Optimization for Machine Learning

Lecture 6: Discrete Optimization

December 8, 2022

TC2 - Optimisation

Université Paris-Saclay

Anne Auger and Dimo Brockhoff

Inria Saclay – Ile-de-France

Course Overview

Date		Topic
Thu, 3.11.2022	DB	Introduction
Thu, 10.11.2022	AA	Continuous Optimization I: differentiability, gradients, convexity, optimality conditions
Thu, 17.11.2022	AA	Continuous Optimization II: constrained optimization, gradient-based algorithms, stochastic gradient
Thu, 24.11.2022	AA	Continuous Optimization III: stochastic algorithms, derivative-free optimization written test / « contrôle continue »
Thu, 1.12.2022	DB	Constrained optimization, ~~Discrete Optimization I:~~ graph theory, ~~greedy algorithms~~
Thu, 8.12.2022	DB	Discrete Optimization: greedy algorithms, branch and bound, dynamic programming
Thu 15.12.2022	DB	Written exam (2 hours starting at 1:30pm)
		classes from 13h30 – 16h45 (2nd break at end)

Concrete Information About Exam

- on site, offline exam
- multiple choice, typically 4 answers each (1-4 answers correct)
- closed book (nothing allowed but pen)
  - → easier questions ☺
  - like in mini-exam
- next Thursday (Dec. 15) @ 1:30pm
- 2 hours

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

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:

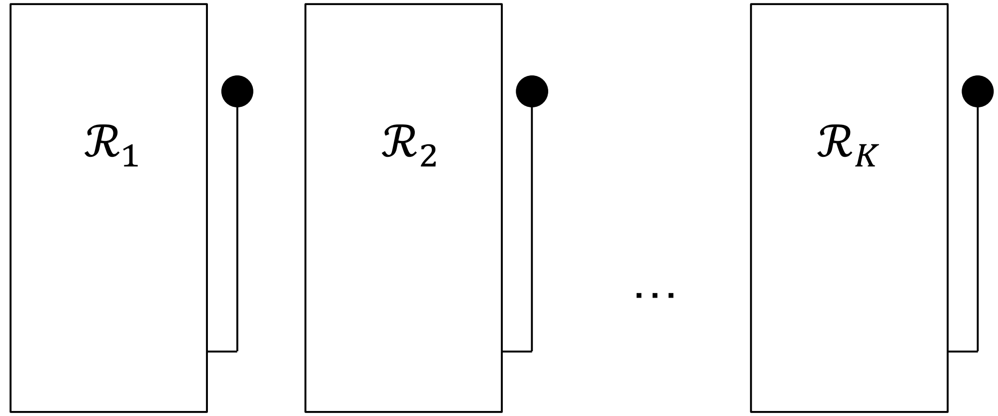
finishing darts (from 501 to 0 with 9 darts)

Example 2: Multi-Armed Bandits

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

Yamaguch
i先生

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)

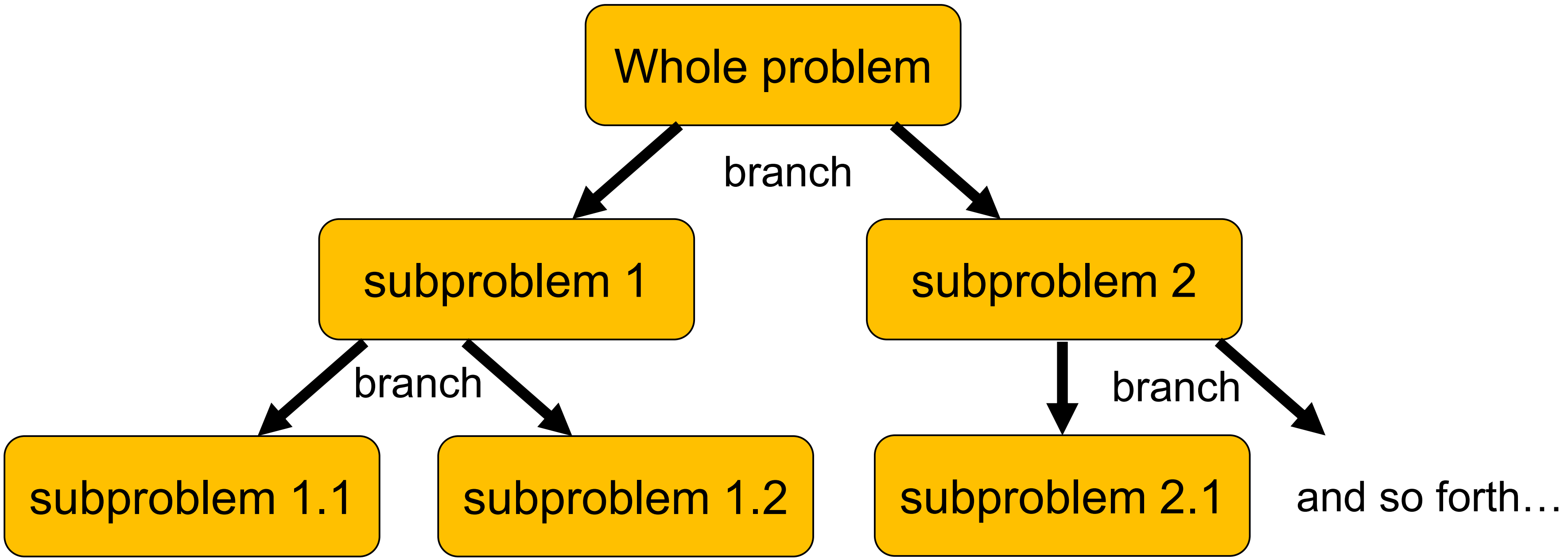
How to choose ϵ (constant vs. decreasing over time)

