Evolutionary Computation

2025/26

Master Artificial Intelligence

Arno Formella

Departamento de Informática
Escola Superior de Enxeñaría Informática
Universidade de Vigo

25/26

How does a genetic algorithm work?

A genetic algorithm is a bio-inspired probabilistic algorithm:

- initialize a set of individuals
- while *stopping criterium* not met
  - evaluate fitness of the individuals (in search space)
  - generate off-springs (mutation and crossover, in the encoding space)
  - generate a new generation, i.e., a subset of parents plus off-springs (selection)
- report best individual generated in the process

You'll work with this approach in the lab hours.

additional information

additional information and benchmark instances can be found at:

- http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/ or
- https://www.math.uwaterloo.ca/tsp/data/index.html
- 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

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 is an important issue in order to achieve good performance on modern systems.

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!
- As already stated: sometimes the properties of the output when using an evolutionary algorithm can be checked only very weakly.
- For TSP at least check: you got a tour, the gap lies inbetween known lower and upper bounds.

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
```

checker help

- besides the fact that a checker helps you to debug your program
- checkers are often easier to implement than the solver
- and you can use the checker to implement an exhaustive search algorithm (for finite search spaces):
- just enumerate the elements of the search space, check all, keep the best.

the bubble sort algorithm

A simple sorting algorithm (bubble sort):

```
def PairSort(v,i,j):  # sorts a pair
  if not PairIsSorted(v,i,j):
    v[i],v[j]=v[j],v[i]

def BubbleSort(v):    # LasVegas type
  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 very long 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

Merge sort is an efficient $O(n \log n)$ algorithm to sort, here 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 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))])
```

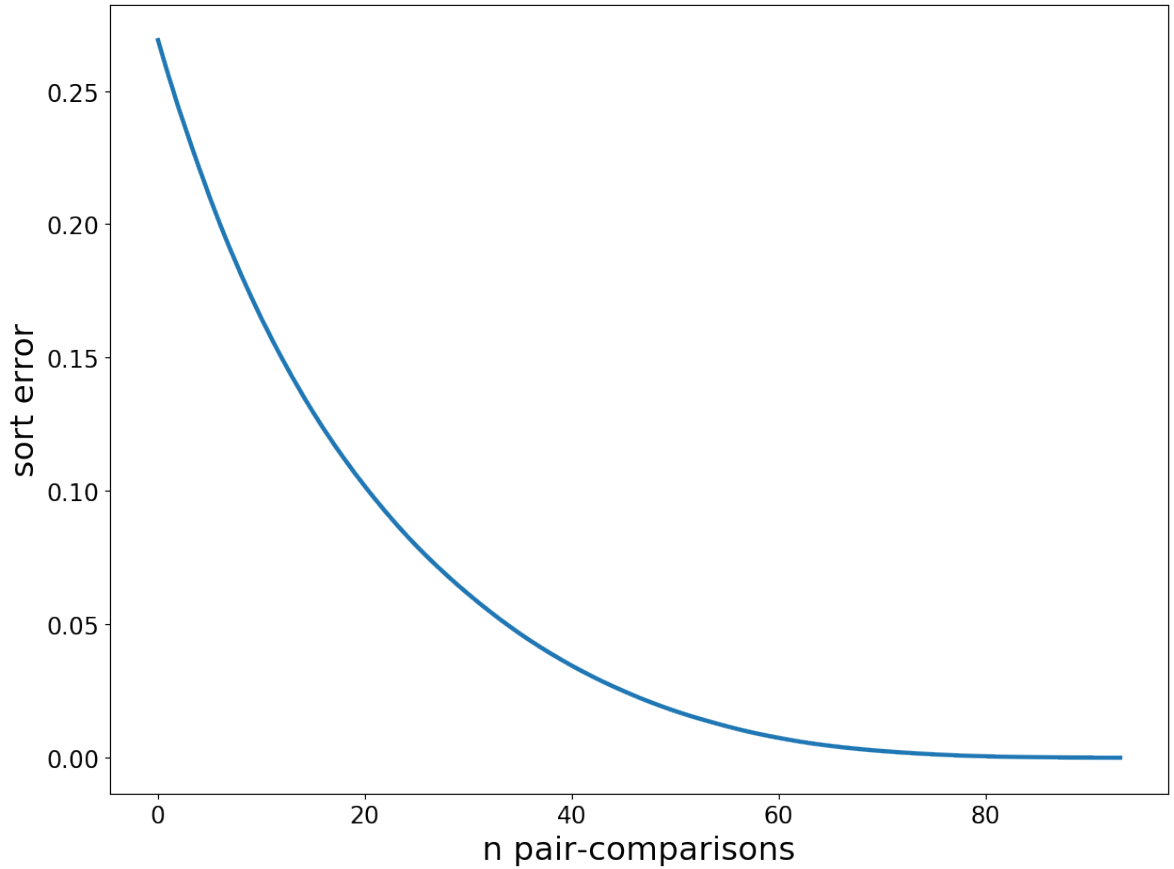
sorting as an optimization problem

Now, we can use a genetic algorithm (e.g. the one we have 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])]
```

