Optimization for Machine Learning

Discrete Optimization

December 9, 2021

TC2 - Optimisation

Université Paris-Saclay

Anne Auger and Dimo Brockhoff

Inria Saclay – Ile-de-France

Course Overview

Date		Topic
Thu, 4.11.2021	DB	Introduction
Thu, 11.11.2021		no lecture
Thu, 18.11.2021	AA	Continuous Optimization I: differentiability, gradients, convexity, optimality conditions
Thu, 25.11.2021	AA	Continuous Optimization II: constrained optimization, gradient-based algorithms, stochastic gradient [written test / « contrôle continue »]
Thu, 2.12.2021	AA	Continuous Optimization III: stochastic algorithms, derivative-free optimization
Thu, 9.12.2021	DB	Discrete Optimization: greedy algorithms, branch&bound, dynamic programming
Thu 16.12.2021	DB	Written exam
		! always 13h30 till 16h00

Concrete Information About Exam

Written exam

- multiple choice, typically 4 answers each (1-4 answers correct)
- closed book (nothing allowed but pen) → easier questions ☺
- next Thursday (Dec. 16) @ 1:30pm
- 1.5 hours

- Back to some examples of optimization problems in Machine Learning …

Supervised Learning

- Classification
  - Is there a cat on the picture?

Yes / No

Supervised Learning

- Classification

  - Is there a cat on the picture?

Yes

Supervised Learning

- Classification
  - Is there a cat on the picture?

Yes

- Classification
  - Is there a cat on the picture?

No

Supervised Learning

- Labelled data / training sets

x^1

$y^1 = 1$

x^2

$y^2 = 1$

x^3

$y^3 = -1$

Input or features

Output labels Target

Given a set of examples $\{(x^1, y^1), \ldots, (x^n, y^n)\}$ with x^i the features and y^i labels/targets

Supervised Learning

Given a set of examples $\{(x^1, y^1), \ldots, (x^n, y^n)\}$ with x^i the features and y^i labels/targets

Find a mapping $h : x \in X \rightarrow y \in \mathbb{R}$ that will assign the "correct" target to each input

Learning algorithm

New image (not in the training set)

$h\Big($  $\Big) = -1$

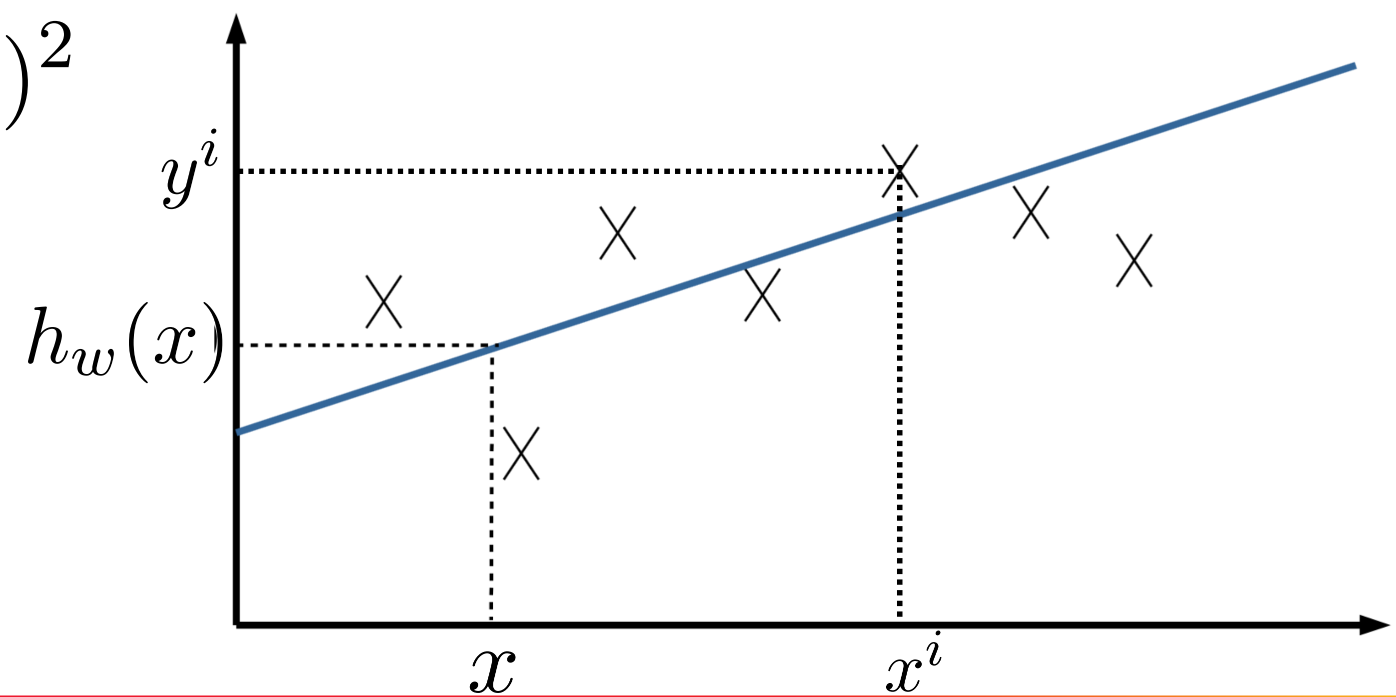
Example 1: Linear Regression

Hypothesis: linear model

$$h_w(x) = w_0 + w_1 x_1 + \ldots + w_{d-1} x_{d-1} \overset{x_0 = 1}{=} \langle w, x \rangle$$

Find $h_w(x)$ via solving the minimization problem

$$\min_{w \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} (h_w(x^i) - y^i)^2$$



