Evolutionary Computation

2022/23

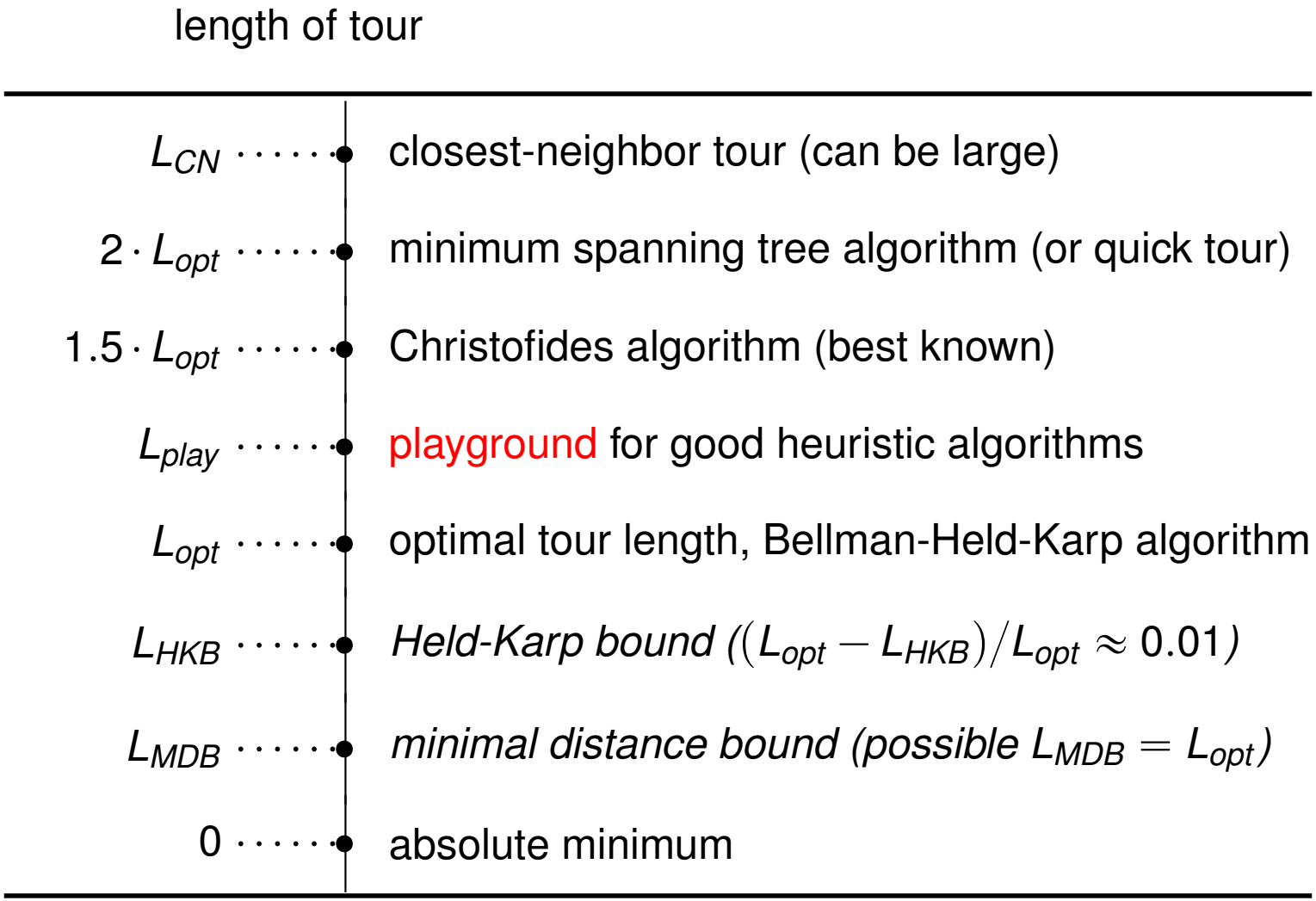
Master Artificial Intelligence

Arno Formella

Departamento de Informática
Escola Superior de Enxeñaría Informática
Universidade de Vigo

22/23

the TSP length-line sum-up



additional information

additional information and benchmark instances can be found at:

- http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/
- almost a counterexample of how to implement GA for TSP https://jaketae.github.io/study/genetic-algorithm/
- we use the work at https://github.com/guofei9987/scikit-opt

the sorting problem

- Sorting is a basic, well known, and well studied problem.
- Given a sequence of n elements belonging to an orderable set, we have to compute a permutation of the input elements such that they are ordered (according to the underlying compare function).
- Simple example: given a sequence of integer numbers; sort ascending.
- Note that there are $n!$ possible permutations.
  (Funny, the same number as there are tours in TSP.)

checking a solution

- Before doing the actual sorting, let's first design an algorithm that checks the results, i.e., checks that the sequence is sorted.
- Remember: we require that we can check with an algorithm that the output/result of our initial algorithm is correct (i.e., fulfills the corresponding properties).
- Personally, I recommend that you always try to design and implement such a checker!

checker for the sorting problem

Check whether a pair of elements is sorted:

```
def PairIsSorted(v,i,j):
  return v[i]<=v[j]
```

Check whether a sequence of elements is sorted:

```
def IsSorted(v):
  for i in range(len(v)-1):
    if not PairIsSorted(v,i,i+1):
      return False
  return True
```

the bubble sort algorithm

A simple sorting algorithm:

```
def BubbleSort(v):
  while not IsSorted(v):
    for i in range(len(v)-1):
      PairSort(v,i,i+1)
```

Runs in quadratic time (worst case) and linear time (best case).

Monte Carlo sorting

Let us implement a Monte Carlo sorting algorithm:
we select a random pair of elements and interchange when necessary:

```
def MonteCarloSort(v,rounds):
  for j in range(rounds):
    i,j=RandomPair(v)
    PairSort(v,i,j)
```

Whenever the number of rounds is sufficiently large and we are lucky the sequence will become sorted.

Las Vegas sorting

Let us implement a Las Vegas sorting algorithm: we select a random pair of elements, interchange when necessary, and stop when the sequence is sorted:

```
def LasVegasSort(v):
  while not IsSorted(v):
    i,j=RandomPair(v)
    PairSort(v,i,j)
```

Maybe we need to wait a lot of time, but we always get a sorted sequence. Observe: Las Vegas algorithms are easy to design, when we have a checker!

Las Vegas Monte Carlo sorting

Whenever we have a checker, we can implement a Las Vegas algorithm on the base of a Monte Carlo algorithm, so for sorting we can do:

```
def LasVegasMonteCarloSort(v,rounds):
  while not IsSorted(v):
    MonteCarloSort(v,rounds)
```

Monte Carlo sort with Las Vegas condition

We can improve the Monte Carlo sort introducing a Las Vegas condition to stop earlier:

```
def MonteCarloLasVegasSort(v,rounds):
  while not IsSorted(v) and rounds>0:
    i,j=RandomPair(v)
    PairSort(v,i,j)
    rounds-=1
```

This idea reflects the general structure of a heuristic probabilistic algorithm: for a certain time do something maybe useful, and stop when a certain condition is met.

Efficient sorting

An efficient $O(n \log n)$ algorithm is the merge sort algorithm, here written in its iterative form (divide and conquer paradigm):

```
def Merge(v,w,left,middle,right):
  i,j=left,middle
  for k in range(left,right):
    if j>=right or (i<middle and PairIsSorted(v,i,j)):
      w[k]=v[i]; i+=1
    else:
      w[k]=v[j]; j+=1

def MergeSort(w):
  s,n=1,len(w)
  while s<n:
    v=w[:]
    for left in range(0,n,2*s):
      Merge(v,w,left,left+s,min(left+2*s,n))
    s*=2
```

sorting as an optimization problem

- In order to state the integer sorting problem as an optimization problem, we need to specify an objective function.
- Let $x = (x_1, x_2, \ldots, x_n)$ be the current sequence of integer values.
- We use $f(x) = \sum_{i=1}^{n} i \cdot x_i$ as objective function.
- Our aim is to maximize $f(x)$ at which point the sequence x is sorted; to minimize we take the negative value:

```
def SortObjective(v):
  f=0
  for i in range(len(v)):
    f+=v[i]*(i+1)
  return -f
```

or

```
def SortObjective(v):
  return -sum([v[i]*(i+1) for i in range(len(v))])
```