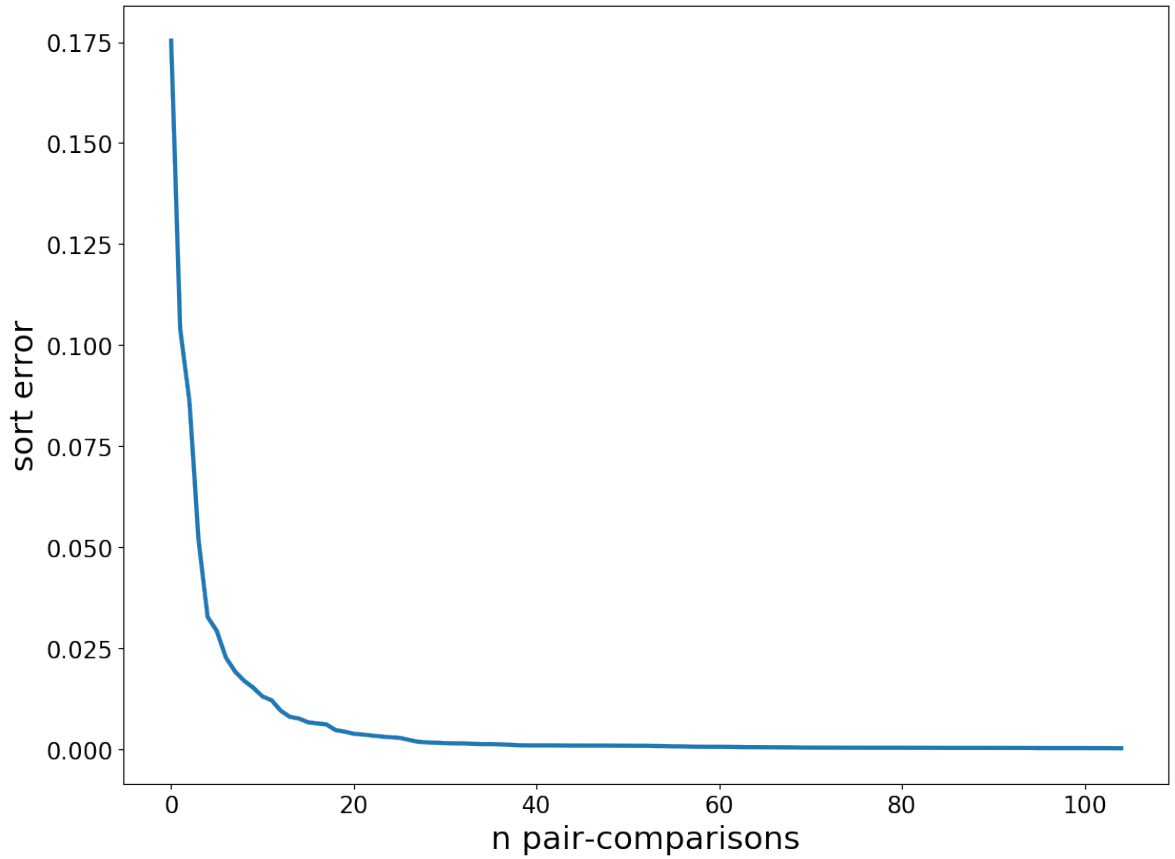
More in lab hours (`fobj` will be computed on a different data structure).