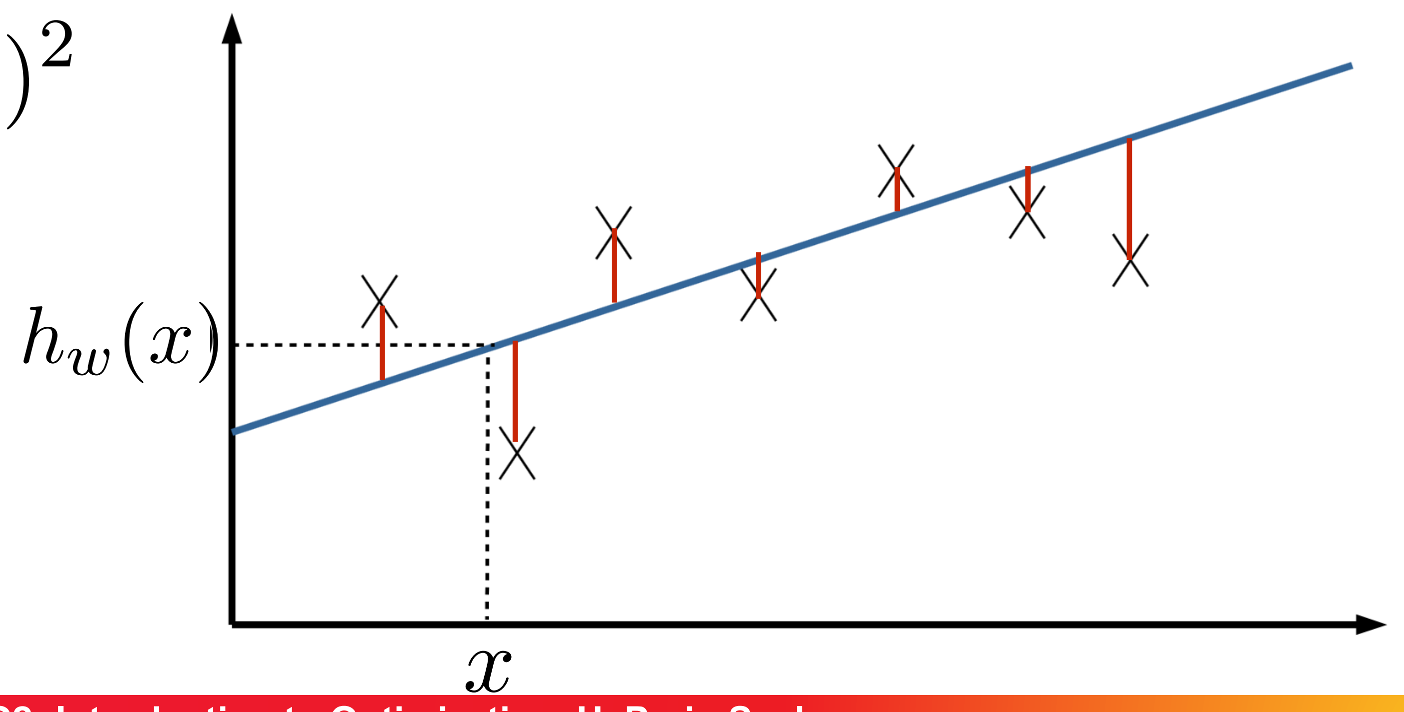
Example 1: Linear Regression

Hypothesis: linear model

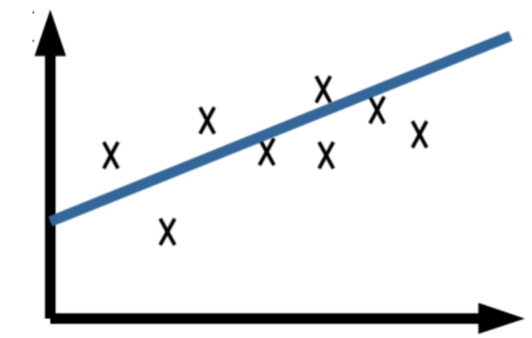
$$h_w(x) = w_0 + w_1 x_1 + \ldots + w_{d-1} x_{d-1} \overset{x_0 = 1}{=} \langle w, x \rangle$$

Find $h_w(x)$ via solving the minimization problem

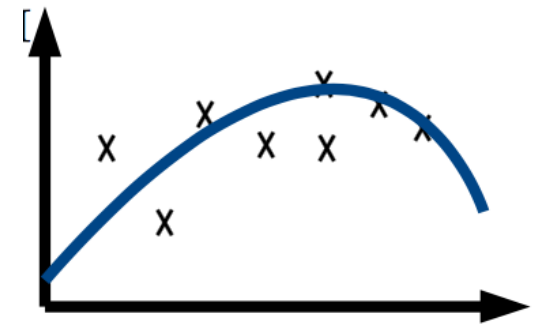
$$\min_{w \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \left(h_w(x^i) - y^i \right)^2$$



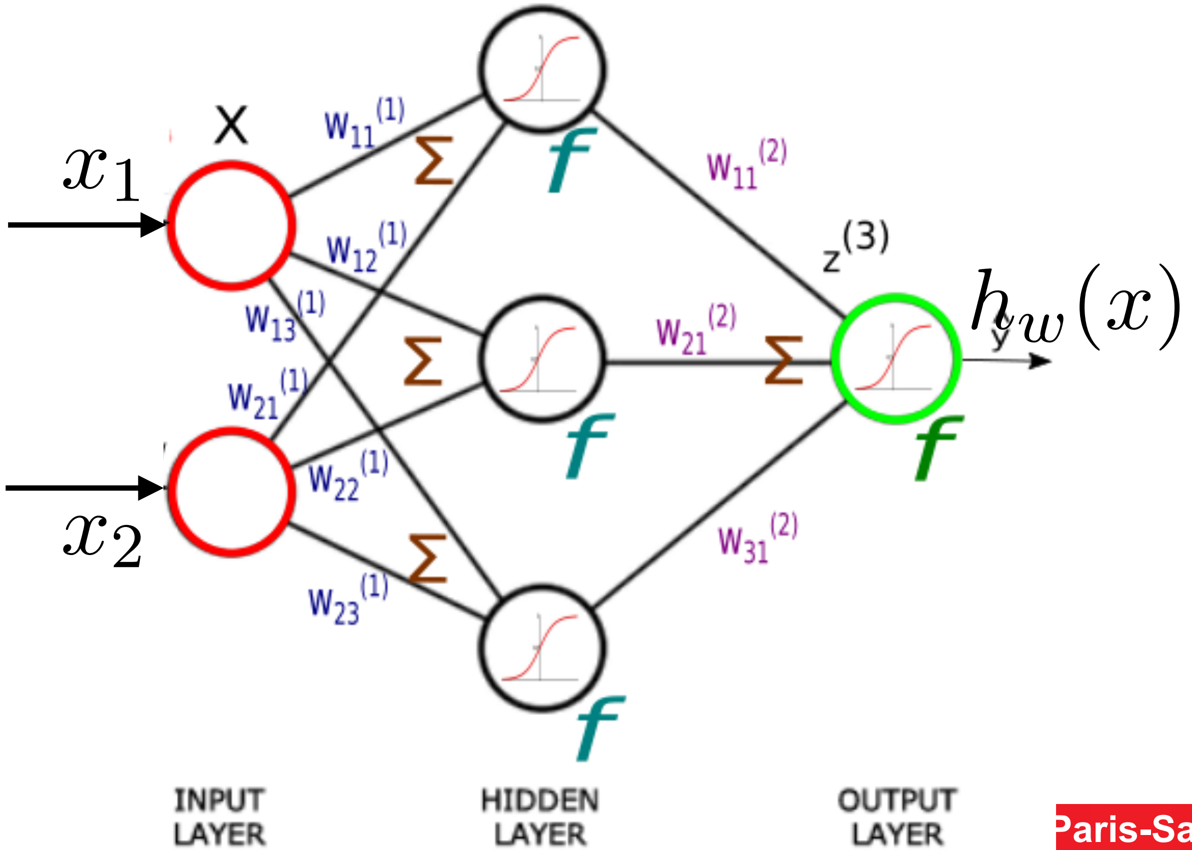
Linear: $$h_w(x) = \langle w, x \rangle = \sum_{i=0}^{d-1} w_i x_i$$



Polynomial: $$h_w(x) = \sum_{i,j=0}^{d-1} w_{i,j} x_i x_j$$



Neural network:



Generalization: Different Loss Functions

Start from the linear regression problem:

$$\min_{w \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} (h_w(x^i) - y^i)^2$$