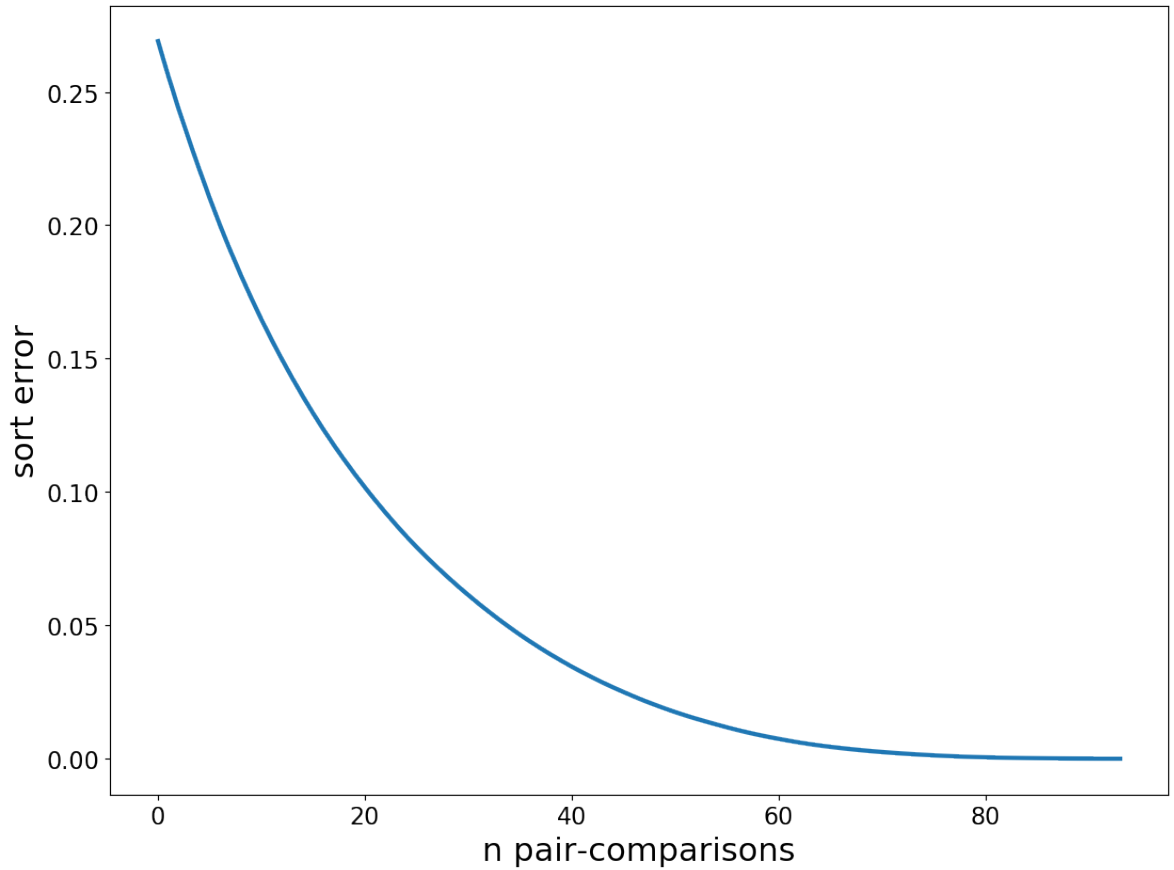
sorting as an optimization problem

Now, we can use a genetic algorithm for the TSP problem to sort our sequence (note, we want an order on the cities, but now with our objective function for sorting and not the one for a minimal tour length).

```
def GASort(w):
  ga=GA_TSP(
    func=fobj,n_dim=len(w),size_pop=100,
    max_iter=1000,prob_mut=1
  )
  best_points,best_val=ga.run()
  v=w.copy()
  for i in range(len(best_points)):
    w[i]=v[int(best_points[i])]
```

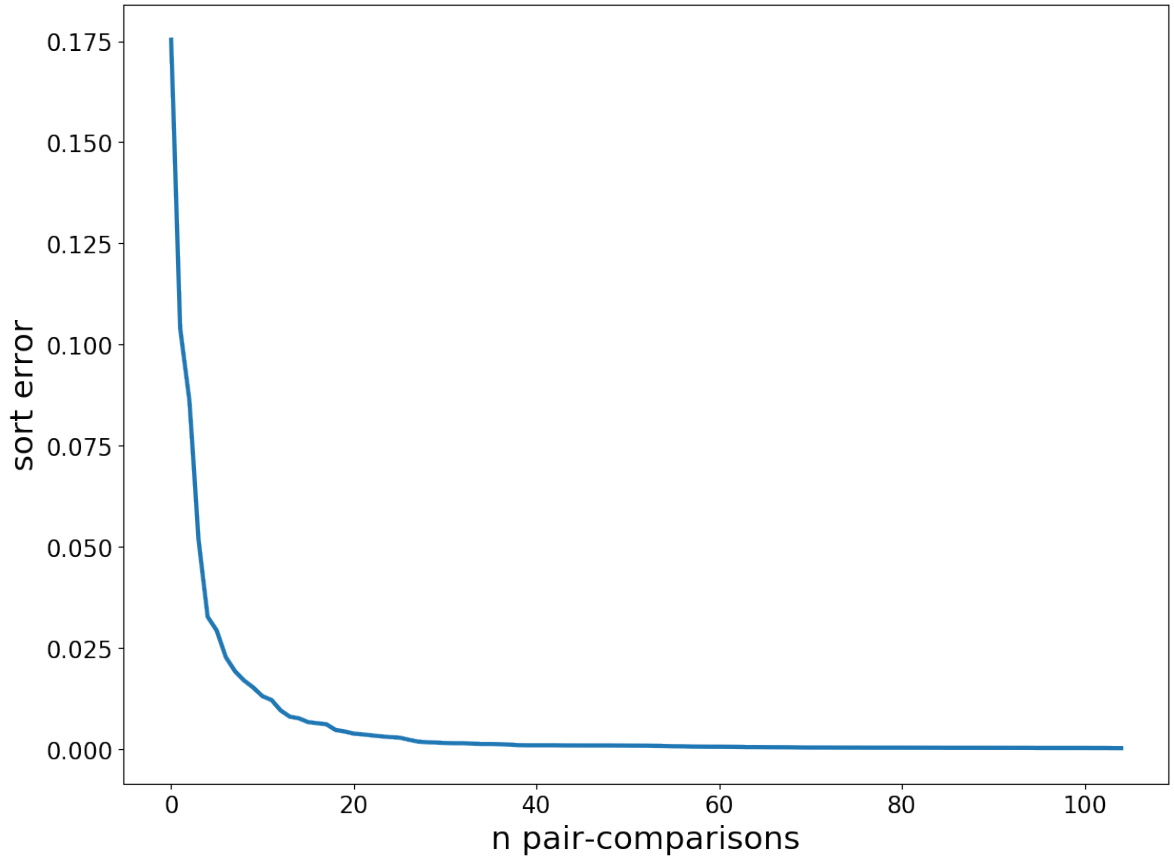
More in lab hours (`fobj` will be computed on a different data structure).