convergence of bubble sort



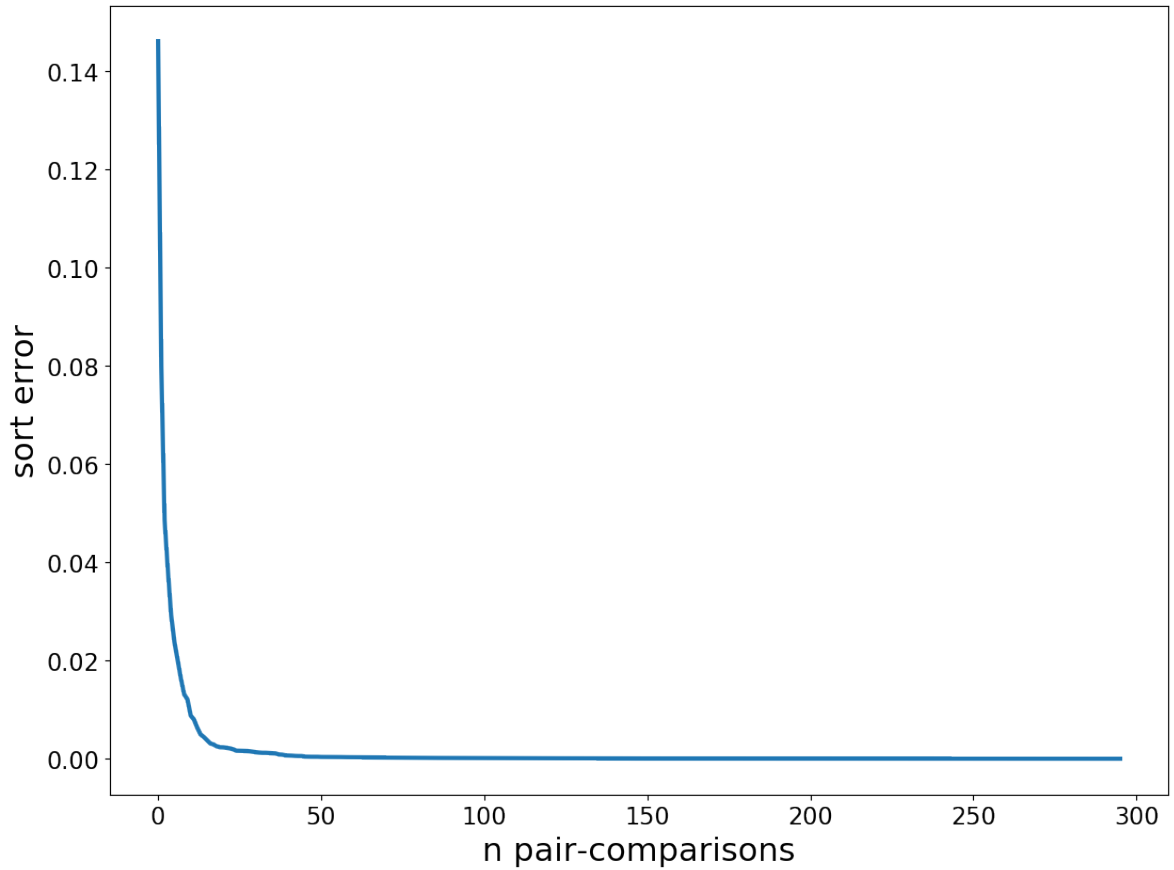
Slow improvement, finds the minimum always.