Let $y_h := h_w(x)$

Loss function: $l : \mathbb{R} \times \mathbb{R} \to \mathbb{R}_+$

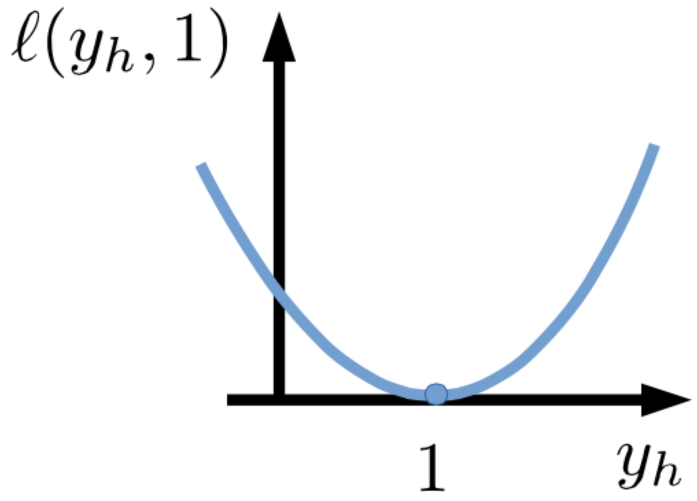
$$(y_h, y) \to l(y_h, y)$$

For linear regression
$l(y_h, y) = (y_h - y)^2$

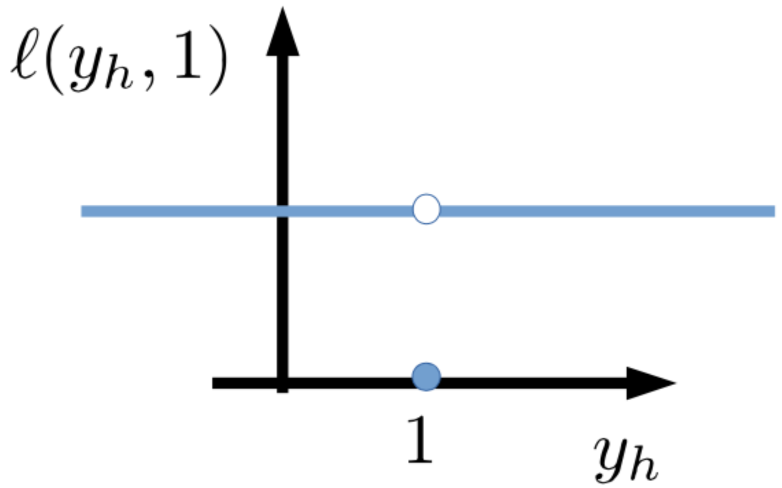
Training (optimization) problem:

$$\min_{w \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} l(h_w(x^i), y^i)$$

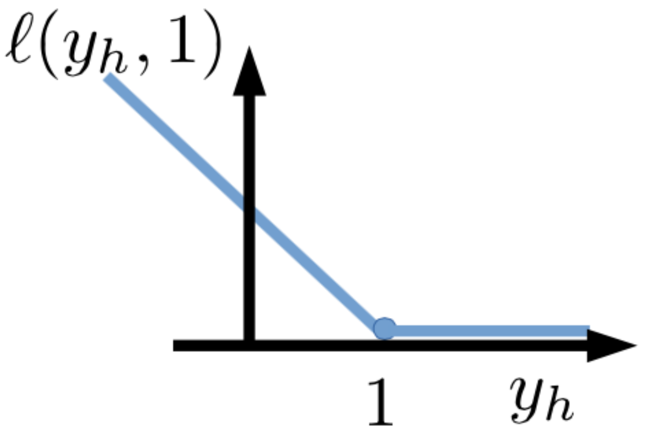
Quadratic loss: $l(y_h, y) = (y_h - y)^2$



Binary loss: $l(y_h, y) = \begin{cases} 0 \text{ if } y_h = y \\ 1 \text{ if } y_h \neq y \end{cases}$



Hinge loss: $l(y_h, y) = \max\{0, 1 - y_h y\}$



Numerical Optimization

Very often it is not possible to solve analytically the equation $\nabla f(x) = 0$ and we have to resort to an iterative algorithm (or numerical optimization algorithm) that will generate a sequence of points $\{x_k : k \geq 0\}$ that should converge to $\mathrm{argmin}_x f(x)$

Optimization algorithm:

input $f, \nabla f, (\nabla^2 f)$

initialize $k = 0, x_0$ [other state variables]

while not happy do

$\quad$ $\text{update } x_k$

$\quad$ $k = k + 1$

$f(x_{k+1}) \leq f(x_k)$ (typically)

end-do

return x_k, k

Goal:

$\lim_{k \to \infty} f(x_k) = \min_x f(x)$

$\lim_{k \to \infty} \|x_k - x^*\| = 0$

Algorithm Classes

Depending on the information the algorithm is using to create a new point (or iterate) we distinguish

Zero-order's algorithms: only use f (no gradients, ...). Those methods are also called derivative-free optimization algorithms. Used when gradient or Hessian are difficult to compute, or when the functions are not differentiable.

First-order algorithms: use f and ∇f. Standard algorithms when f is differentiable, convex.

Second-order algorithms: use $f, \nabla f$ and $\nabla^2 f$. When we can have an "easy" access to the Hessian matrix.

descent direction

Illustration idea from "Alexander & Michael Bronstein" Numerical Optimization slides

Descent Algorithm

Generic algorithm:

choose an initial point x_0, $k = 0$

while not happy

choose a descent direction d_k

line-search: choose a step-size σ_k

$$x_{k+1} = x_k + \sigma_k d_k$$

$$k = k + 1$$

Line search: 1-d minimization along the descent direction

$$\sigma \to f(x_k + \sigma d_k)$$

Descent direction: direction such that for σ small enough

$$f(x_k + \sigma d_k) < f(x_k)$$

Stopping criteria

When are we "happy", i.e. when do we stop the algorithm?

- when gradient norm becomes small

$$\|\nabla f(x_k)\| \leq \epsilon$$

- when step-size becomes small

$$\|x_{k+1} - x_k\| \leq \epsilon$$

- when progress in f becomes small

$$\frac{|f(x_{k+1}) - f(x_k)|}{|f(x_k)|} \leq \epsilon$$

Newton's Algorithm

Take as descent direction the Newton step:

$$d_k = -[\nabla^2 f(x_k)]^{-1} \nabla f(x_k)$$

The Newton's direction minimizes the best locally quadratic approximation of f. Indeed, by Taylor's expansion we can approximate f locally in x by

$$\begin{aligned} g(h) &= f(x) + \nabla f(x)^\top h + \tfrac{1}{2} h^\top \nabla^2 f(x) h \\ &\approx f(x+h) \end{aligned}$$

Minimizing g with respect to h yields:

$$h = -[\nabla^2 f(x)]^{-1} \nabla f(x)$$

Quasi-Newton's Methods

In quasi-Newton's methods, the Newton direction is approximated by using solely first order information (gradient)