constant ϵ gives linear regret

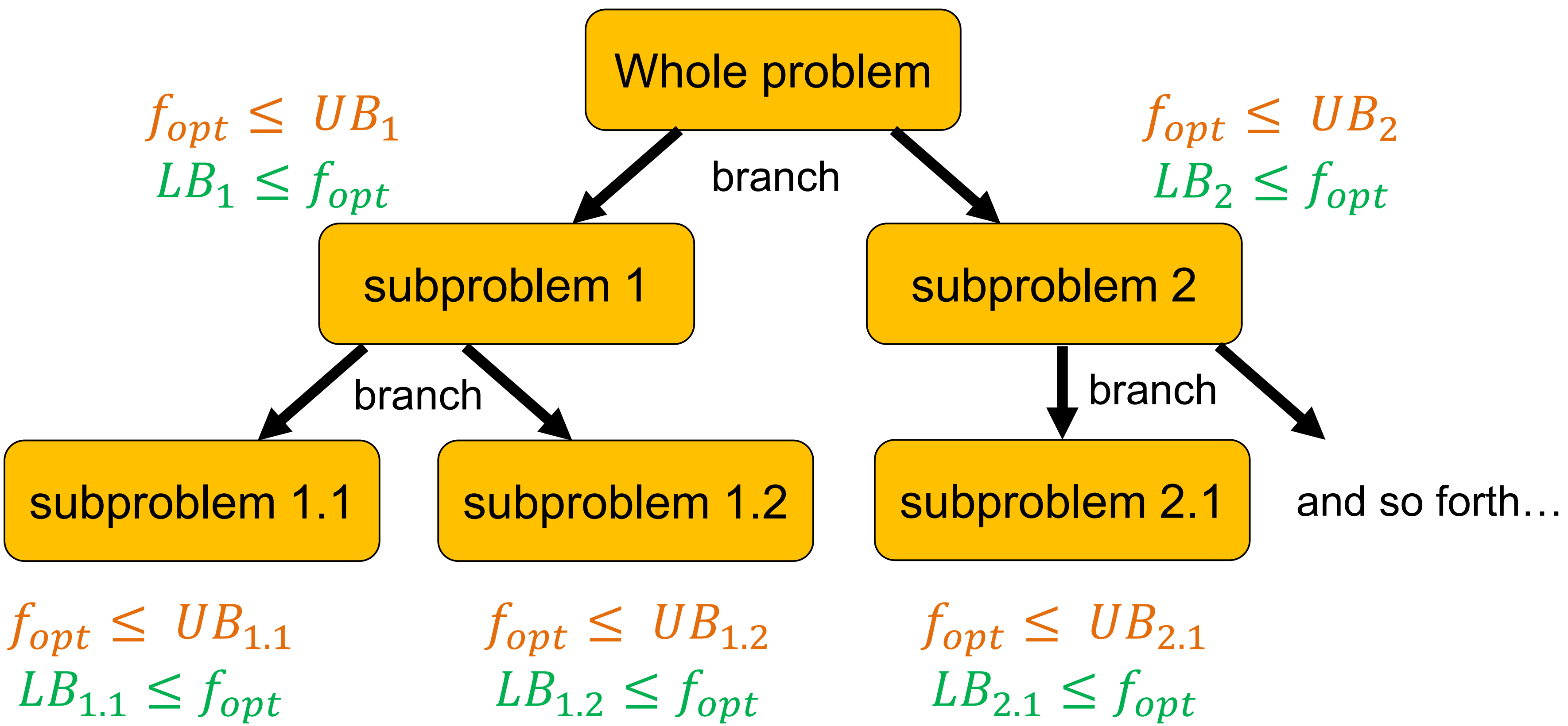
Branch and Bound

Idea Behind Branch and Bound

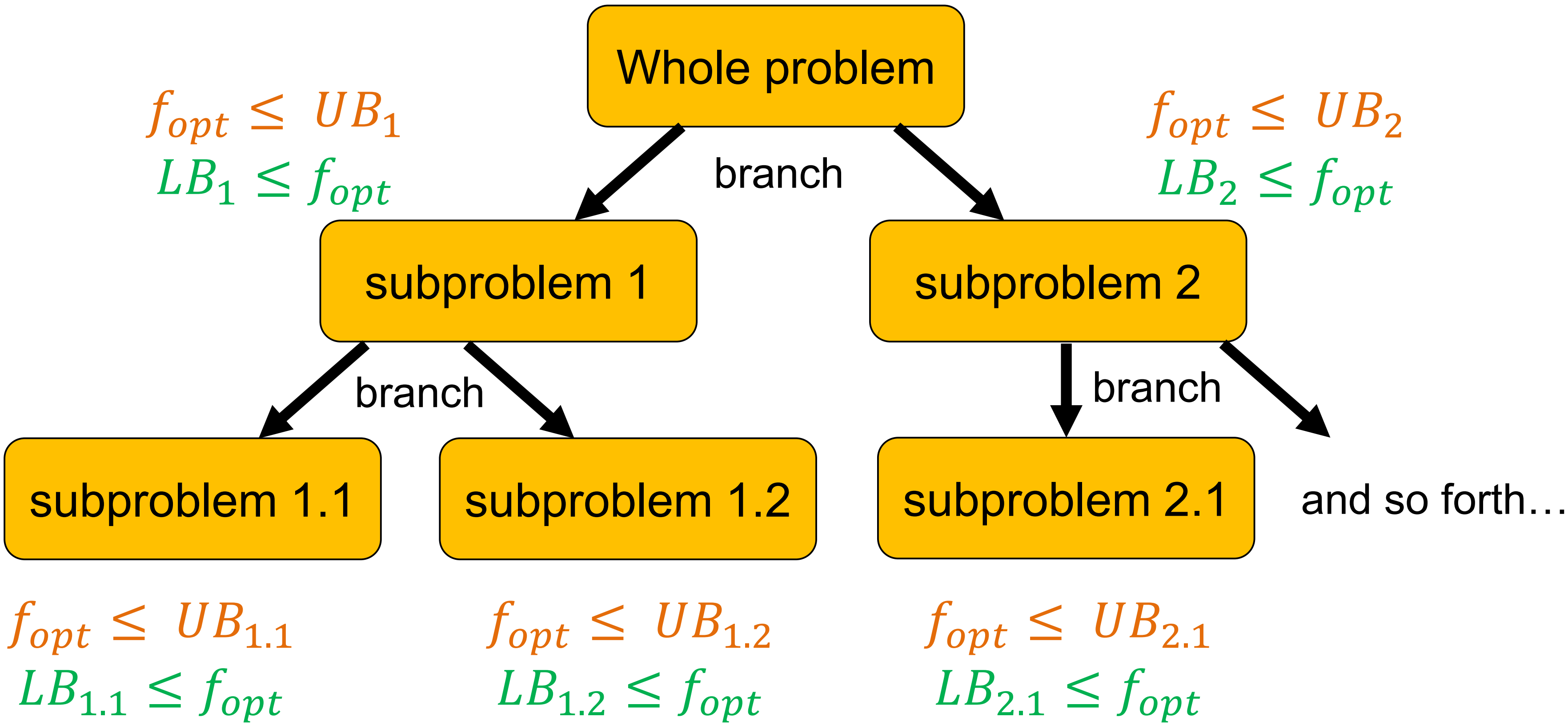
- Basically enumerates the entire search space
- But uses clever strategies to avoid enumerations in bad areas



