Metaheuristics for Smart Manufacturing

5. Evolutionary Algorithms

Thomas Weise · 汤卫思

tweise@hfuu.edu.cn · http://iao.hfuu.edu.cn

Hefei University, South Campus 2
Faculty of Computer Science and Technology
Institute of Applied Optimization
230601 Shushan District, Hefei, Anhui, China
Econ. & Tech. Devel. Zone, Jinxiu Dadao 99

合肥学院 南艳湖校区/南2区
计算机科学与技术系
应用优化研究所
中国 安徽省 合肥市 蜀山区 230601
经济技术开发区 锦绣大道99号

Outline

The slides are available at http://iao.hfuu.edu.cn/155, the book at http://thomasweise.github.io/aitoa, and the source code at http://www.github.com/thomasWeise/aitoa-code

course book

course material

An Introduction to Optimization Algorithms

The contents of this course are available as free electronic book *"An Introduction to Optimization Algorithms"* [1] at http://thomasweise.github.io/aitoa in pdf, html, azw3, and epub format, created with our bookbuildeR tool chain.

Section Outline

1. Introduction
2. Algorithm Concept: Population
3. Experiment and Analysis: Population
4. Algorithm Concept: Binary Operator
5. Experiment and Analysis: Binary Operator

Introduction

- Hill Climbers are local search.

- Hill Climbers are local search.
- They begin at some point x in the search space and then investigate its neighborhood $N(x)$.

- Hill Climbers are local search.
- They begin at some point x in the search space and then investigate its neighborhood $N(x)$.
- The neighborhood is defined by the (unary) search operator, in our case `1swap`.

Introduction

- Hill Climbers are local search.
- They begin at some point x in the search space and then investigate its neighborhood $N(x)$.
- The neighborhood is defined by the (unary) search operator, in our case `1swap`.
- If they reach a local optimum, they are trapped.

- Hill Climbers are local search.
- They begin at some point x in the search space and then investigate its neighborhood $N(x)$.
- The neighborhood is defined by the (unary) search operator, in our case `1swap`.
- If they reach a local optimum, they are trapped.
- We then can restart them, but this means
  1. to start again at "0"

Introduction

- Hill Climbers are local search.
- They begin at some point x in the search space and then investigate its neighborhood $N(x)$.
- The neighborhood is defined by the (unary) search operator, in our case `1swap`.
- If they reach a local optimum, they are trapped.
- We then can restart them, but this means
  1. to start again at "0" and
  2. they may still land again in a local optimum.

- Hill Climbers are local search.
- They begin at some point x in the search space and then investigate its neighborhood $N(x)$.
- The neighborhood is defined by the (unary) search operator, in our case `1swap`.
- If they reach a local optimum, they are trapped.
- We then can restart them, but this means
  1. to start again at "0" and
  2. they may still land again in a local optimum.
- Idea: Why not investigate multiple points in the search space at once and use the additional information in a clever way?

- Population-based metaheuristics [2–7] try to maintain a set of points in the search space which are iteratively refined.

- Population-based metaheuristics [2–7] try to maintain a set of points in the search space which are iteratively refined.
- This has a couple of advantages

- Population-based metaheuristics [2–7] try to maintain a set of points in the search space which are iteratively refined.
- This has a couple of advantages:
  - we are less likely to get trapped in a local optimum.

- Population-based metaheuristics [2–7] try to maintain a set of points in the search space which are iteratively refined.
- This has a couple of advantages:
  - we are less likely to get trapped in a local optimum.
  - we are more likely to find a better (local) optimum.

- Population-based metaheuristics [2–7] try to maintain a set of points in the search space which are iteratively refined.
- This has a couple of advantages:
  - we are less likely to get trapped in a local optimum.
  - we are more likely to find a better (local) optimum.
  - if we have different good points from the search space in our population, we can try to use this additional information...

Section Outline

- Evolutionary Algorithms (EAs) are the most successful family of population-based metaheuristics. [2, 4–6]

- Evolutionary Algorithms (EAs) are the most successful family of population-based metaheuristics. [2, 4–6]
- Here we focus on $(\mu+\lambda)$ EAs

- Evolutionary Algorithms (EAs) are the most successful family of population-based metaheuristics. [2, 4–6]
- Here we focus on $(\mu+\lambda)$ EAs, which work as follows:
  1. Generate a population of $\mu+\lambda$ random points in the search space (and map them to solutions and evaluate them).

- Evolutionary Algorithms (EAs) are the most successful family of population-based metaheuristics. [2, 4–6]
- Here we focus on $(\mu + \lambda)$ EAs, which work as follows:
  1. Generate a population of $\mu + \lambda$ random points in the search space (and map them to solutions and evaluate them).
  2. From the population, select the μ best points as "parents" for the next "generation", discard the remaining λ points.

$(\mu + \lambda)$ EA

- Evolutionary Algorithms (EAs) are the most successful family of population-based metaheuristics. [2, 4–6]
- Here we focus on $(\mu + \lambda)$ EAs, which work as follows:
  1. Generate a population of $\mu + \lambda$ random points in the search space (and map them to solutions and evaluate them).
  2. From the population, select the μ best points as "parents" for the next "generation", discard the remaining λ points.
  3. Generate λ new "offspring" points by applying a unary search operator (which creates a randomly modified copy from a selected point).