Key idea: successive iterates x_k, x_{k+1} and gradients $\nabla f(x_k)$ yield second order information

$$q_k \approx \nabla^2 f(x_{k+1}) p_k$$

$$p_k = x_{k+1} - x_k, \; q_k = \nabla f(x_{k+1}) - \nabla f(x_k)$$

Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm

BFGS algorithm:

B_k approximation of Hessian matrix

$$
\begin{aligned}
d_k &= -B_k^{-1}\nabla f(x_k) \\
x_{k+1} &= x_k + \sigma_k d_k \text{ (find } \sigma_k \text{ via line-search)} \\
y_k &= \nabla f(x_{k+1}) - \nabla f(x_k) \\
B_{k+1} &= B_k + \frac{y_k y_k^\top}{y_k^\top \sigma_k d_k} - \frac{B_k d_k d_k^\top B_k}{d_k^\top B_k d_k}
\end{aligned}
$$

efficient update to compute the inverse of B_k

Considered as the state-of-the-art quasi-Newton's algorithm.
Implemented in all (good) optimization toolboxes

Gradient Descent - Simple Theoretical Analysis

Theorem[Linear convergence of gradient descent] Assume $f : \mathbb{R}^d \to \mathbb{R}$ is twice continuously differentiable, convex and for all x, $\mu I_d \preccurlyeq \nabla^2 f(x) \preccurlyeq L I_d$ with $\mu > 0$. Let x^* be the unique global minimum of f. The gradient descent algorithm with fixed step-size $\sigma_t = \frac{1}{L}$ satisfies

$$\|x_{k+1} - x^*\|^2 \leq \left(1 - \frac{\mu}{L}\right) \|x_k - x^*\|^2 \ .$$

That is the algorithm converges geometrically (also called linearly):

$$\|x_k - x^*\|^2 \leq \left(1 - \frac{\mu}{L}\right)^k \|x_0 - x^*\|^2$$

algorithm slower and slower with increasing condition number

In comparison, convergence of Newton's method is quadratic:

$$\|x_{k+1} - x^*\| \leq c \, \|x_k - x^*\|^2 \text{ with } c < 1$$

$$\|x_{k+1} - x^*\|^2 \leq c^2 \, \left(\|x_k - x^*\|^2\right)^2 \text{ with } c < 1$$

Stochastic Gradient - Motivation

We now come back to our training optimization problem

$$\min_{w \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \underbrace{l(h_w(x^i), y^i)}_{f_i(w)}$$

the f_i can include a regularization term

Gradient descent update:

$$w_{k+1} = w_k - \sigma_k \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(w_k)$$

Problem: each iteration requires to compute a gradient $\nabla f_i(w)$ for each data point. We don't want to do that when n is large (quite typical).

Stochastic Gradient

The gradient of $f(w) = \frac{1}{n}\sum_{i=1}^{n} f_i(w)$ is approximated by the gradient of a single data function $f_i(w)$ at each iteration

$$\nabla f(w) \approx \nabla f_i(w) \text{ for } j \text{ chosen at random}$$

Stochastic gradient descent update:

$$\text{sample } j \in \{1, \ldots, n\}$$
$$w_{k+1} = w_k - \sigma_k \nabla f_i(w_k)$$

Course Overview

Date		Topic
Thu, 4.11.2021	DB	Introduction
Thu, 11.11.2021		no lecture
Thu, 18.11.2021	AA	Continuous Optimization I: differentiability, gradients, convexity, optimality conditions
Thu, 25.11.2021	AA	Continuous Optimization II: constrained optimization, gradient-based algorithms, stochastic gradient [written test / « contrôle continue »]
Thu, 2.12.2021	AA	Continuous Optimization III: stochastic algorithms, derivative-free optimization
Thu, 9.12.2021	DB	Discrete Optimization: greedy algorithms, branch&bound, dynamic programming
Thu 16.12.2021	DB	Written exam
		! always 13h30 till 16h00

Discrete Optimization

Integer Programming

- variables are integers
- simplest example: optimization in $\{0, 1\}^n$

ML example:

hyperparameter tuning with algorithm parts being present ($x_i = 1$) or not ($x_i = 0$)

Combinatorial Optimization

- Search space not necessarily anymore a subset of $\mathbb{R}^n$
- for example, optimization on graphs

ML example:

structure optimization of neural networks

Discrete vs. Continuous Optimization

Important Differences/Observations

- finite search space → still: enumeration impracticable
- discrete neighborhood, sometimes not even clear how to define
- gradient inexistent → follow locally best neighbor?
- different neighborhoods, different definition of local optimum!

example later

- partial evaluations common for discrete problems
- blackbox vs. greybox vs. whitebox

…meaning that solvers for discrete problems are typically more specialized

Overview Discrete Optimization

Algorithms for discrete problems:

- often highly problem-specific
- but some general concepts are repeatedly used:
  - greedy algorithms
  - branch and bound
  - dynamic programming
  - randomized search heuristics [not in this lecture]

Motivation for this Last Part of the Lecture:

- get an idea of the most common algorithm design principles
- we cannot
  - go into details and present many examples of algorithms

    …but for a few
  - analyze algorithms theoretically with respect to their runtime

Greedy Algorithms

Greedy Algorithms

From Wikipedia:

> “A *greedy algorithm* is an algorithm that follows the problem solving *heuristic* of making the locally optimal choice at each stage with the hope of finding a global optimum.”

- Note: typically greedy algorithms do not find the global optimum

Lecture Outline Greedy Algorithms

What we will see:

❶ Example 1: Money Change problem

❷ Example 2: ϵ-Greedy Algorithm for Multi-Armed Bandits

Example 1: Money Change

Change-making problem

- Given n coins of distinct values $w_1=1$, w_2, ..., w_n and a total change W (where w_1, ..., w_n, and W are integers).
- Minimize the total amount of coins Σx_i such that $\Sigma w_i x_i = W$ and where x_i is the number of times, coin i is given back as change.

Greedy Algorithm

Unless total change not reached:

add the largest coin which is not larger than the remaining amount to the change

Note: only optimal for standard coin sets, not for arbitrary ones!