Idea Behind Branch and Bound

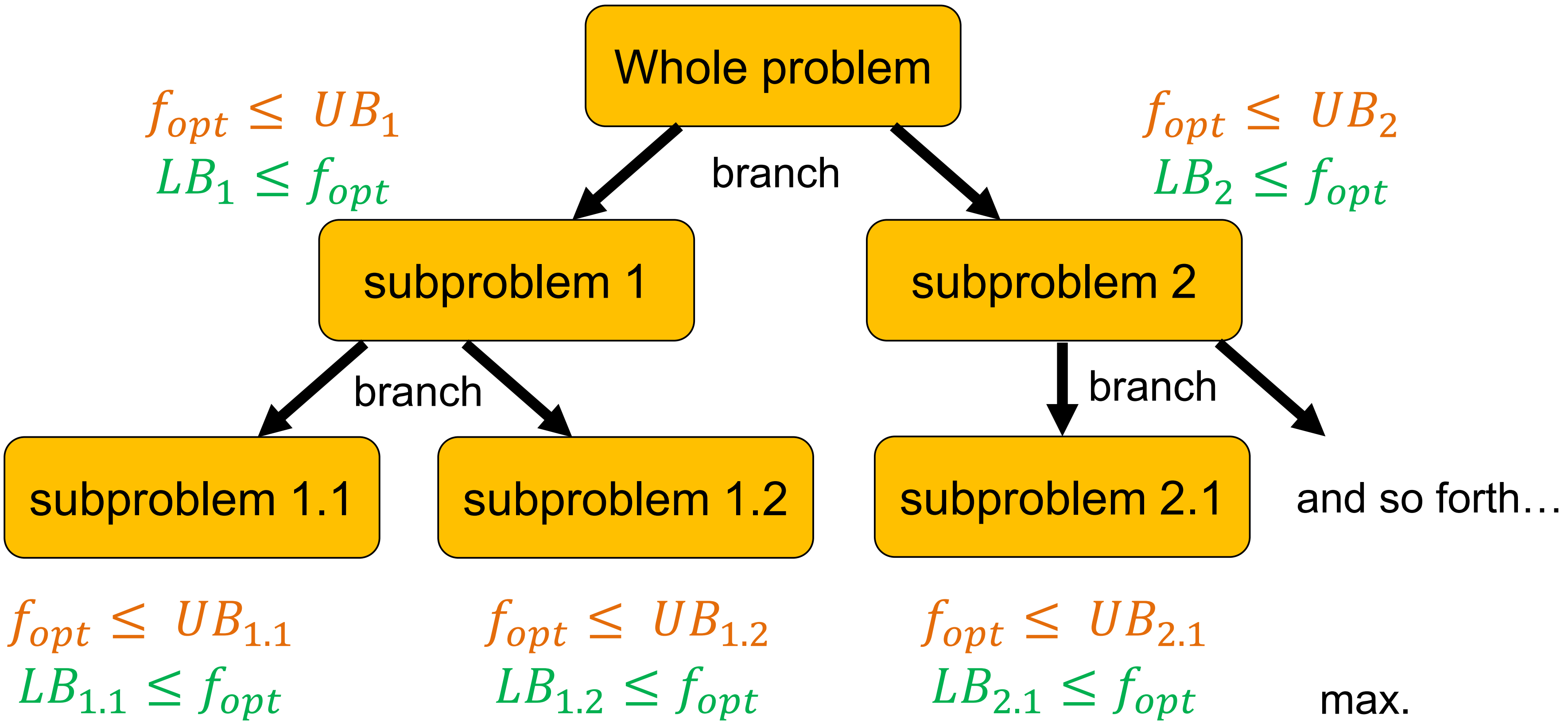


Idea Behind Branch and Bound



when can we actually avoid evaluating all solutions?