$(\mu + \lambda)$ EA

- Evolutionary Algorithms (EAs) are the most successful family of population-based metaheuristics. [2, 4–6]
- Here we focus on $(\mu + \lambda)$ EAs, which work as follows:
  1. Generate a population of $\mu + \lambda$ random points in the search space (and map them to solutions and evaluate them).
  2. From the population, select the μ best points as "parents" for the next "generation", discard the remaining λ points.
  3. Generate λ new "offspring" points by applying a unary search operator (which creates a randomly modified copy from a selected point).
  4. Evaluate the λ offsprings, add them to the population, and go back to step 2.

Section Outline

- The hill climber had no parameters.

- The hill climber had no parameters.
- The hill climber with restarts had 2, namely the starting steps until restarts and their increment (256 and 5% in case of `hcr_256+5%_1swap`, respectively).

- The hill climber had no parameters.
- The hill climber with restarts had 2, namely the starting steps until restarts and their increment (256 and 5% in case of `hcr_256+5%_1swap`, respectively).
- Our basic $(\mu + \lambda)$ EA also has two, namely μ and λ.

Setup

- The hill climber had no parameters.
- The hill climber with restarts had 2, namely the starting steps until restarts and their increment (256 and 5% in case of `hcr_256+5%_1swap`, respectively).
- Our basic $(\mu + \lambda)$ EA also has two, namely μ and λ.
- Let us choose $\mu = \lambda$ and test the two values $\mu = \lambda = 2048$ and $\mu = \lambda = 4096$.

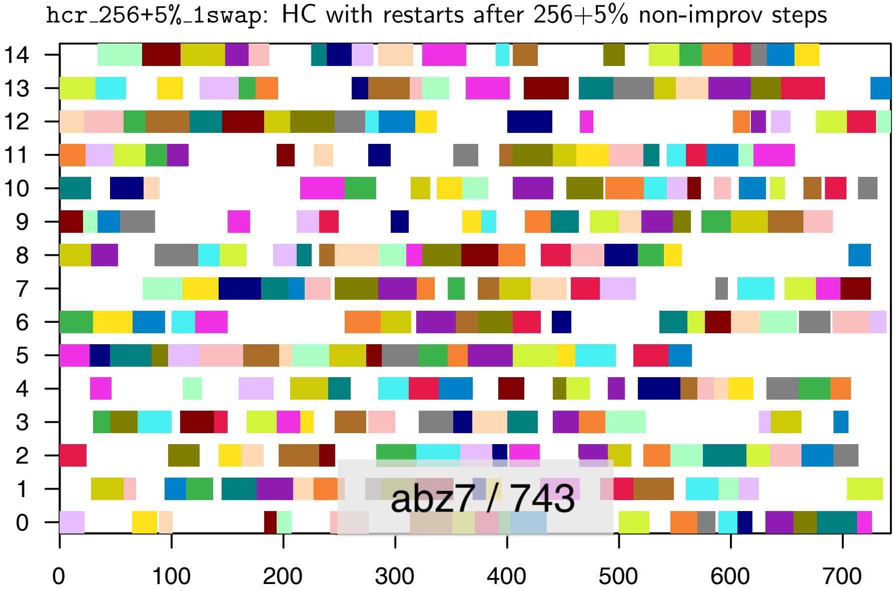
So what do we get?

- I execute the program 101 times for each of the datasets `abz7`, `la24`, `swv15`, and `yn4`

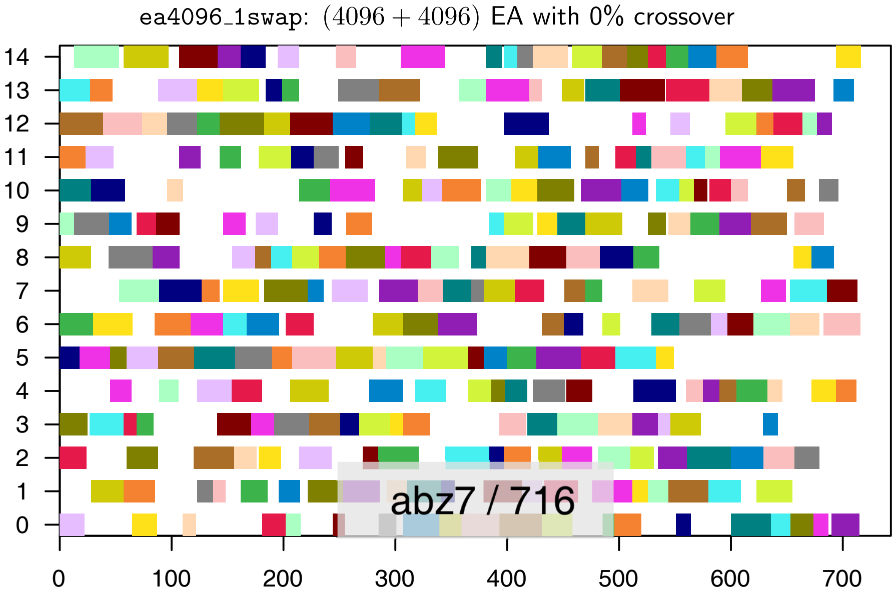
- I execute the program 101 times for each of the datasets abz7, la24, swv15, and yn4