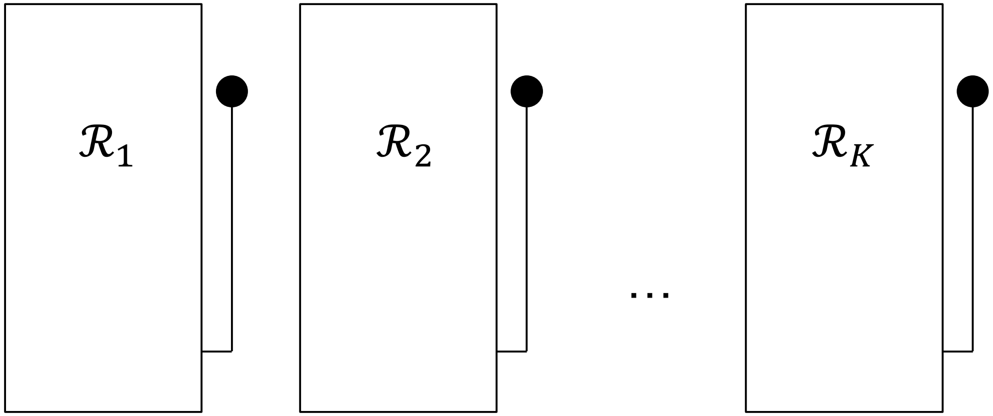
Related Problem:

finishing darts (from 501 to 0 with 9 darts)

Example 2: Multi-Armed Bandits

- generic problem of resource allocation
- classic reinforcement learning problem showing the exploration–exploitation tradeoff dilemma

Example 2: Multi-Armed Bandits



- K single-arm bandits with a lever
- Each bandit has a fixed but unknown probability distribution $\mathcal{R}\_i$ attached to it with a mean μ_i
- At each time step t, we decide to pull a lever (i) and get a reward r_t according to $\mathcal{R}\_i$
- Overall, we want to maximize the sum of the rewards
- The regret after T steps is defined as $\rho = T\mu_{max} - \sum_{t=1}^{T} r_t$

Exploration vs. Exploitation: The ϵ-Greedy Algorithm

Exploration: pull new levers (or underexplored ones) to get better estimates on the expected rewards

Exploitation: pull the arm, we think is the best arm

…the latter being the greedy approach here

The ϵ-Greedy Algorithm

- With probability 1-ϵ: pull the lever, we think is best
- With probability ϵ: pull a random lever (uniformly)

To be decided (not discussed further here):

How to estimate the probabilities (e.g. pulling each lever once at first)

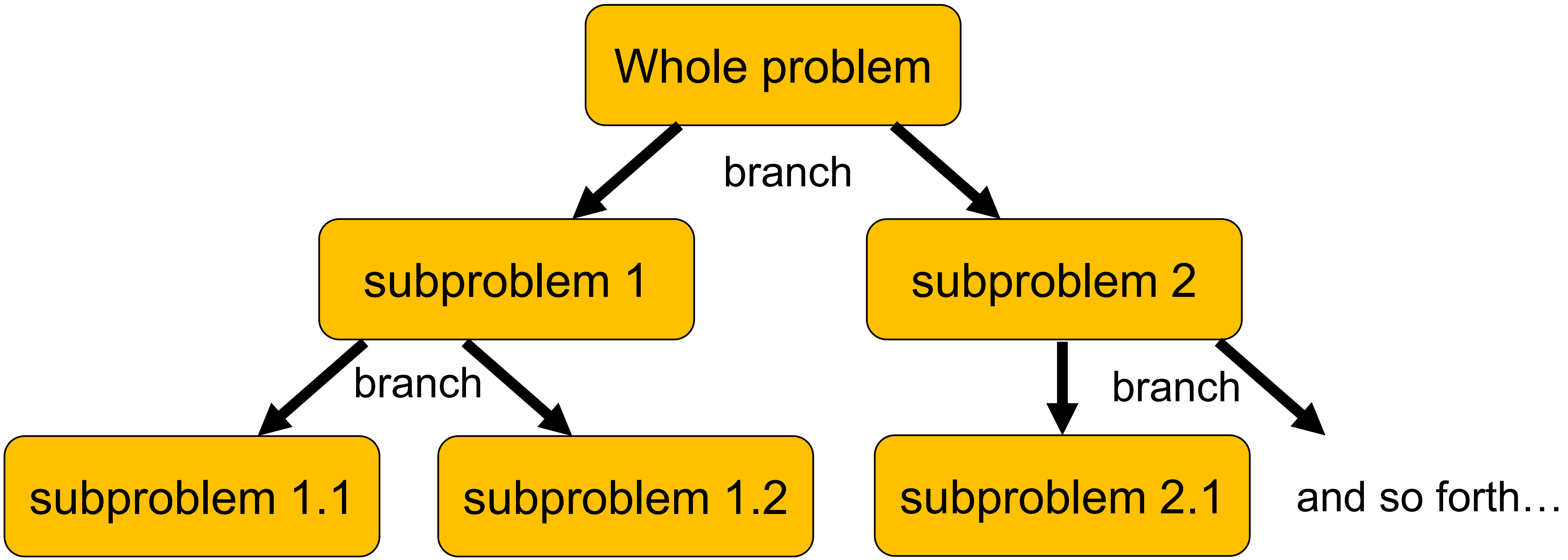
How to choose ϵ (constant vs. decreasing over time)