convergence of bubble sort



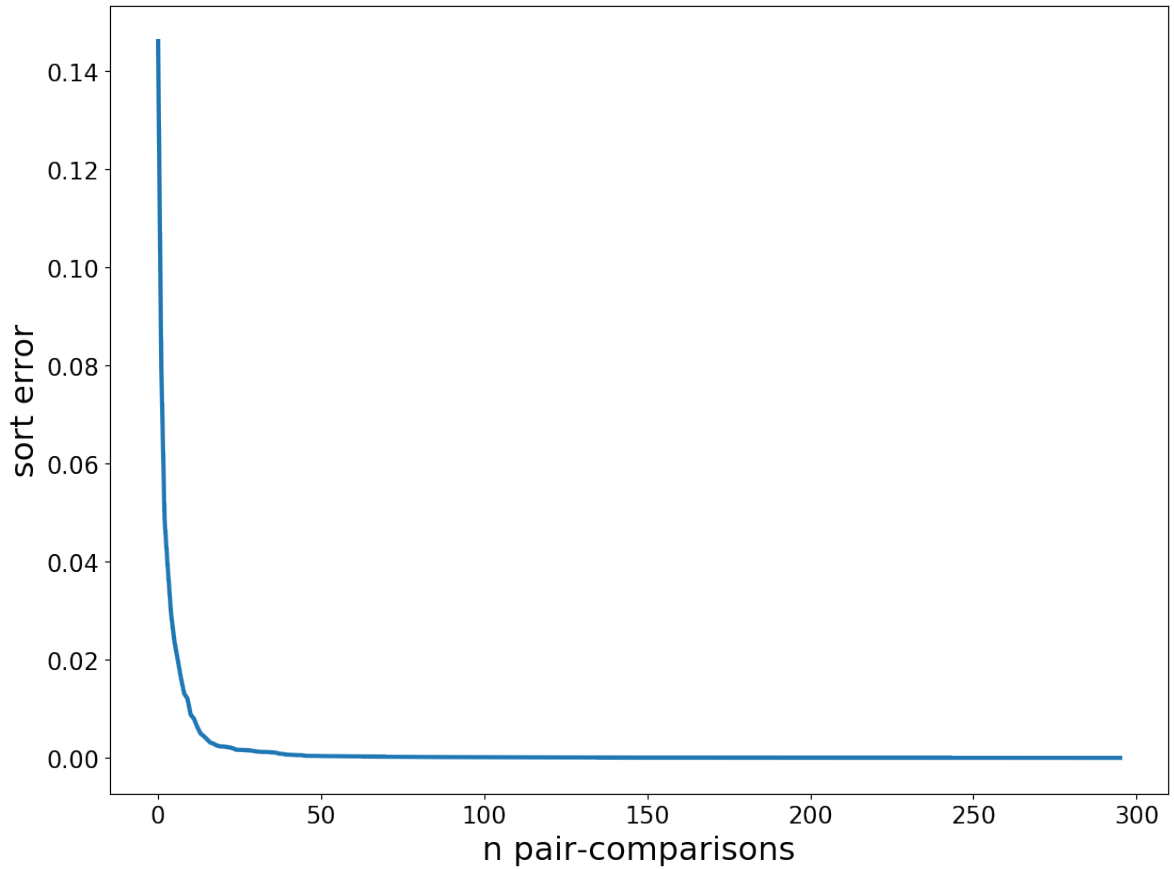
Slow improvement, finds the minimum always.

convergence of Monte Carlo sort



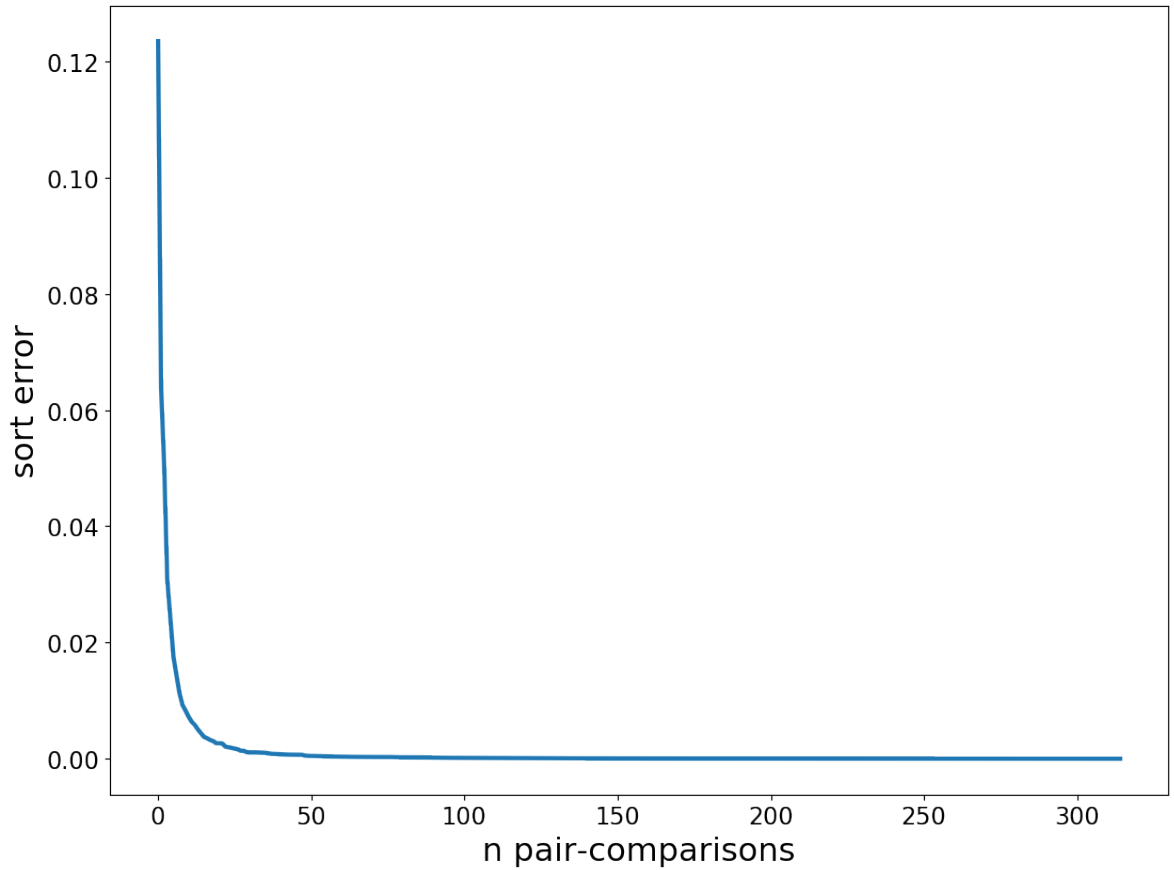
Fast improvement, fixed number of steps, might not find minimum.

convergence of Las Vegas sort

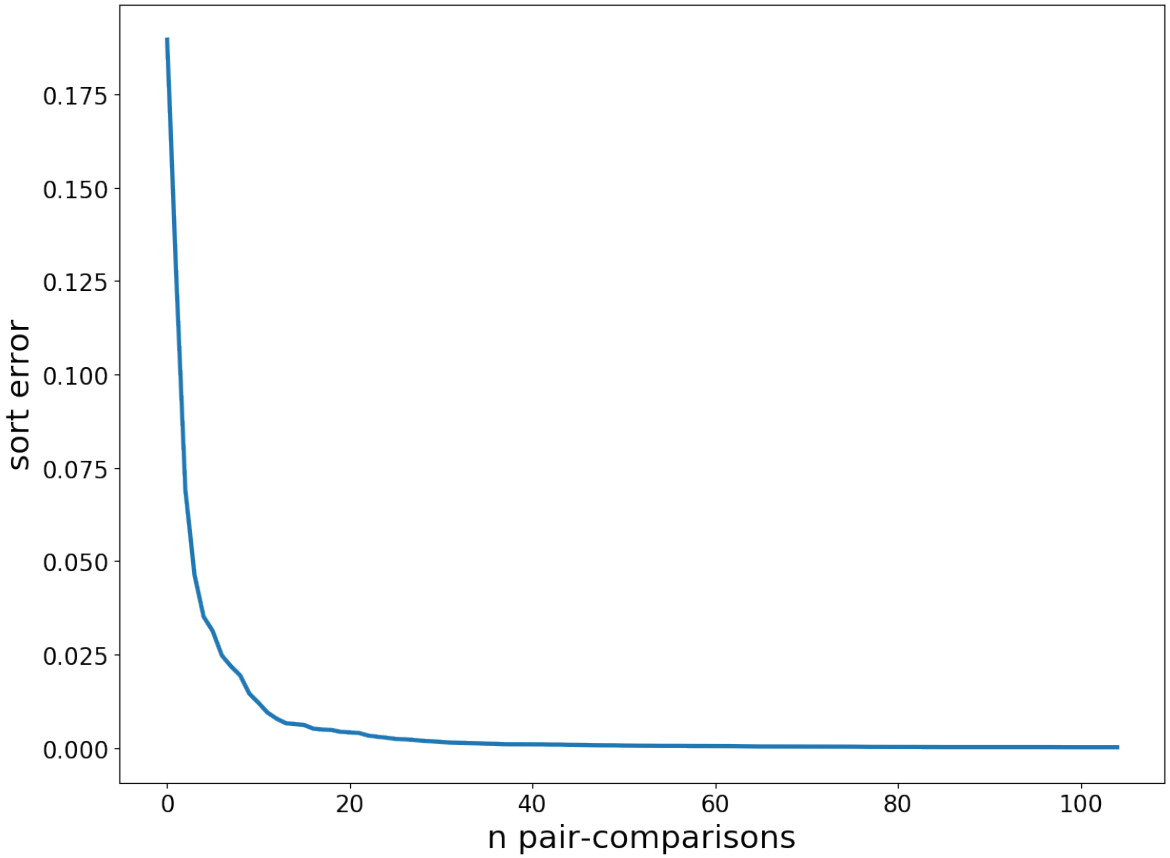


Fast improvement at the beginning, and slowly finds the minimum.