convergence of Monte Carlo sort



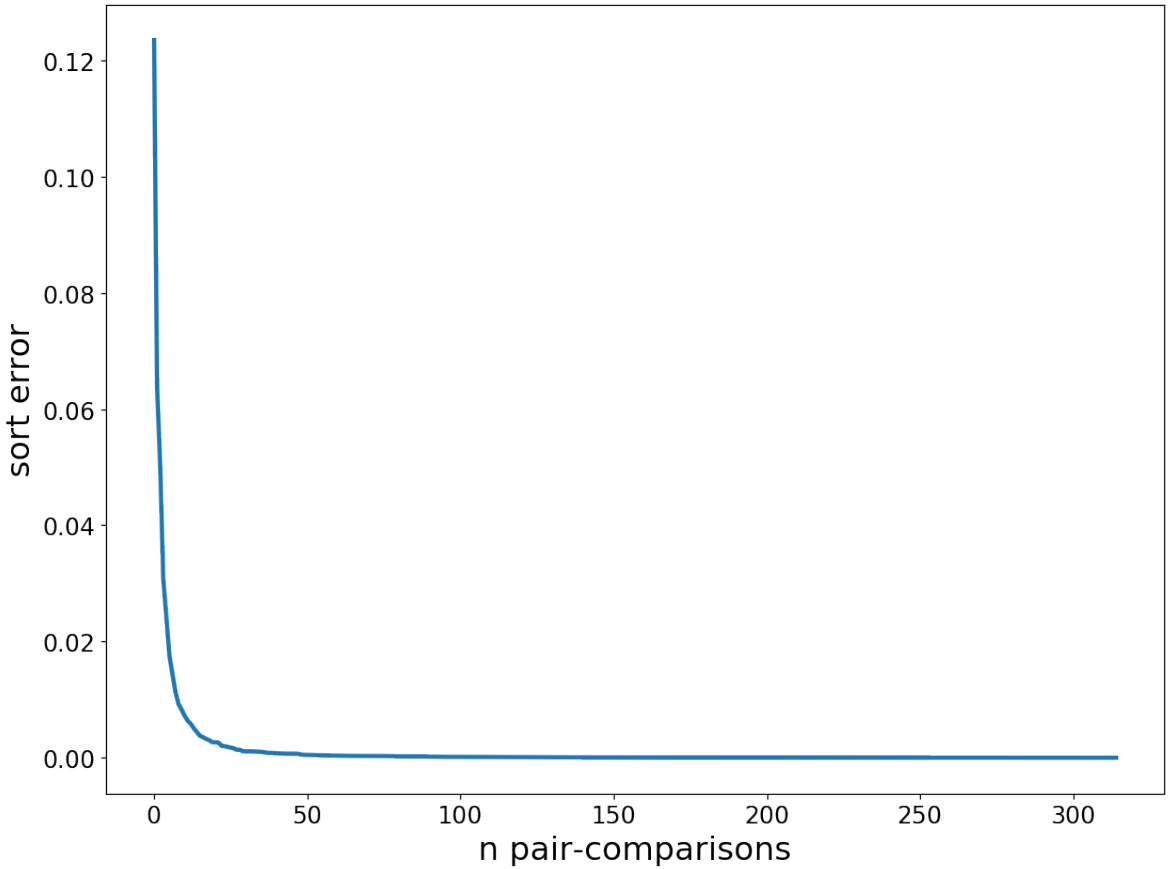
Fast improvement, fixed number of steps, might not find minimum.