constant ϵ gives linear regret

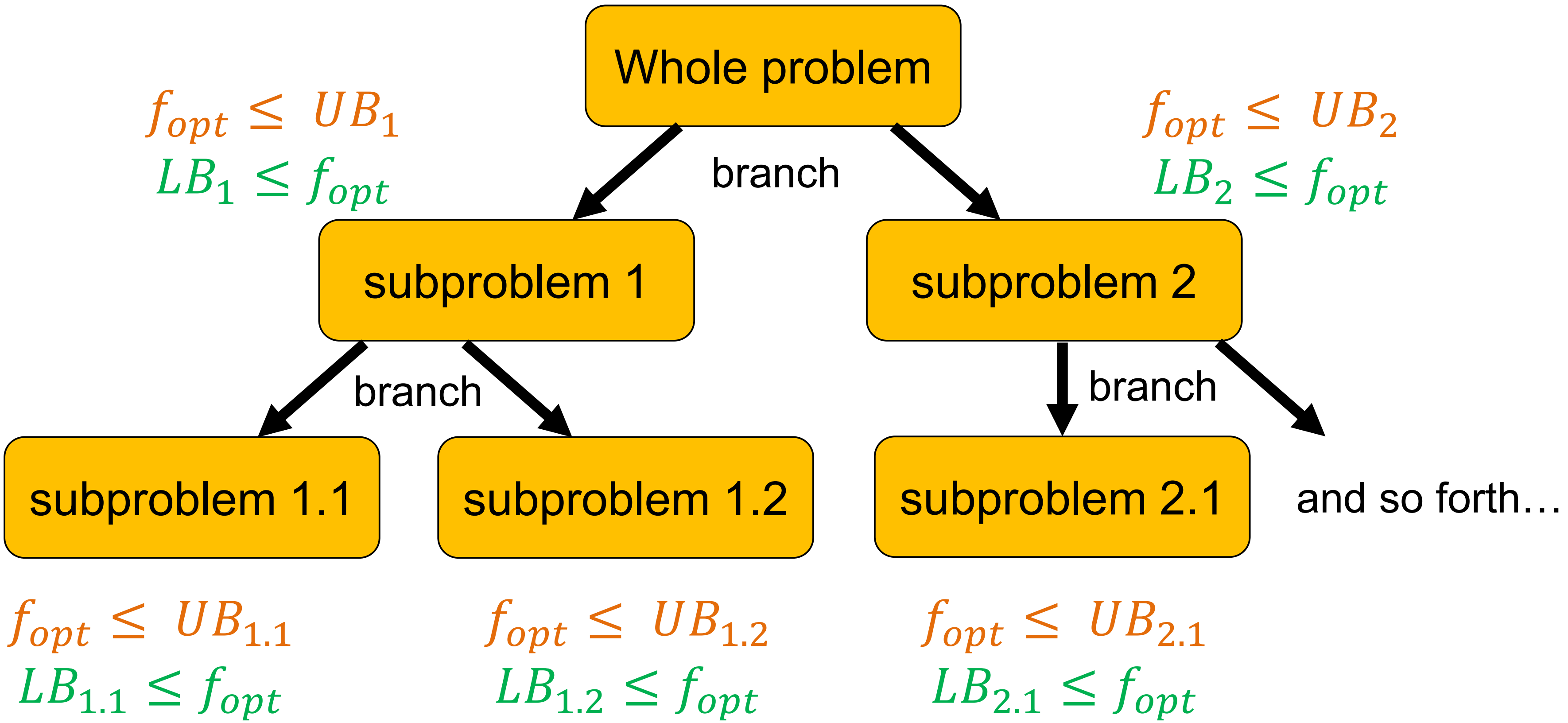
Branch and Bound

Idea Behind Branch and Bound

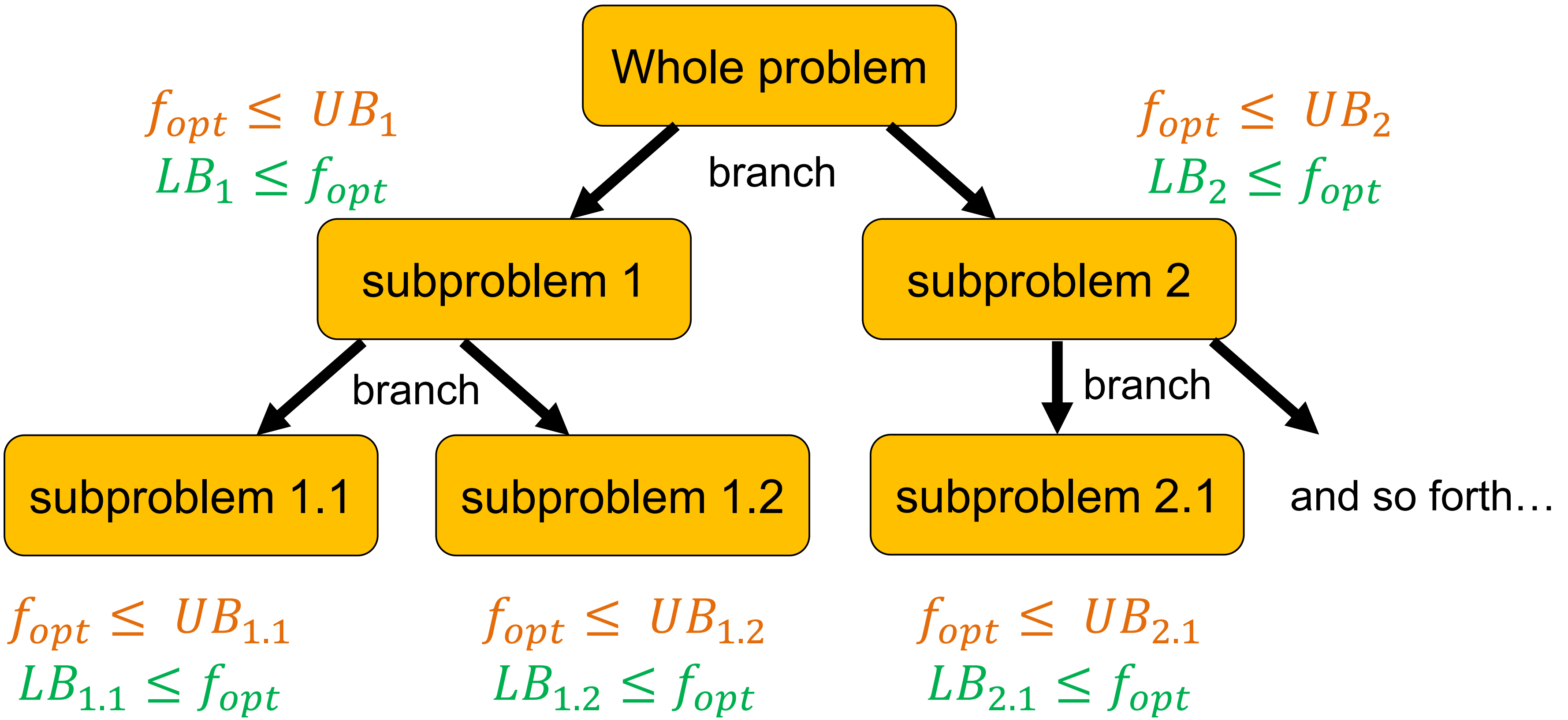
- Basically enumerates the entire search space
- But uses clever strategies to avoid enumerations in bad areas



Idea Behind Branch and Bound



Idea Behind Branch and Bound



when can we actually avoid evaluating all solutions?

Idea Behind Branch and Bound

Whole problem

branch

$f_{opt} \leq UB_1$
$LB_1 \leq f_{opt}$

$f_{opt} \leq UB_2$
$LB_2 \leq f_{opt}$

subproblem 1

subproblem 2

branch

branch

subproblem 1.1

subproblem 1.2

subproblem 2.1

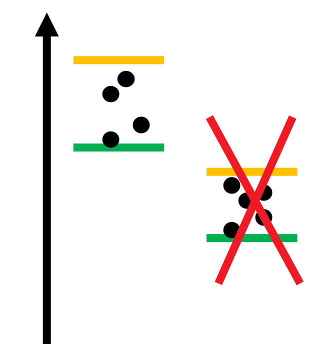
and so forth…

$f_{opt} \leq UB_{1.1}$
$LB_{1.1} \leq f_{opt}$

$f_{opt} \leq UB_{1.2}$
$LB_{1.2} \leq f_{opt}$

$f_{opt} \leq UB_{2.1}$
$LB_{2.1} \leq f_{opt}$

max.



We can stop exploring/branching if

- UB=LB
- UB for new subproblem lower than LB for another

[when maximizing]

How do we get Upper and Lower Bounds?