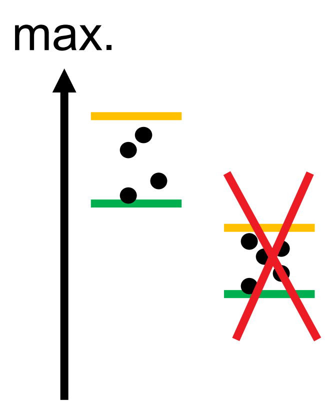
Idea Behind Branch and Bound



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

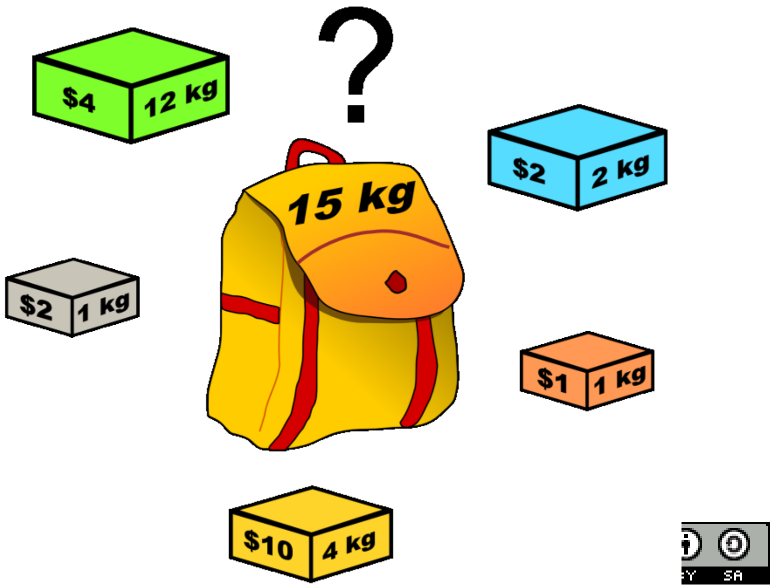
  the optimum will be at least as good as a solution, we know
- Hence, fast (non-exact) algorithms such as greedy can give us lower bounds
- For upper bounds, we can relax the problem

  for example, by removing constraints

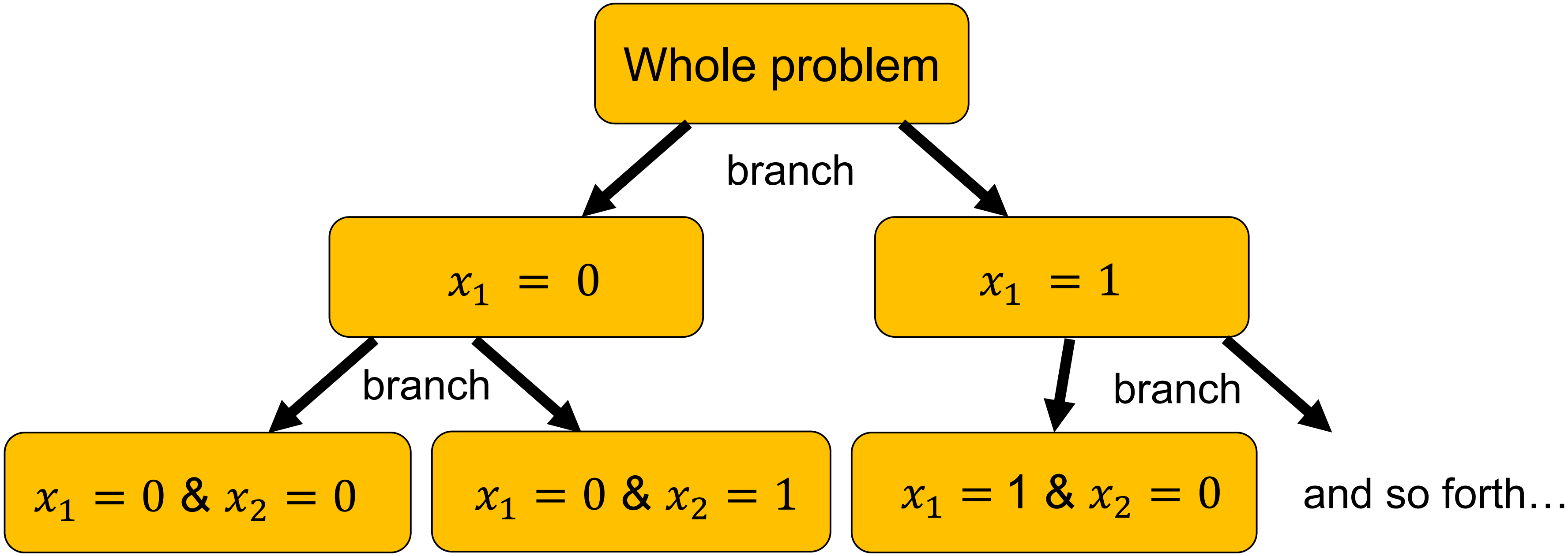
Knapsack Problem

$$\max. \sum_{j=1}^{n} p_j x_j \;\; with \; x_j \in \{0,1\}$$

$$\text{s.t.} \sum_{j=1}^{n} w_j x_j \leq W$$



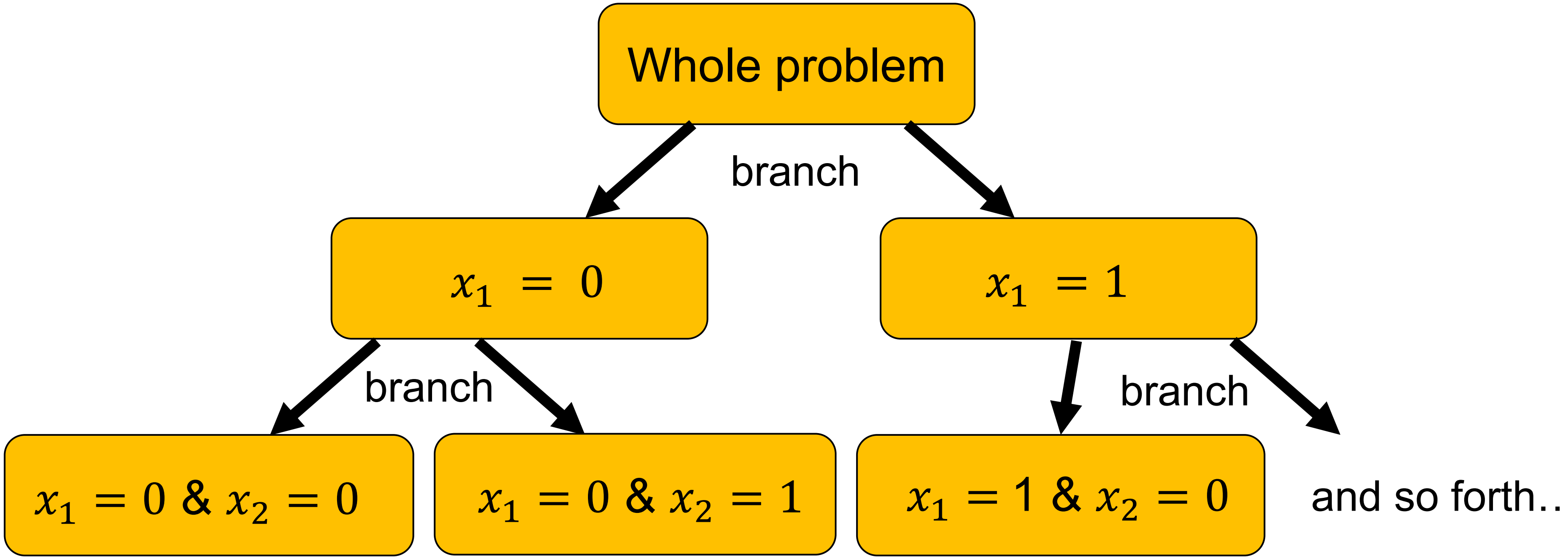
KP: How to Branch?