convergence of Las Vegas sort

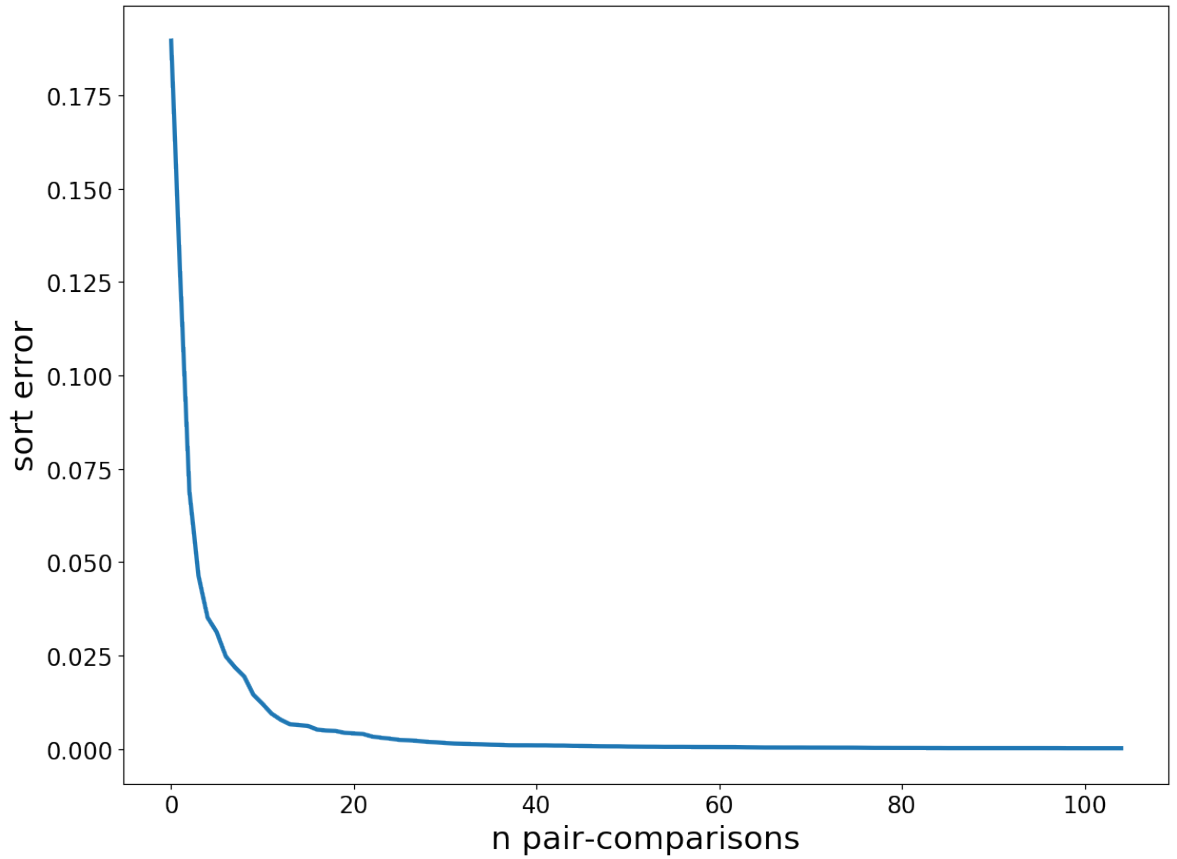


Fast improvement at the beginning, and slowly finds the minimum.