convergence of Las Vegas with Monte Carlo sort

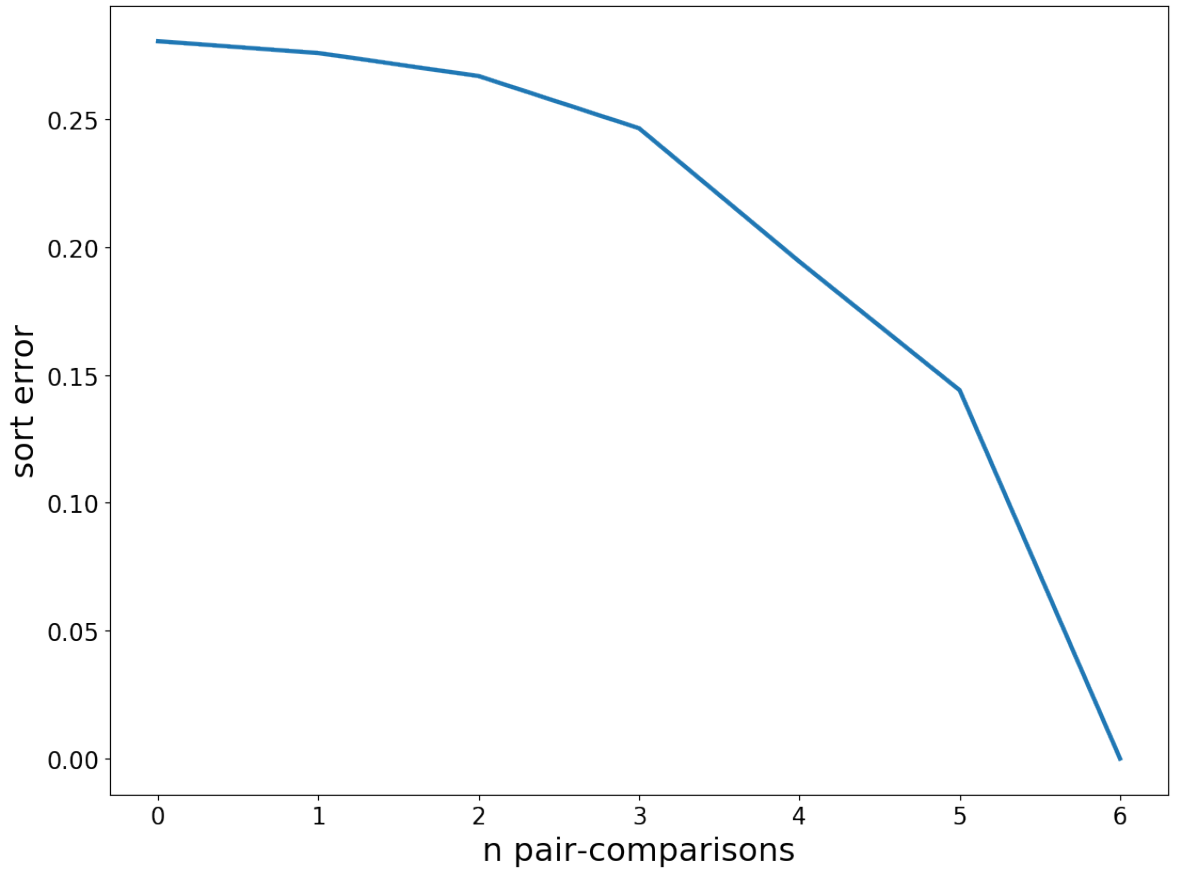


convergence of Monte Carlo sort with Las Vegas condition



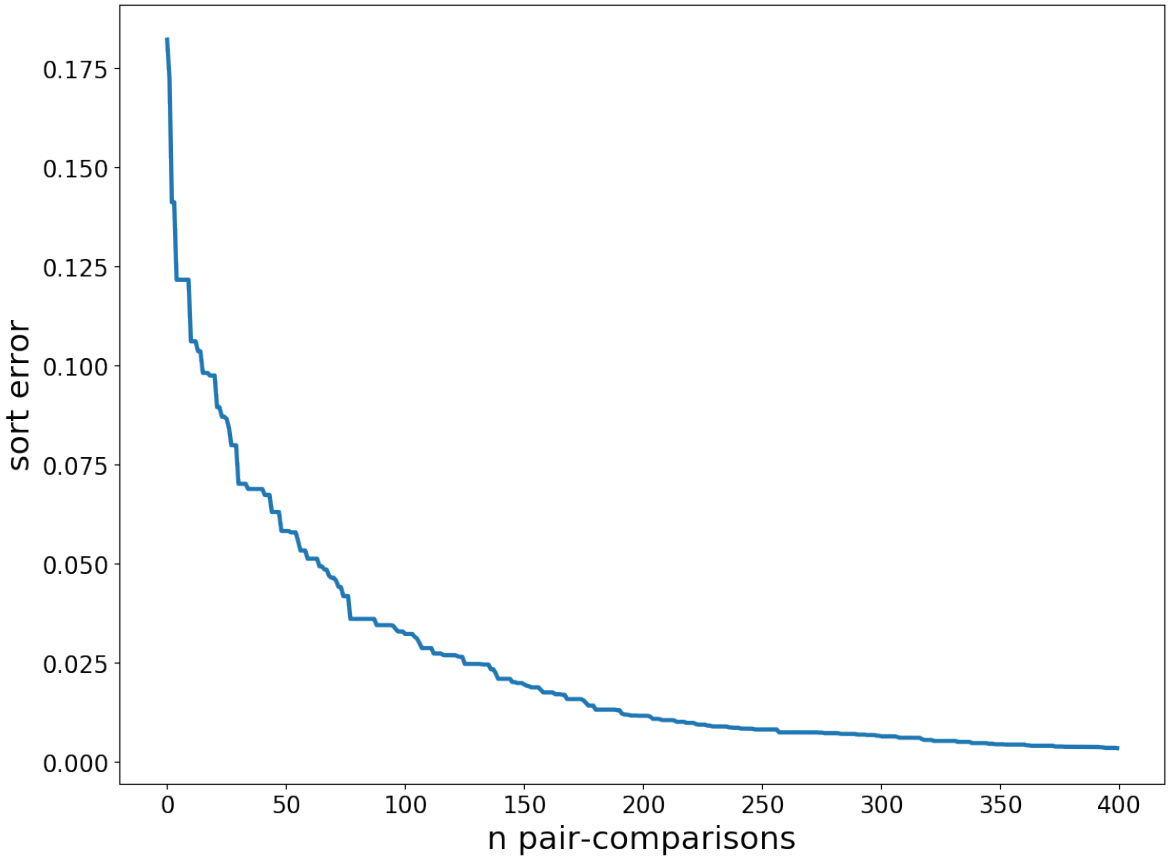
Fast improvement, but might not find minimum (however, stops if found).