! 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]** refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner."

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:

❶ decompose the problem into subproblems and think about how to solve a larger problem with the solutions of its subproblems

❷ specify how you compute the value of a larger problem recursively with the help of the optimal values of its subproblems ("Bellman equation")

❸ 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

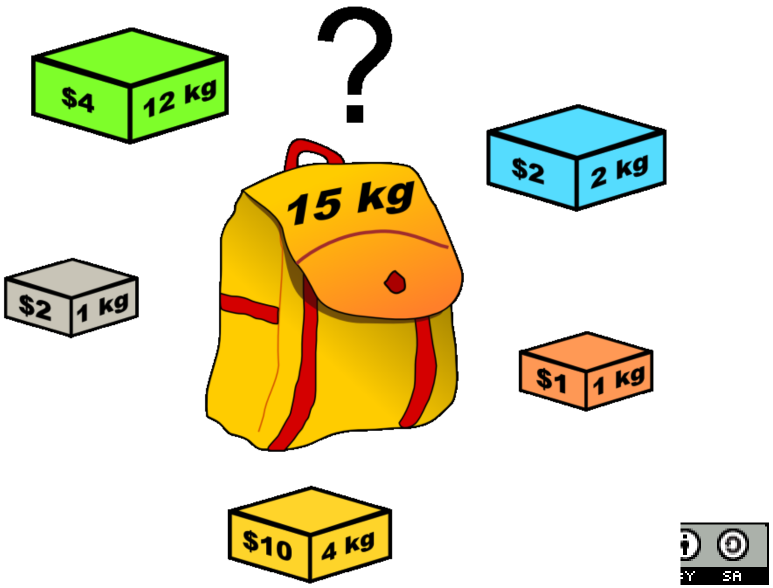
❹ eventually construct the final solution (can be omitted if only the value of an optimal solution is sought)

Example: The Knapsack Problem (KP)

Knapsack Problem

$$\max. \sum_{j=1}^{n} p_j x_j \;\; with \; x_j \in \{0,1\}$$

$$\text{s.t.} \sum_{j=1}^{n} w_j x_j \leq W$$



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:

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

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						
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 →

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 →

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											
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					
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	10				
3	0											
4	0											
5	0											

$+p_2 (= 10)$

for $i = 1$ to n:

for $j = 1$ to W:

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

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											
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	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:

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

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 $+p_3(=3)$	4	4					
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	10	etc.			
4	0											
5	0											

$+p_3(=3)$

for $i = 1$ to n:

for $j = 1$ to W:

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

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	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:

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

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	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

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:

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

(Randomized) Search Heuristics

Motivation General Search Heuristics

- often, problem complicated and not much time available to develop a problem-specific algorithm
- search heuristics are a good choice:
  - relatively easy to implement
  - easy to adapt/change/improve
    - e.g. when the problem formulation changes in an early product design phase
    - or when slightly different problems need to be solved over time
- randomized/stochastic algorithms are a good choice because they are robust to noise

Lecture Outline Randomized Search Heuristics

Which algorithms will we touch?

❶ Randomized Local Search (RLS)

❷ Variable Neighborhood Search (VNS)

❸ Tabu Search (TS)

❹ Evolutionary Algorithms (EAs)

Neighborhoods

For most (stochastic) search heuristics, we need to define a *neighborhood structure*

- which search points are close to each other?

Example: k-bit flip / Hamming distance k neighborhood

- search space: bitstrings of length n ($\Omega=\{0,1\}^n$)
- two search points are neighbors if their Hamming distance is k
- in other words: x and y are neighbors if we can flip exactly k bits in x to obtain y
- 0001001101 is neighbor of
  0001000101 for k=1
  0101000101 for k=2
  1101000101 for k=3

Neighborhoods II

Example: possible neighborhoods for the knapsack problem

- search space again bitstrings of length n ($\Omega=\{0,1\}^n$)
- Hamming distance 1 neighborhood:
  - add an item or remove it from the packing
- replacing 2 items neighborhood:
  - replace one chosen item with an unchosen one
  - makes only sense in combination with other neighborhoods because the number of items stays constant
- Hamming distance 2 neighborhood on the contrary:
  - allows to change 2 arbitrary items, e.g.
    - add 2 new items
    - remove 2 chosen items
    - or replace one chosen item with an unchosen one

Randomized Local Search (RLS)

Idea behind (Randomized) Local Search:

- explore the local neighborhood of the current solution (randomly)

Pure Random Search:

- go to randomly chosen neighbor

First Improvement Local Search:

- go to first (randomly) chosen neighbor which is better

Best Improvement strategy:

- always go to the best neighbor
- not random anymore
- computationally expensive if neighborhood large

Variable Neighborhood Search

Main Idea: [N. Mladenovic and P. Hansen, 1997]

- change the neighborhood from time to time
  - local optima not necessarily the same for different neighborhood operators
  - but often close to each other
  - global optimum is local optimum for all neighborhoods
- rather a framework than a concrete algorithm
  - e.g. deterministic and stochastic neighborhood changes
- typically combined with (i) first improvement, (ii) a random order in which the neighbors are visited and (iii) restarts

N. Mladenovic and P. Hansen (1997). "Variable neighborhood search". Computers and Operations Research 24 (11): 1097–1100.