convergence of Las Vegas with Monte Carlo sort

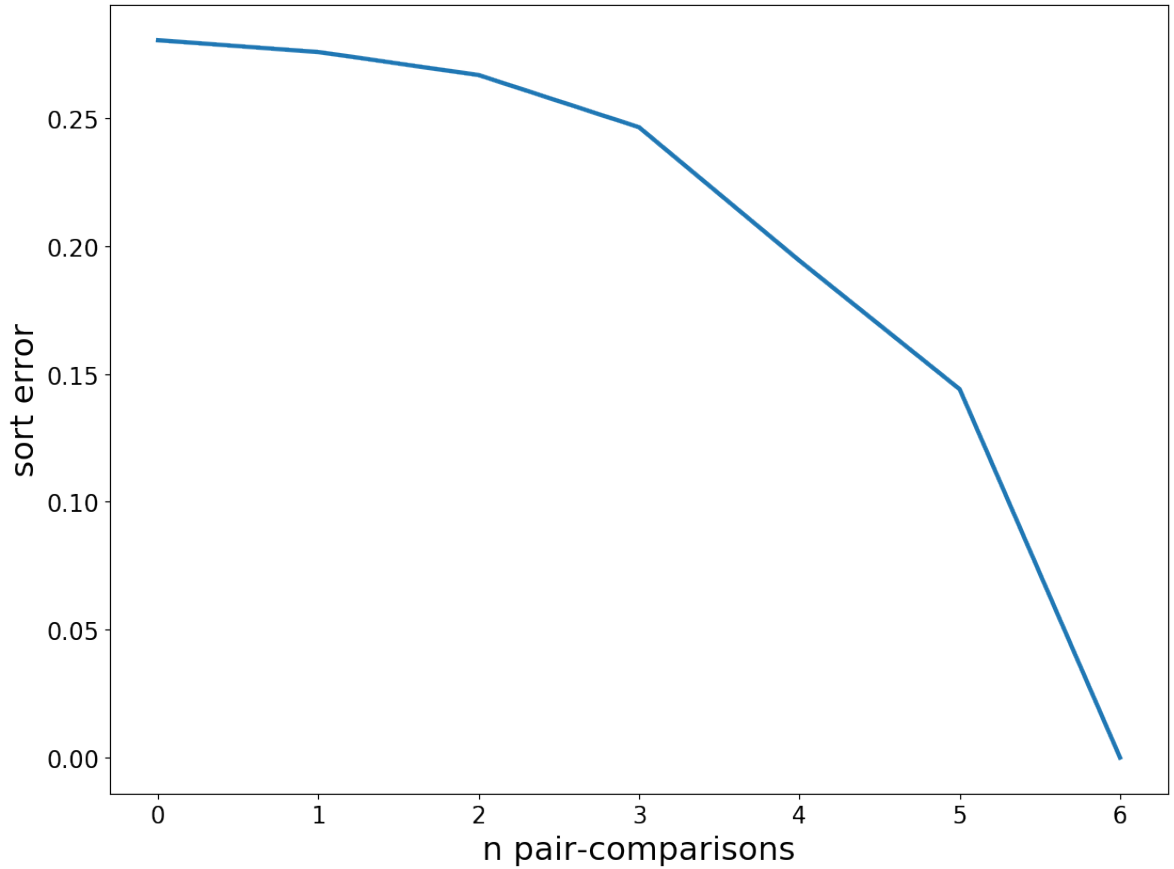


convergence of Monte Carlo sort with Las Vegas condition



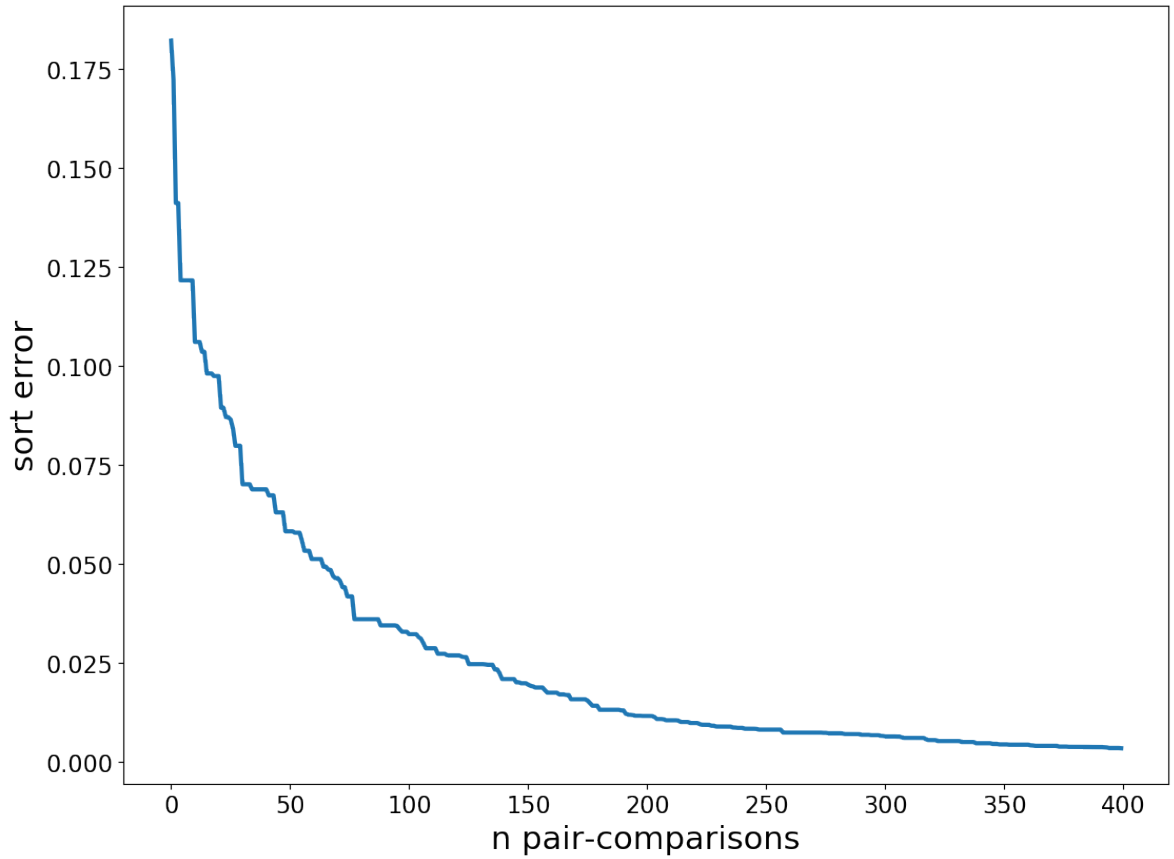
Fast improvement, but might not find minimum (however, stops if found).

