \setminus K$
  - $D[\{t\}, v] \leftarrow \mathsf{dist}(t, v)$ // shortest $t - v$ path
- for $K' \subseteq K, v \in V \setminus K$ in non-decreasing order of $|K'|$
  - $D[K', v] = \min\{\mathsf{dist}(v, u) + D[K'', u] + D[K' \setminus K'', u] \mid u \in V \setminus K, \emptyset \subsetneq K'' \subsetneq K'\}$
- return $\min\{D[K, v] \mid v \in V \setminus K\}$

Analysis (correctness and running time): blackboard

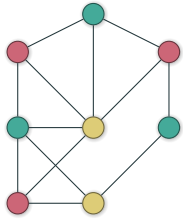
Limits of parameterized algorithms

Graph coloring

Vertex Coloring problem

Given a graph $G = (V, E)$ color the vertices with a minimal number of colors such that for every edge $e = (u, v)$ vertices u and v are colored differently.

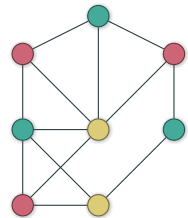
A natural parameter is the minimal number of colors necessary.



Graph coloring

Vertex Coloring problem

Given a graph $G = (V, E)$ color the vertices with a minimal number of colors such that for every edge $e = (u, v)$ vertices u and v are colored differently.



A natural parameter is the minimal number of colors necessary.

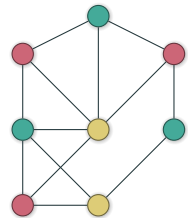
It is NP-hard to decide whether a graph is colorable with ≤ 3 colors. Consequences:

- Can there be an XP algorithm ($n^{f(k)}$), where k is the optimal number of colors?
- Can there be an FPT algorithm ($f(k) \cdot n^{O(1)}$), where k is the optimal number of colors?

Graph coloring

Vertex Coloring problem

Given a graph $G = (V, E)$ color the vertices with a minimal number of colors such that for every edge $e = (u, v)$ vertices u and v are colored differently.



A natural parameter is the minimal number of colors necessary.

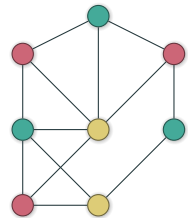
It is NP-hard to decide whether a graph is colorable with ≤ 3 colors. Consequences:

- Can there be an XP algorithm ($n^{f(k)}$), where k is the optimal number of colors? **No, unless P=NP**
- Can there be an FPT algorithm ($f(k) \cdot n^{O(1)}$), where k is the optimal number of colors?

Graph coloring

Vertex Coloring problem

Given a graph $G = (V, E)$ color the vertices with a minimal number of colors such that for every edge $e = (u, v)$ vertices u and v are colored differently.



A natural parameter is the minimal number of colors necessary.

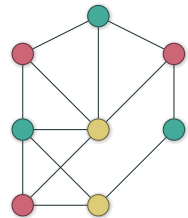
It is NP-hard to decide whether a graph is colorable with ≤ 3 colors. Consequences:

- Can there be an XP algorithm ($n^{f(k)}$), where k is the optimal number of colors? **No, unless P=NP**
- Can there be an FPT algorithm ($f(k) \cdot n^{O(1)}$), where k is the optimal number of colors? **No, unless P=NP**

Graph coloring

Vertex Coloring problem

Given a graph $G = (V, E)$ color the vertices with a minimal number of colors such that for every edge $e = (u, v)$ vertices u and v are colored differently.



A natural parameter is the minimal number of colors necessary.

It is NP-hard to decide whether a graph is colorable with ≤ 3 colors. Consequences:

- Can there be an XP algorithm ($n^{f(k)}$), where k is the optimal number of colors? **No, unless P=NP**
- Can there be an FPT algorithm ($f(k) \cdot n^{O(1)}$), where k is the optimal number of colors? **No, unless P=NP**

There are also problems for which there are XP algorithms and evidence that there is **no** FPT algorithm. This requires a complexity theory beyond P and NP, which we will explore later in the course.

Summary and outlook

We briefly touched three fundamental algorithmic tools: **preprocessing**, **branching**, and **dynamic programming**

These will be the pillars of the topics of this course:

- Kernelization (theory of preprocessing)
- Advanced branching
- Branch-and-Bound and integer programming
- Integer programming techniques for FPT
- Parameterized complexity theory
- Tree decompositions
- Randomized methods in FPT
- Mildly exponential algorithms
- Fine-grained complexity