Tabu Search

Disadvantages of local searches (with or without varying neighborhoods)

- they get stuck in local optima
- have problems to traverse large plateaus of equal objective function value (“random walk”)

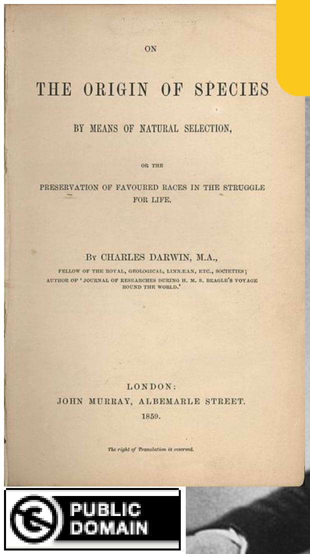
Tabu search addresses these by

- allowing worsening moves if all neighbors are explored
- introducing a tabu list of temporarily not allowed moves
- those restricted moves are
  - problem-specific and
  - can be specific solutions or not permitted “search directions” such as “don’t include this edge anymore” or “do not flip this specific bit”
- the tabu list is typically restricted in size and after a while, restricted moves are permitted again

One class of (bio-inspired) stochastic optimization algorithms: Evolutionary Algorithms (EAs)

- Class of optimization algorithms originally inspired by the idea of biological evolution
- selection, mutation, recombination

1859


ON

THE ORIGIN OF SPECIES

BY MEANS OF NATURAL SELECTION,

OR THE

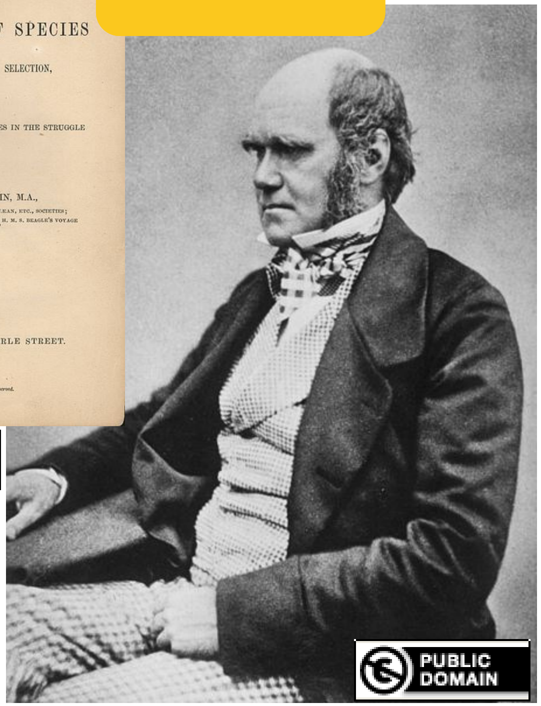
PRESERVATION OF FAVOURED RACES IN THE STRUGGLE FOR LIFE.

BY CHARLES DARWIN, M.A.,

LONDON:

JOHN MURRAY, ALBEMARLE STREET.

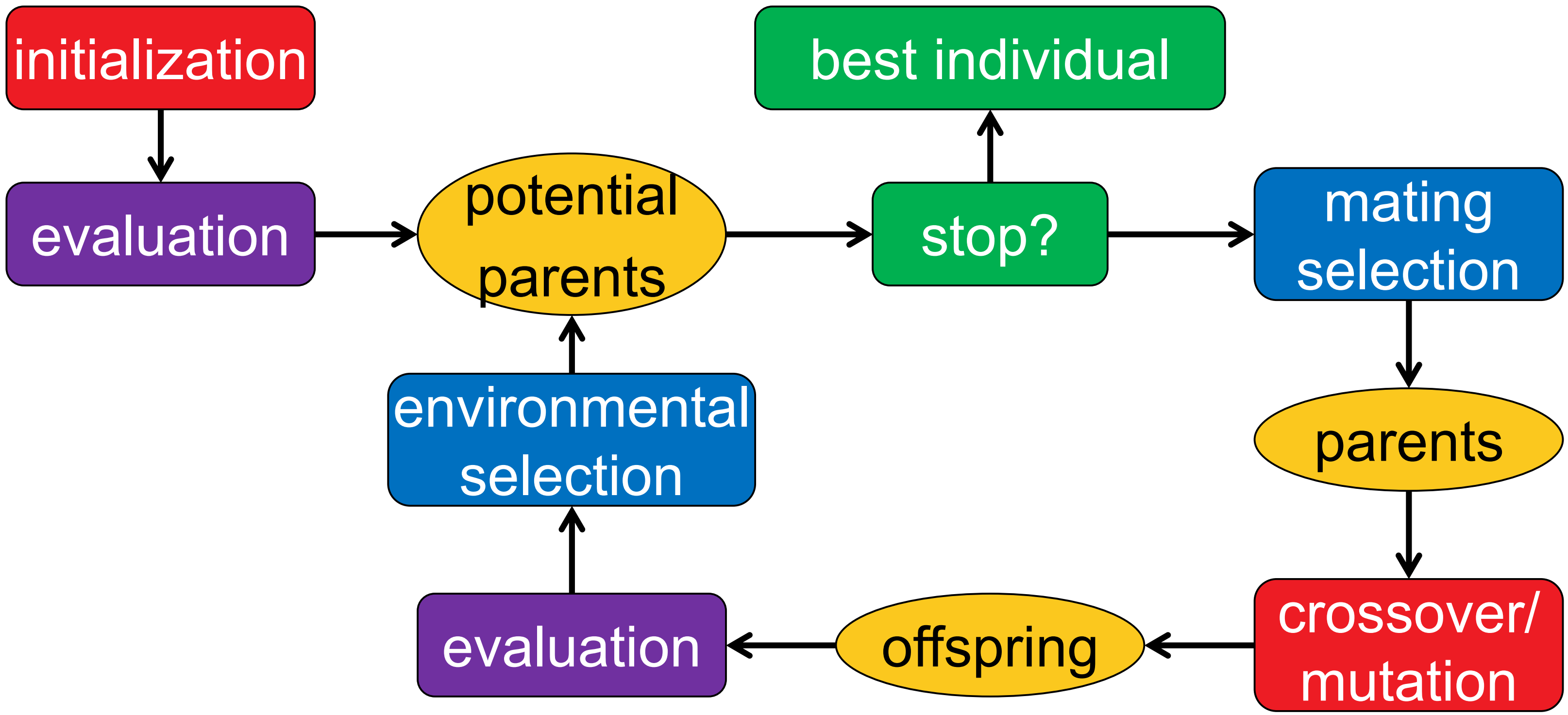
1859.