We assume again maximization here…

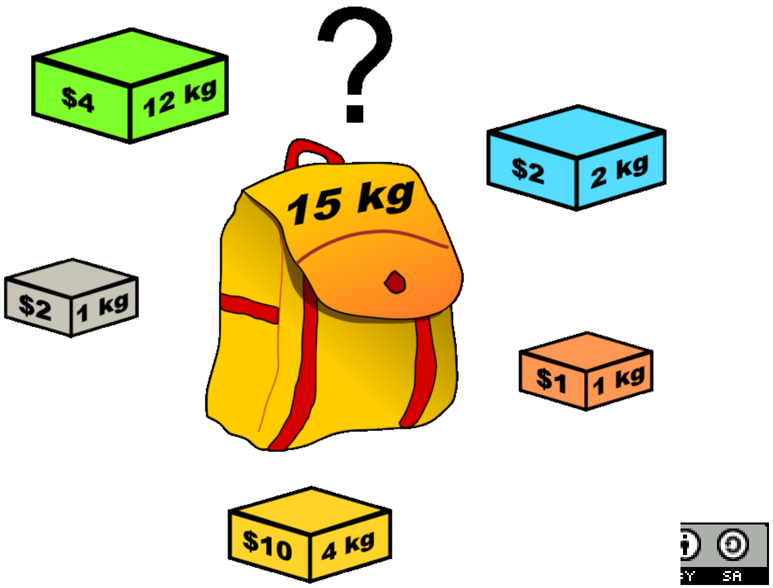
- A feasible solution gives us a lower bound

  the optimum will be at least as good as a solution, we know
- Hence, fast (non-exact) algorithms such as greedy can give us lower bounds
- For upper bounds, we can relax the problem

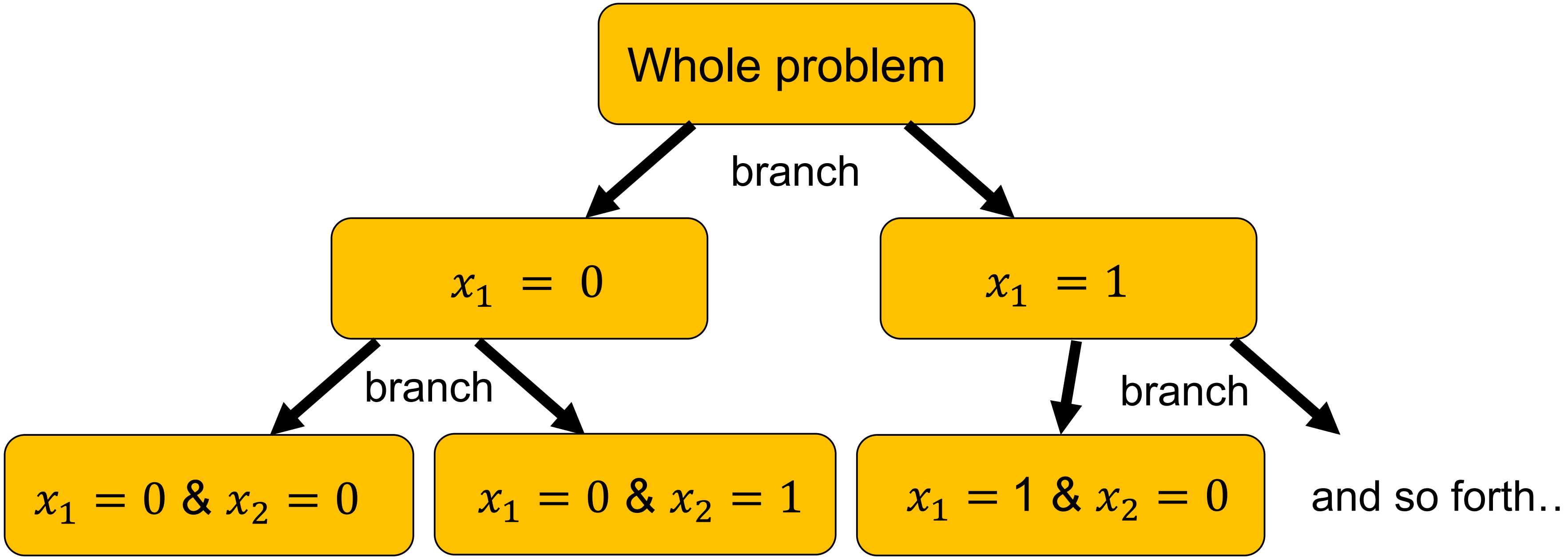
  for example, by removing constraints

An Example: Branch&Bound for the KP

[redacted]



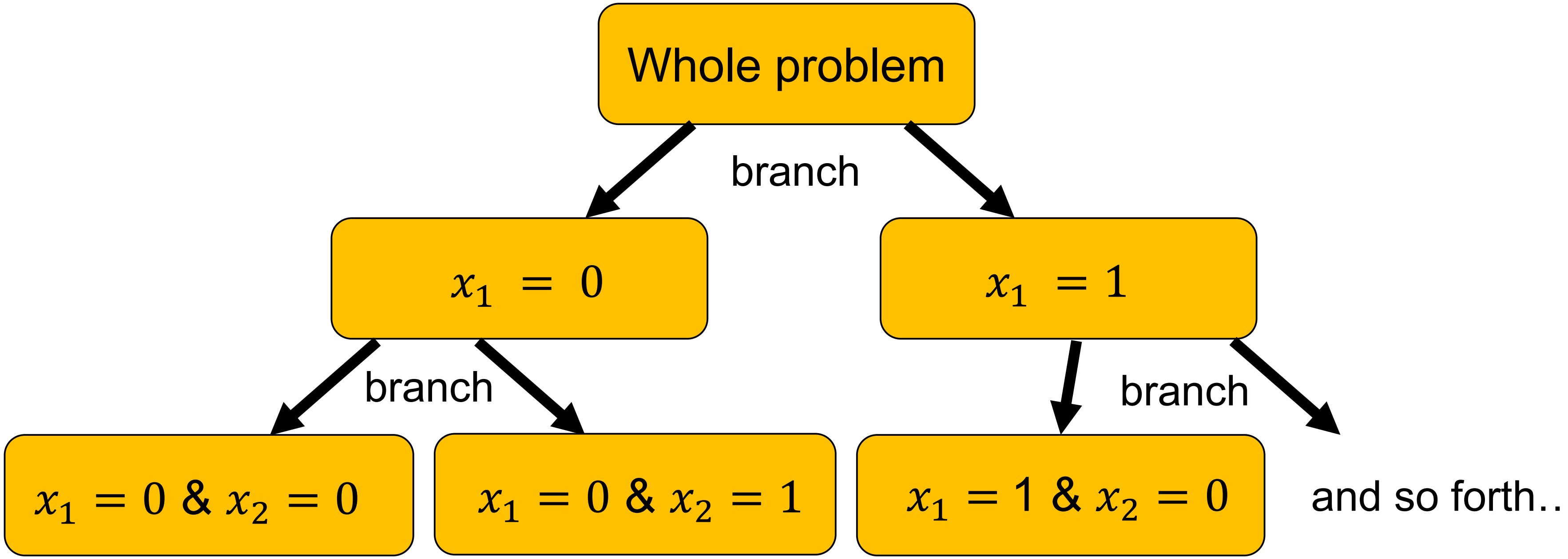
KP: How to Branch?



! order of variables plays an important role

optimally, the subproblems don't overlap

KP: How to Bound?



Maximization, so LB by greedy approach for example:

Choose items in decreasing profit/weight ratio until knapsack full

UB by relaxation of constraints (on the variables here):

Use greedy algorithm and pack add. item partially if there is space

…this variable can be used to branch next

Dynamic Programming

Dynamic Programming

Wikipedia:

"[...] **dynamic programming** is a method for solving a complex problem by breaking it down into a collection of simpler subproblems."