convergence of merge sort



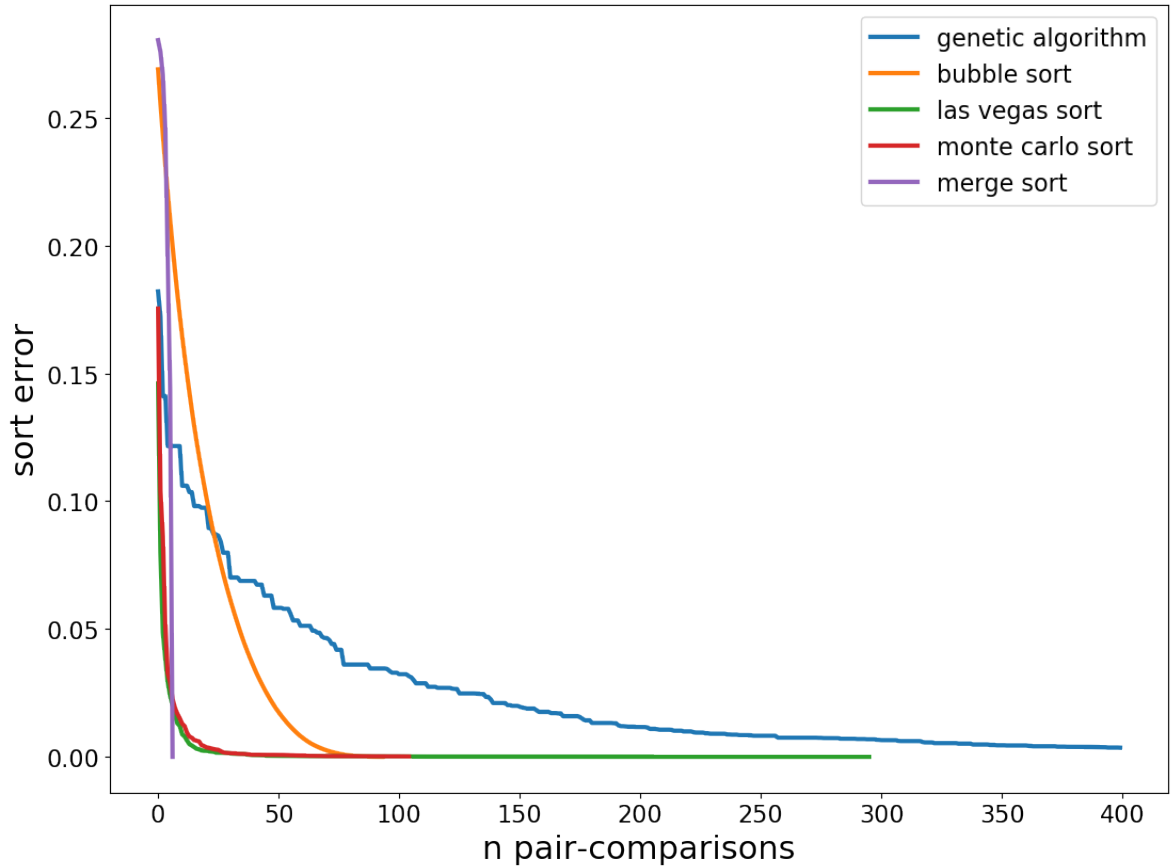
Deterministic very fast improvement, finds the minimum always!

convergence of genetic algorithm sort



Well, works, but might not find the minimum.

summary of *convergence* for sorting algorithms



Maybe the genetic algorithm is not the right choice,
better stick to the deterministic classic one.

Can we sort faster?

You can always ask: can we sort faster?

- It depends... when we have more information about the data, maybe we can sort faster!
- In the given example, we started with a random permutation of n consecutive numbers.
- So sorting them is easy: just count—starting at the minimum—up to n, hence, a linear time algorithm!
- It's always worthwhile to analyse the underlying data!

interpretation of the results

Don't get betrayed by a small number of program runs that might even *suggest* some good results (both in precision as well as in runtime). You should always ask to see several/many runs, and to determine the

variance of the results, so that you can compute the Monte Carlo standard error.

The (0,1)-knapsack problem

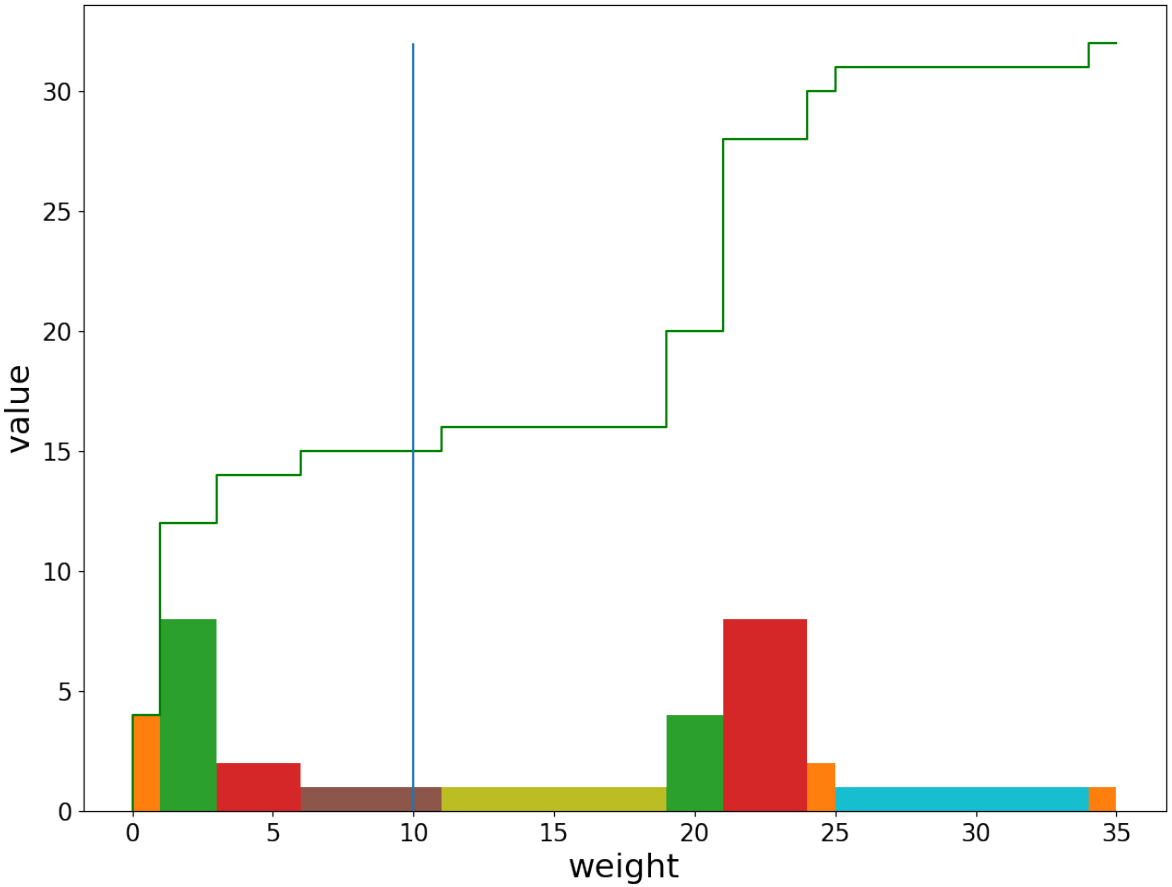
The (0,1)-knapsack problem (KSP) is another classical combinatorial optimization problem, where

- Given a set of items, each with a certain weight and value, and
- given a knapsack with a certain weight capacity,
- find the maximum total value you can carry with the knapsack.

Note that in this problem (in comparison to TSP or sorting) we have infeasible combinations (i.e., the subset might be too heavy).

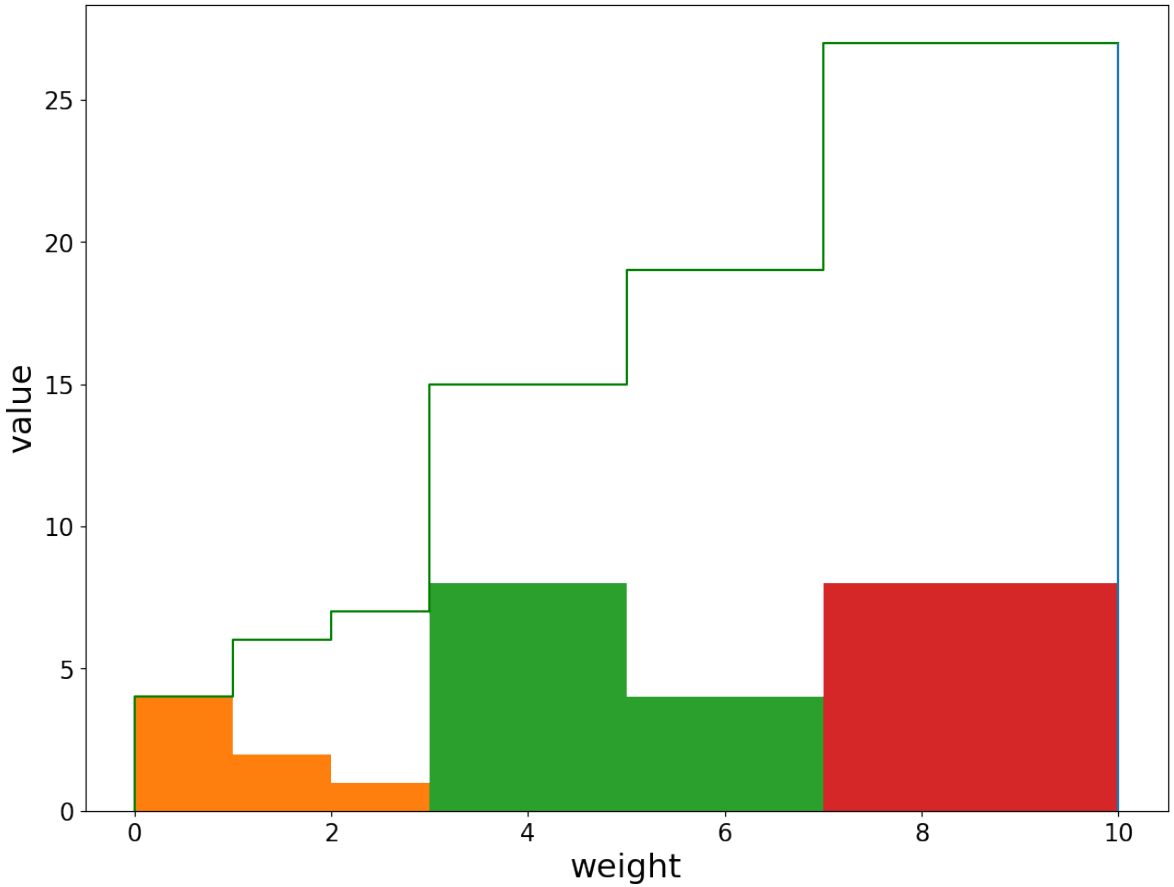
an example knapsack problem

If we take all available items:



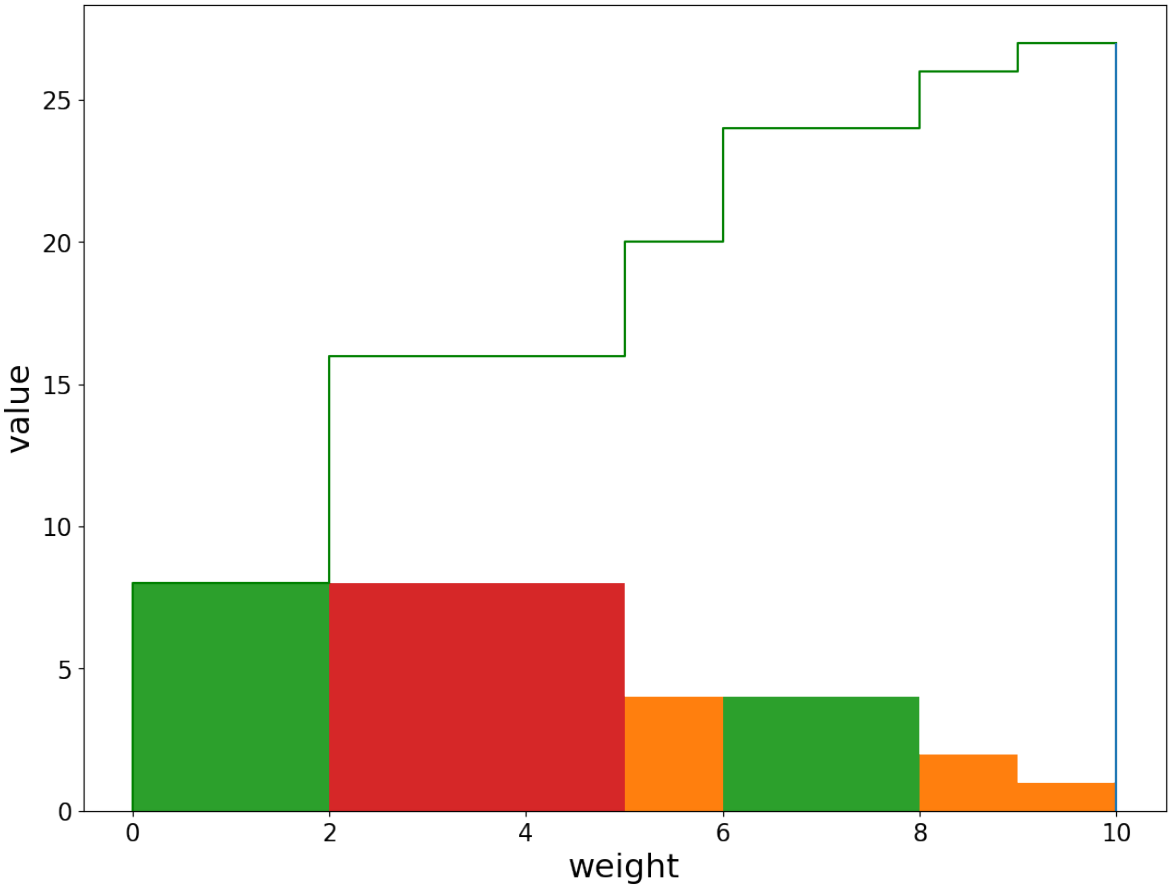
packing the knapsack with greedy weight algorithm

We take the lightest items as long as they fit:



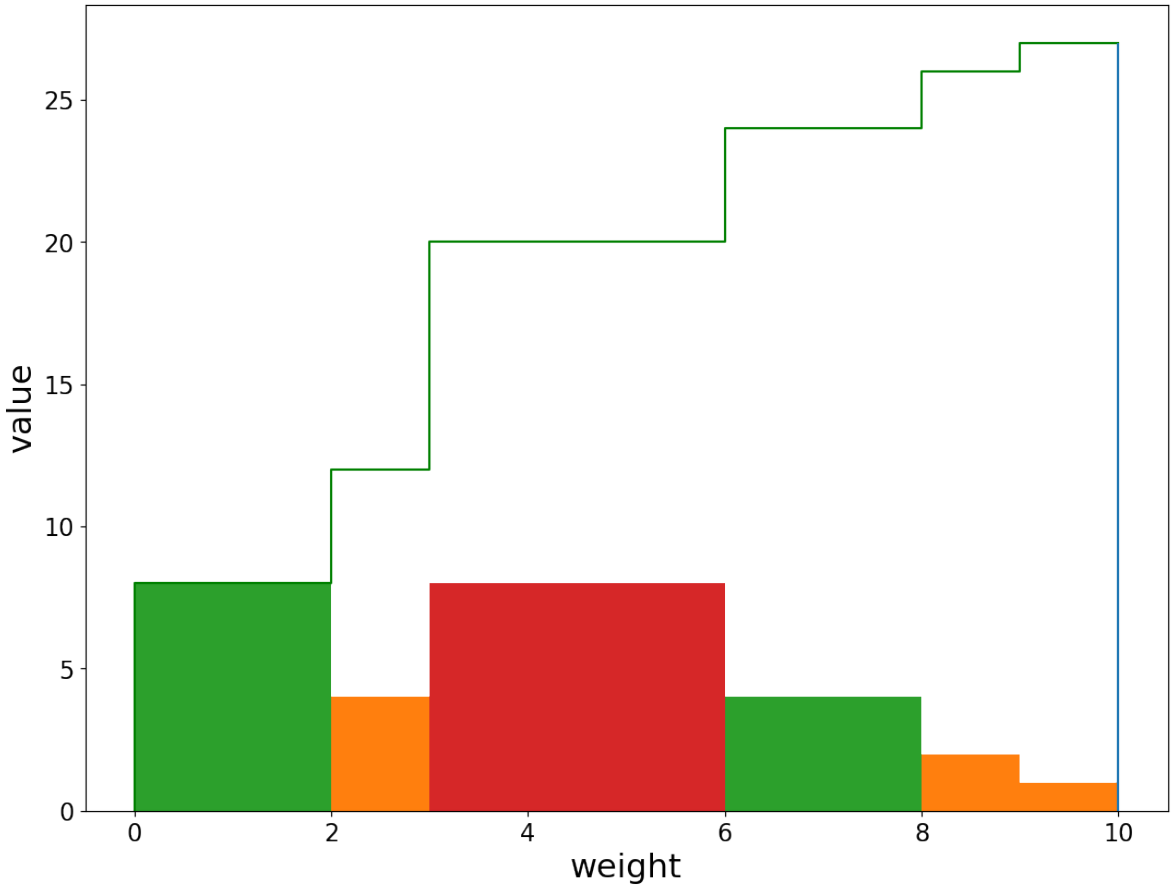
packing the knapsack with greedy value algorithm

We take the most valued items as long as they fit:



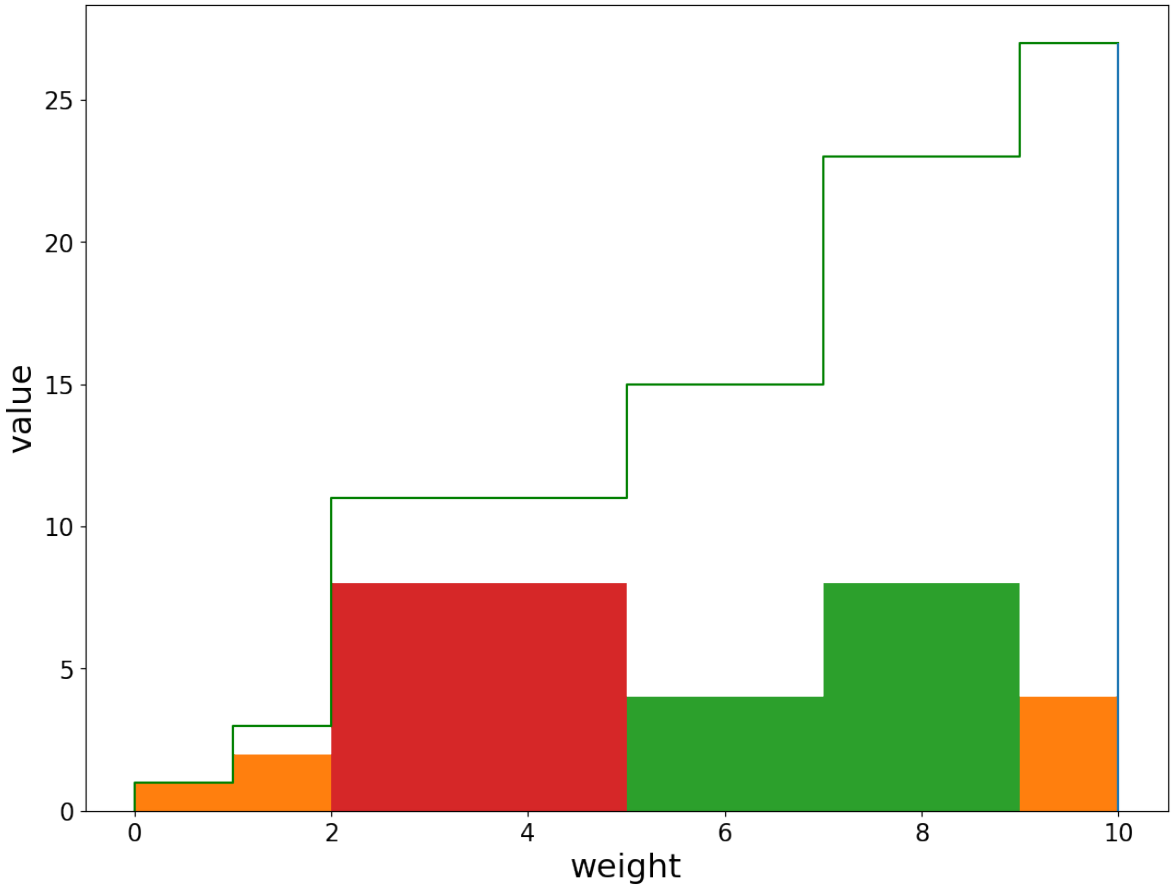
packing the knapsack with greedy ratio algorithm

We take the best rated (value per weight unit) items as long as they fit:



optimal packing the knapsack with dynamic programming

We find the optimal solution with dynamic programming:



It seems all algorithms are great?

The previous algorithms all packed a value of 27 into the knapsack...

- You noticed that I have cheated?
- All algorithms found an optimal packaging!
- You know why?
- I was lucky.

parallelization

- Monte Carlo algorithms, and hence, evolutionary algorithms are often quite easy to parallelize.
- We will not talk about parallelization in this course, however, it's an important issue in order to achieve performance on modern systems.

Evolutive methods

- Evolutive methods work with populations of individuals (or only one individual and a certain type of memory).
- There are probabilistic modification processes (mutation, reproduction, recombination/crossover) that change the population from one to the next generation.
- The performance of the individuals is based on a fitness which usually is the objective function (but not necessarily).
- There is a selection process to maintain a (more or less) stable state (size) of the population.
- Most of the algorithmic decisions are drawn probabilistically.

I will not give details on the history and researchers, please, take a look at the literature/bibliography.

Genetic algorithms (GA)

- We distinguish the genotype (codification of the individuals) and the phenotype (elements of the search space).
- There must exist a bijection between genotype and phenotype.
- The genotype encodes the free parameters of an individual.
- The modifications (mutation and recombination/crossover) are carried out over the genotype.
- The fitness is evaluated over the phenotype (our objective function).
- We have to explain: codification (of the genotype), initialization, mutation, recombination/crossover, selection, and stopping.

GA: principal loop

A genetic algorithm can be summarized in the following principal loop:

```
InitializePopulation()          # initialization
EvaluateIndividuals()           # evaluation
while not Stopping():           # stopping
  DetermineParents()              # selection
  GenerateChildren()              # recombination
  MutateChildren()                # mutation
  EvaluateIndividuals()           # evaluation
  ReestablishPopulation()         # selection
```

GA: encoding of the individuals

There are many possibilities how to encode the free parameters of an individual to form its genotype:

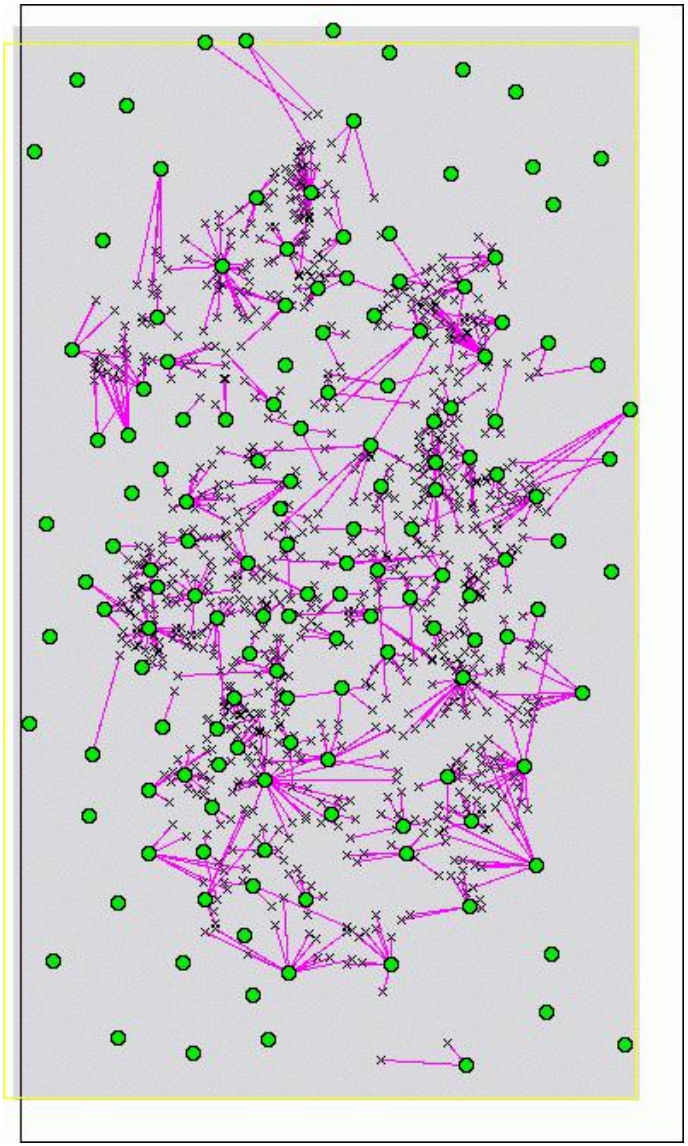
- use a binary bitstring, e.g., (101101)
- use a sequence of integer values in certain ranges, e.g., $(2, 6, 98, 3) \in [1:2] \times [1:10] \times [0:100] \times [1:5]$
- use a sequence of real values in certain ranges, e.g., $(1.23, 34.4, -2.1) \in [-50.0, 50.0]$
- use a permutation
- use a k-dimensional structure
- use a binary tree
- use a general graph
- use whatever you like (remember: do something, be happy...)

Remember: we need a bijection between genotype and phenotype and we need to implement crossovers and mutations that are able to explore the entire search space (or at least the region of interest).