Metaphors

Classical Optimization	Evolutionary Computation
variables or parameters	variables or chromosomes
candidate solution vector of decision variables / design variables / object variables	individual, offspring, parent
set of candidate solutions	population
objective function loss function cost function error function	fitness function
iteration	generation

Generic Framework of an EA



stochastic operators

“Darwinism”

stopping criteria

Important:
representation (search space)

The Historic Roots of EAs

Genetic Algorithms (GA)

J. Holland 1975 and D. Goldberg (USA)

$$\Omega = \{0, 1\}^n$$

Evolution Strategies (ES)

I. Rechenberg and H.P. Schwefel, 1965 (Berlin)

$$\Omega = \mathbb{R}^n$$

Evolutionary Programming (EP)

L.J. Fogel 1966 (USA)

Genetic Programming (GP)

J. Koza 1990 (USA)

$$\Omega = \text{space of all programs}$$

nowadays one umbrella term: evolutionary algorithms

Note: Handling Constraints

Several generic ways to handle constraints, e.g.:

- resampling until a new feasible point is found ("often bad idea")
- penalty function approach: add constraint violation term (potentially scaled)
- repair approach: after generation of a new point, repair it (e.g. with a heuristic) to become feasible again if infeasible
  - continue to use repaired solution in the population or
  - use repaired solution only for the evaluation?
- multiobjective approach: keep objective function and constraint functions separate and try to optimize all of them in parallel
- ...

Examples for some EA parts

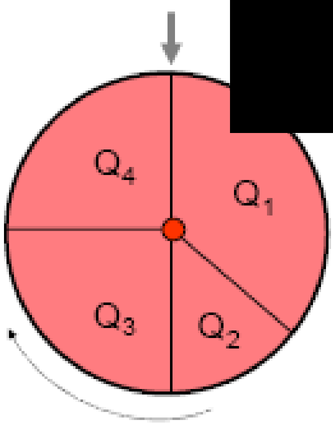
Selection is the major determinant for specifying the trade-off between exploitation and exploration

Selection is either

stochastic — or — deterministic

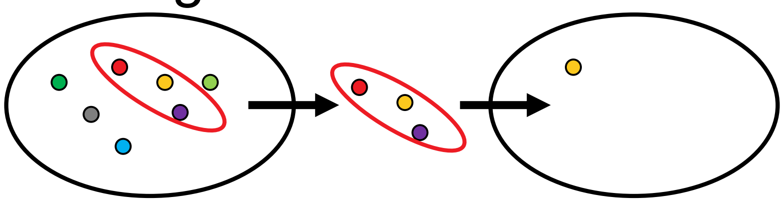
e.g. fitness proportional

[redacted]



Disadvantage: depends on scaling of f

e.g. via a tournament



e.g. (μ+λ), (μ,λ)

best μ from offspring *and* parents

best μ from offspring only

Mating selection (selection for variation): usually stochastic

Environmental selection (selection for survival): often deterministic

Variation Operators

Variation aims at generating new individuals on the basis of those individuals selected for mating

Variation = Mutation and Recombination/Crossover

mutation: *mut:* [redacted]

recombination: *recomb:* [redacted] where [redacted] and [redacted]

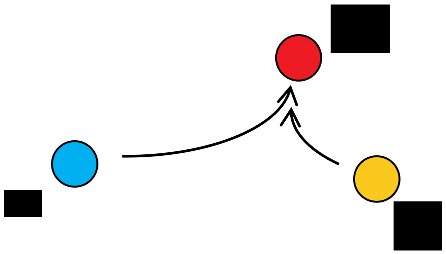
- choice always depends on the problem and the chosen representation
- however, there are some operators that are applicable to a wide range of problems and tailored to standard representations such as vectors, permutations, trees, etc.

Two desirable properties for mutation operators:

- every solution can be generation from every other with a probability greater than 0 (“exhaustiveness”)
- $d(x, x') < d(x, x'') => Prob(\mathrm{mut}(x) = x') > Prob(\mathrm{mut}(x) = x'')$ (“locality”)

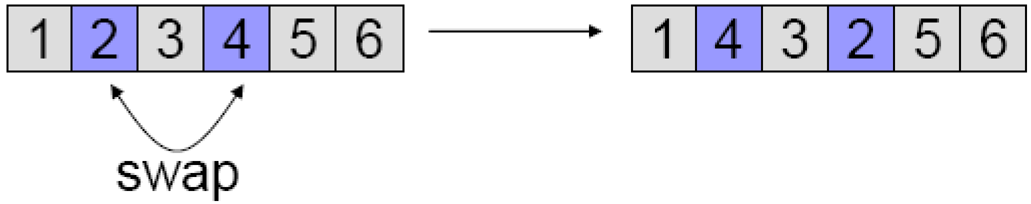
Desirable property of recombination operators (“in-between-ness”):

$$x'' = \ \mathrm{recomb}(x, x') \Rightarrow d(x'', x) \leq d(x, x') \wedge d(x'', x') \leq d(x, x')$$