But that's not all:

- dynamic programming also makes sure that the subproblems are not solved too often but only once by keeping the solutions of simpler subproblems in memory ("trading space vs. time")
- it is an exact method, i.e. in comparison to the greedy approach, it always solves a problem to optimality

Two Properties Needed

Optimal Substructure

A solution can be constructed efficiently from optimal solutions of sub-problems

Overlapping Subproblems

Wikipedia: "[...] a problem is said to have **overlapping subproblems** if the problem can be broken down into subproblems which are reused several times or a recursive algorithm for the problem solves the same subproblem over and over rather than always generating new subproblems."

Main Idea Behind Dynamic Programming

Main idea: solve larger subproblems by breaking them down to smaller, easier subproblems in a recursive manner

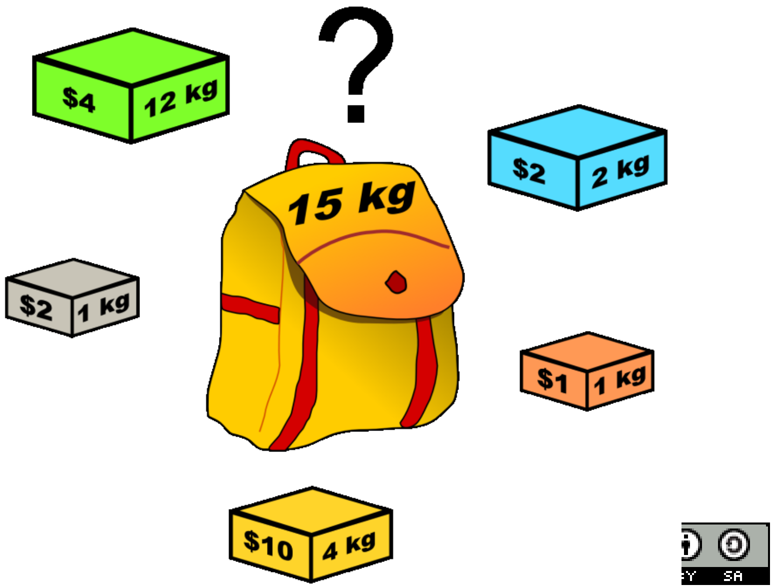
Typical Algorithm Design:

1. decompose the problem into subproblems and think about how to solve a larger problem with the solutions of its subproblems
2. specify how you compute the value of a larger problem recursively with the help of the optimal values of its subproblems ("Bellman equation")
3. bottom-up solving of the subproblems (i.e. computing their optimal value), starting from the smallest by using the Bellman equality and a table structure to store the optimal values
4. eventually construct the final solution (can be omitted if only the value of an optimal solution is sought)

Example: The Knapsack Problem (KP)

Knapsack Problem

[redacted]



What are Good Subproblem Definitions for the KP?

Consider the following subproblems:

1) $P(i)$: optimal profit when packing exactly i items
2) $P(i)$: optimal profit when packing at most i items
3) $P(i,j)$: optimal profit when allowing to pack the first i items into a knapsack of size j

Which one allows us to solve larger subproblems from the solutions of smaller ones?

Which value are we actually interest in, when trying to solve the problem?

Opt. Substructure and Overlapping Subproblems

Consider the following subproblem:

$P(i,j)$: optimal profit when allowing to pack the first i items into a knapsack of size j

Optimal Substructure

The optimal choice of whether taking item i or not can be made easily for a knapsack of weight j if we know the optimal choice for items $1 \dots i-1$:

$$P(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1, j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Overlapping Subproblems

a recursive implementation of the Bellman equation is simple, but the $P(i,j)$ might need to be computed more than once!

Dynamic Programming Approach to the KP

To circumvent solving the subproblems more than once, we can store their results (in a matrix for example)...

knapsack weight →

items ↓

P(i,j)	0	1	2	3	...			W-1	W
0									
1					P(i,j)				
2									
...									
n-1									
n									

best achievable profit with items 1...i and a knapsack of size j

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits (5,4), (7,10), (2,3), (4,5), and (3,3). Weight restriction is W=11.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0												
1												
2												
3												
4												
5												

initialization:

$P(i,j) = 0$ if $i = 0$ or $j = 0$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits (5,4), (7,10), (2,3), (4,5), and (3,3). Weight restriction is W=11.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0											
2	0											
3	0											
4	0											
5	0											

initialization:
$P(i,j) = 0$ if $i = 0$ or $j = 0$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0											
2	0											
3	0											
4	0											
5	0											

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0										
2	0											
3	0											
4	0											
5	0											

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5{,}4)$, $(7{,}10)$, $(2{,}3)$, $(4{,}5)$, and $(3{,}3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0									
2	0											
3	0											
4	0											
5	0											

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0							
2	0											
3	0											
4	0											
5	0											

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \le j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4						
2	0											
3	0											
4	0											
5	0											

$+p_1 (= 4)$

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \le j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4					
2	0											
3	0											
4	0											
5	0											

$+p_1(= 4)$

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight $\longrightarrow$

items $\downarrow$

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0											
3	0											
4	0											
5	0											

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4					
3	0											
4	0											
5	0											

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \le j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5{,}4)$, $(7{,}10)$, $(2{,}3)$, $(4{,}5)$, and $(3{,}3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10				
3	0											
4	0											
5	0											

$+p_2(= 10)$

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j),\, p_i + P(i-1, j-w_i)\} & \text{if } w_i \le j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5{,}4)$, $(7{,}10)$, $(2{,}3)$, $(4{,}5)$, and $(3{,}3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0											
4	0											
5	0											

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j),\ p_i + P(i-1,j-w_i)\} & \text{if } w_i \le j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3							
4	0											
5	0											

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3	4						
4	0											
5	0											

$+p_3(=3)$

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5{,}4)$, $(7{,}10)$, $(2{,}3)$, $(4{,}5)$, and $(3{,}3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3	4	4					
4	0											
5	0											

$+p_3(=3)$

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j),\, p_i + P(i-1,j-w_i)\} & \text{if } w_i \le j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5{,}4)$, $(7{,}10)$, $(2{,}3)$, $(4{,}5)$, and $(3{,}3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3	4	4	10	etc.			
4	0											
5	0											

$+p_3(=3)$

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j),\, p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5{,}4)$, $(7{,}10)$, $(2{,}3)$, $(4{,}5)$, and $(3{,}3)$. Weight restriction is $W = 11$.

knapsack weight

items

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3	4	4	10	10	13	13	13
4	0	0	3	3	5	5	8	10	10	13	13	15
5	0	0	3	3	5	6	8	10	10	13	13	15

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5{,}4)$, $(7{,}10)$, $(2{,}3)$, $(4{,}5)$, and $(3{,}3)$. Weight restriction is $W = 11$.

knapsack weight ⟶

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3	4	4	10	10	13	13	13
4	0	0	3	3	5	5	8	10	10	13	13	15
5	0	0	3	3	5	6	8	10	10	13	13	**15**

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Question: How to obtain the actual packing?

Answer: we just need to remember where the $\max$ came from!

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3	4	4	10	10	13	13	13
4	0	0	3	3	5	5	8	10	10	13	13	15
5	0	0	3	3	5	6	8	10	10	13	13	15

$x_1 = 0$ $x_2 = 1$ $x_3 = 0$ $x_4 = 1$ $x_5 = 0$

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$