GA: names taken from biology

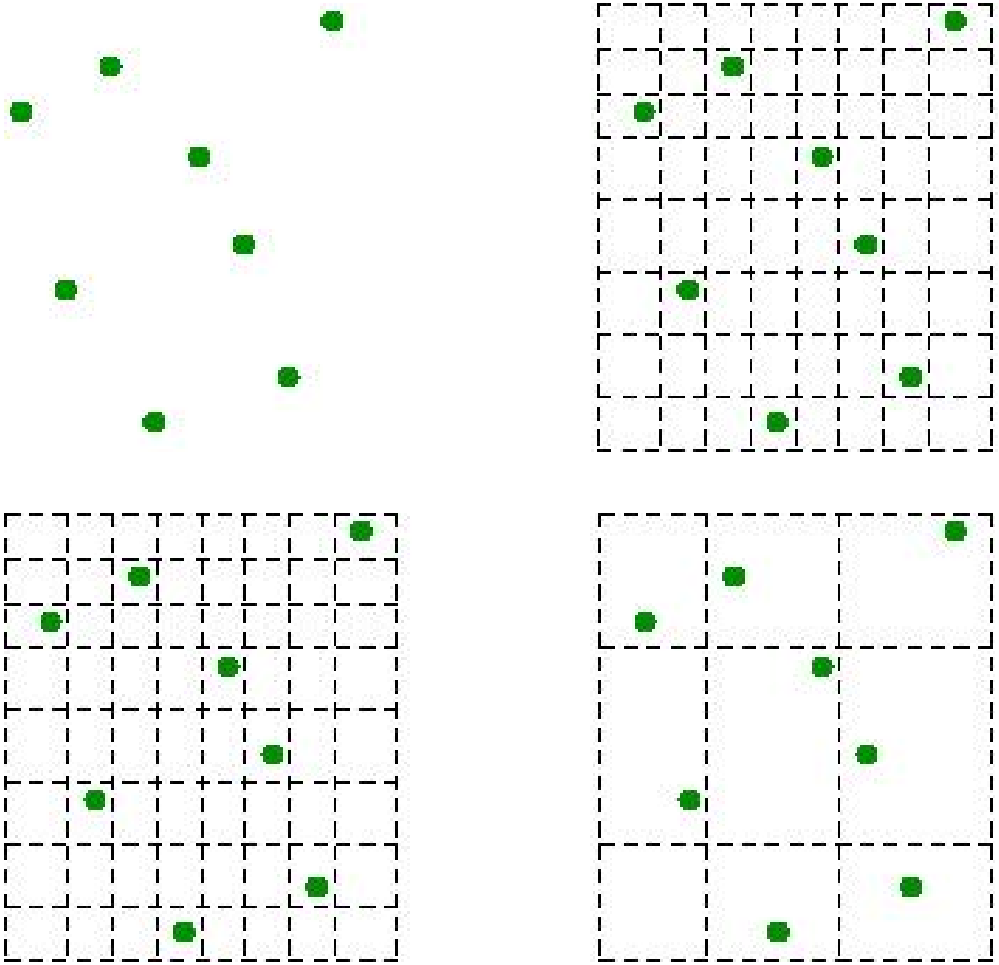
- The individial components of the sequences are called genes.
- The possible values of a gene are called allele.
- The encoding of an individual is called its genome or chromosome.

GA: genotype an example



- green: base stations
- crosses: mobile users
- magenta: assignment
- goal: find the minimal subset of base stations that guarantees an assignment of all mobiles
- Note: computation of the objective function is quite complex (and will not be detailed here).

GA: genotype an example



- initial
  8×8 grid
- reduced to
  3×3 grid
- 4 allele
  (2-bit strings)
  - unusable
  - used
  - unused
  - fixed

GA: mutation possiblities

gene mutation:
: just change one (or more) genes to another permitted allele

gene flip:
: interchange the values of two genes

gene sequence displacement:
: cut a sequence and insert at another position

gene sequence inversion:
: revert the order of a (partial) sequence

what-ever-you-like:
: do something, be happy...

GA: mutation rules of thumb

- The mutation rate should be inversely proportional to the size of the genome.
- For larger populations maybe reduce mutation rate in the on-going optimization process.

GA: crossover possiblities

simple crossover:

parents	cut	children
(101101)	$(10\|1101)$	$\longrightarrow (100111)$
(010111)	$(01\|0111)$	$\longrightarrow (011101)$
$(2,8,98,3)$	$(2,8,\|98,3)$	$\longrightarrow (2,8,40,4)$
$(1,9,40,4)$	$(1,9,\|40,4)$	$\longrightarrow (1,9,98,3)$

k-point crossover:

cut at k points and interchange the corresponding parts

(variation: take k at random)

uniform crossover:

interchange each gene with certain probability

multiple parent mating:

use more then two parents and interchange genes

(variation: merge entire parent set)

GA: crossover possiblities (continued)

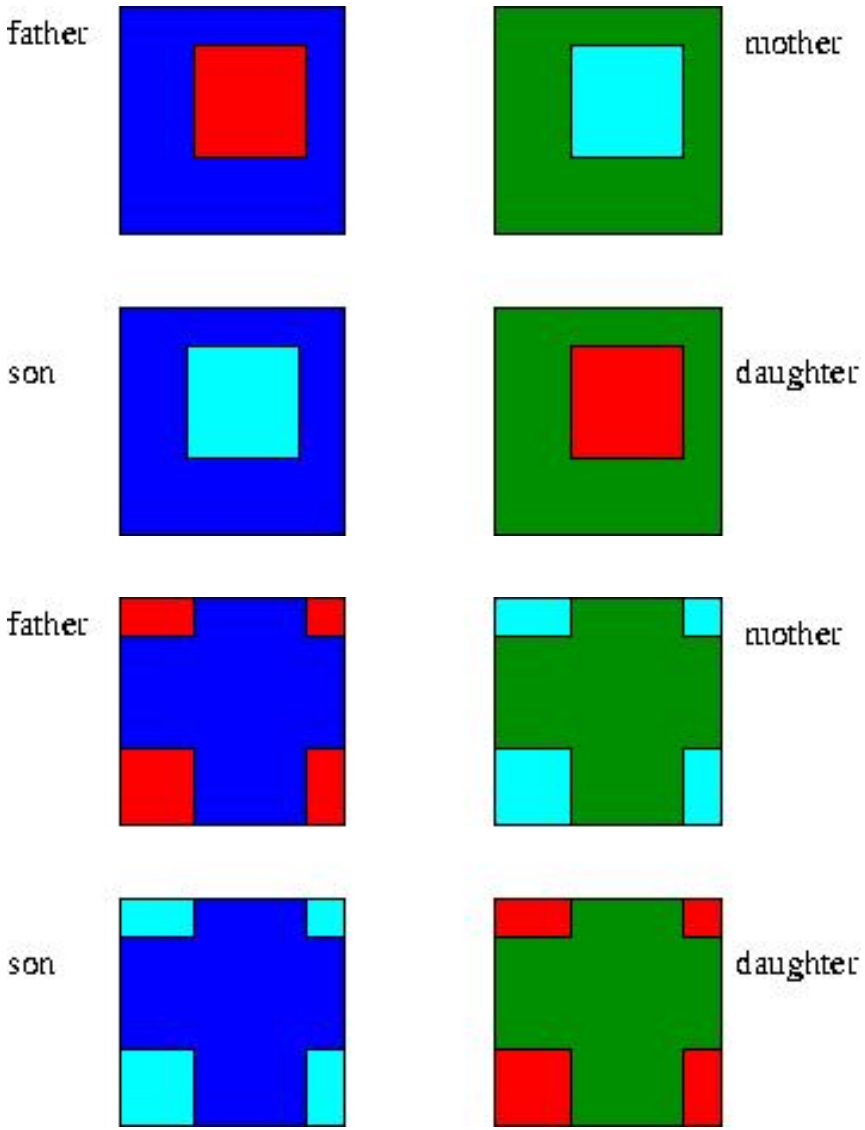
arithmetic crossover: assign to children linear combination of parent genes with some random weight, $\alpha \in [0, 1]$, e.g., with $\alpha = 0.7$ on second gene:
$(1.23, 34.5, -2.1) \longrightarrow (1.23, 28.2, -2.1)$
$(10.5, 13.5, 23.1) \longrightarrow (1.23, 19.8, 23.1)$
(again variations: as k-point, or with all genes, or with k at random)

blended crossover: blend two corresponding parent genes with a certain, usually fixed, value $\alpha \in [-0.5, \infty]$ according to the current gene spread

simulated binary crossover: blend two corresponding parent genes according to a suitable probability density function

what-ever-you like: remember, do something, be happy...

GA: crossover example (2-point cyclic crossover)



- select two grid points
- interchange rectangles

GA: principal loop

A genetic algorithm can be summarized in the following principal loop:

```
InitializePopulation()      # initialization
EvaluateIndividuals()       # evaluation          DONE
while not Stopping():       # stopping
  DetermineParents()          # selection
  GenerateChildren()          # recombination     DONE
  MutateChildren()            # mutation          DONE
  EvaluateIndividuals()       # evaluation        DONE
  ReestablishPopulation()     # selection
```