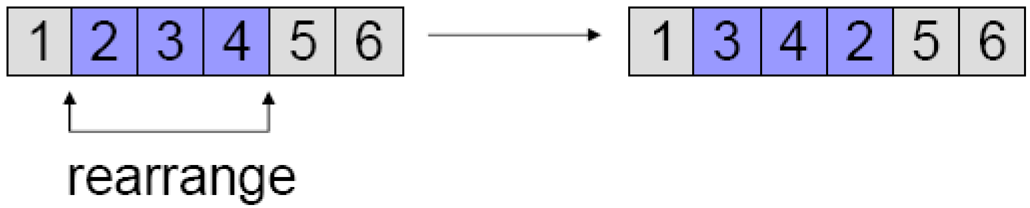


Examples of Mutation Operators on Permutations

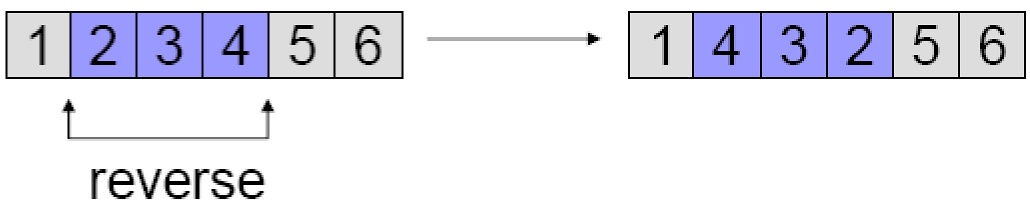
Swap:



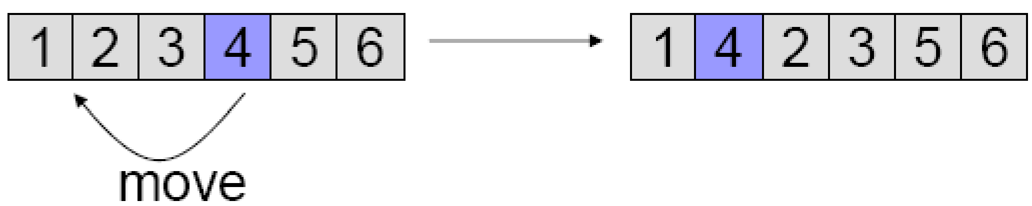
Scramble:



Invert:

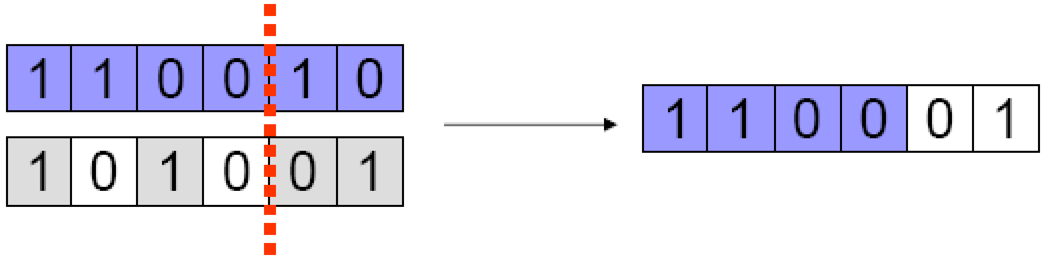


Insert:

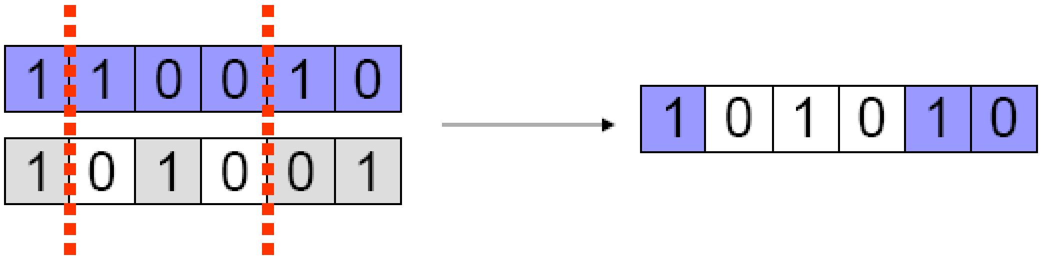


Examples of Recombination Operators: $\{0,1\}^n$

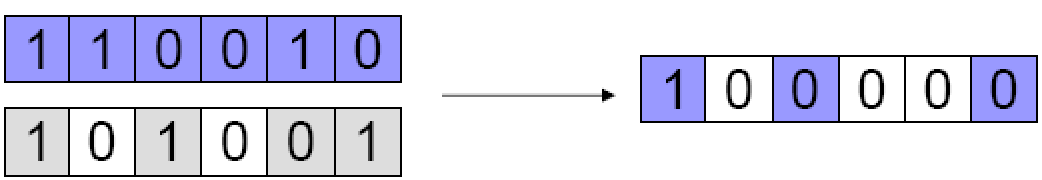
1-point crossover



n-point crossover



uniform crossover



choose each bit independently from one parent or another

A Canonical Genetic Algorithm

- binary search space, maximization
- uniform initialization
- generational cycle: of the population
  - evaluation of solutions
  - mating selection (e.g. roulette wheel)
  - crossover (e.g. 1-point)
  - environmental selection (e.g. plus-selection)

Conclusions

- EAs are generic algorithms (randomized search heuristics, meta-heuristics, ...) for black box optimization

  no or almost no assumptions on the objective function

- They are typically less efficient than problem-specific (exact) algorithms (in terms of #funevals)

  less differences in the continuous case (as we have seen)

- Allow for an easy and rapid implementation and therefore to find good solutions fast

  easy to incorporate (and recommended!) to incorporate problem-specific knowledge to improve the algorithm