convergence of merge sort



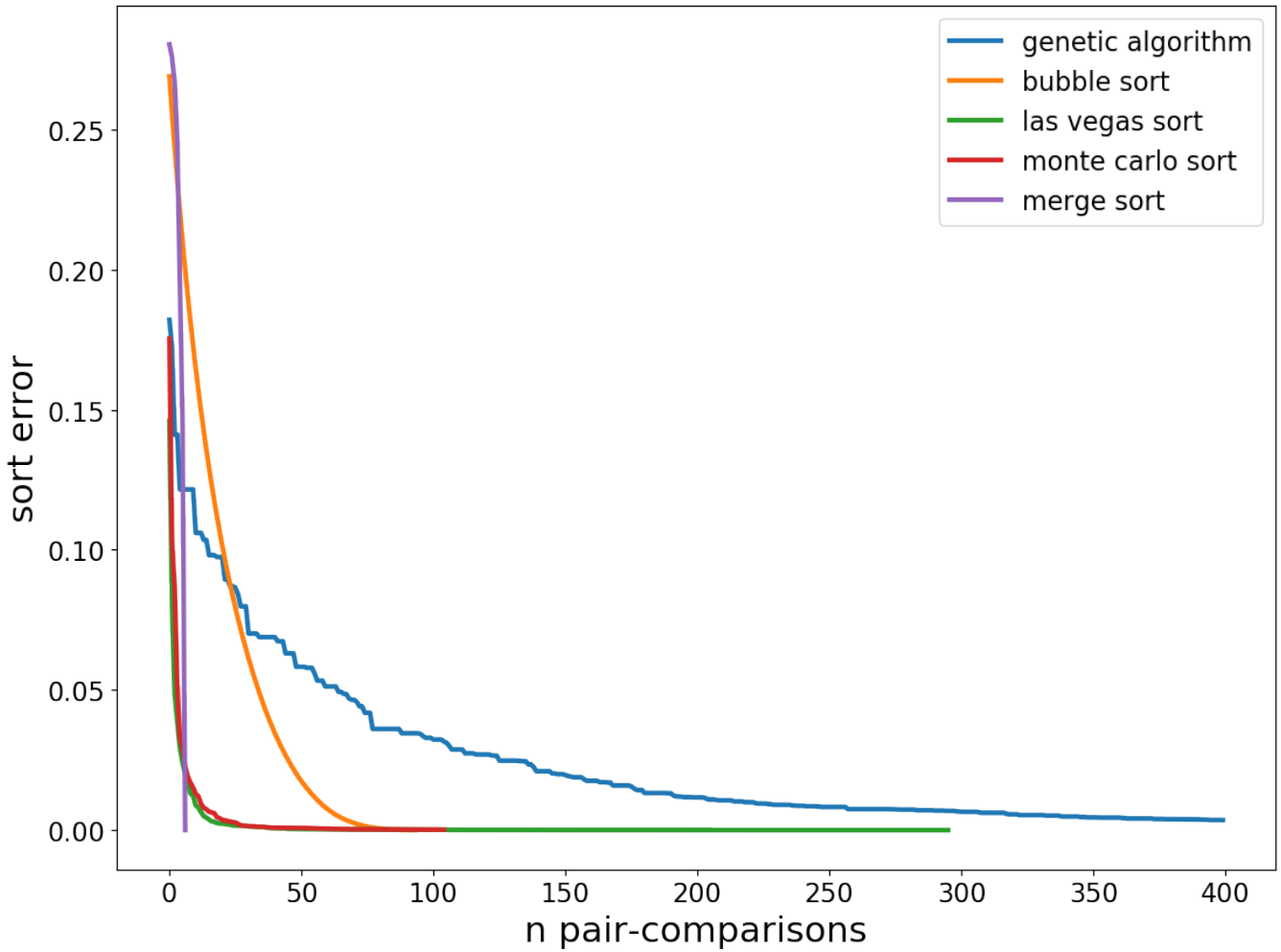
Deterministic very fast improvement, finds the minimum always!

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.