		makespan				last improvement	
$\mathcal{I}$	algo	best	mean	med	sd	med(t)	med(FEs)
abz7	hcr_256+5%_1swap	723	742	743	**7**	21s	5'681'591
	ea2048_1swap	695	719	718	13	11s	2'581'614
	ea4096_1swap	**688**	**716**	**716**	12	19s	4'416'129
la24	hcr_256+5%_1swap	970	997	998	**9**	6s	3'470'368
	ea2048_1swap	945	983	983	16	2s	927'000
	ea4096_1swap	**941**	**980**	**978**	14	5s	1'897'387
swv15	hcr_256+5%_1swap	3701	3850	3857	**40**	60s	9'874'102
	ea2048_1swap	**3395**	3535	**3530**	78	128s	19'290'521
	ea4096_1swap	3397	**3533**	3533	54	171s	25'073'630
yn4	hcr_256+5%_1swap	1095	1129	1130	**14**	22s	4'676'669
	ea2048_1swap	1032	1082	1082	22	26s	4'792'622
	ea4096_1swap	**1020**	**1076**	**1074**	21	39s	6'907'692

So what do we get?

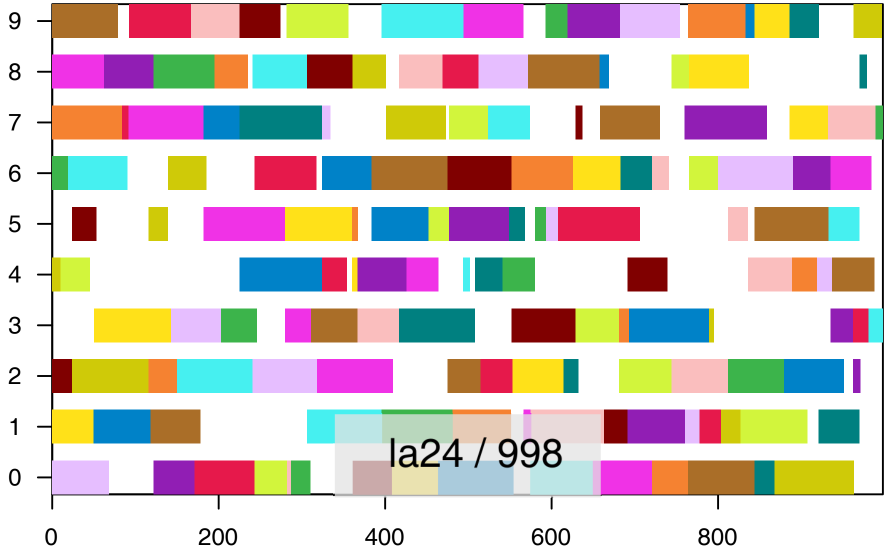


So what do we get?

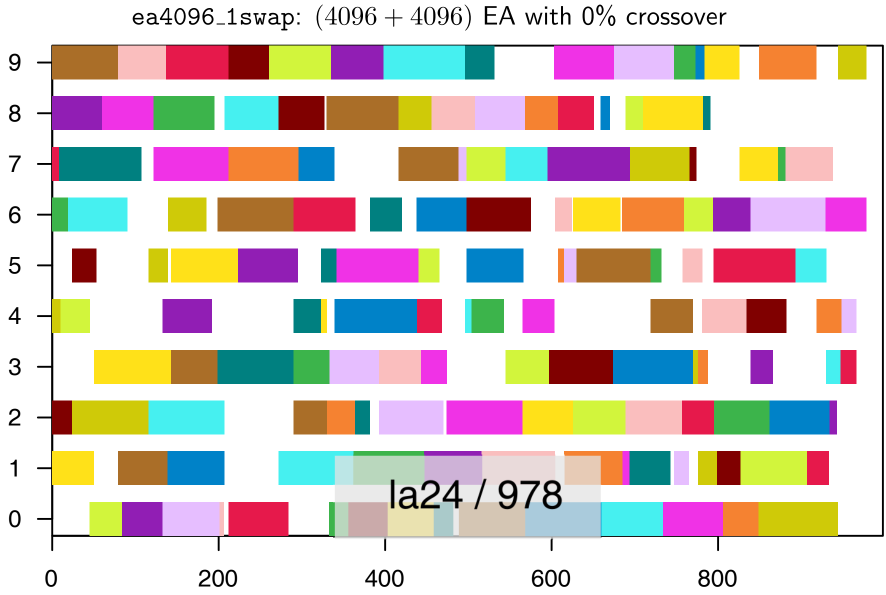


So what do we get?

hcr_256+5%_1swap: HC with restarts after 256+5% non-improv steps

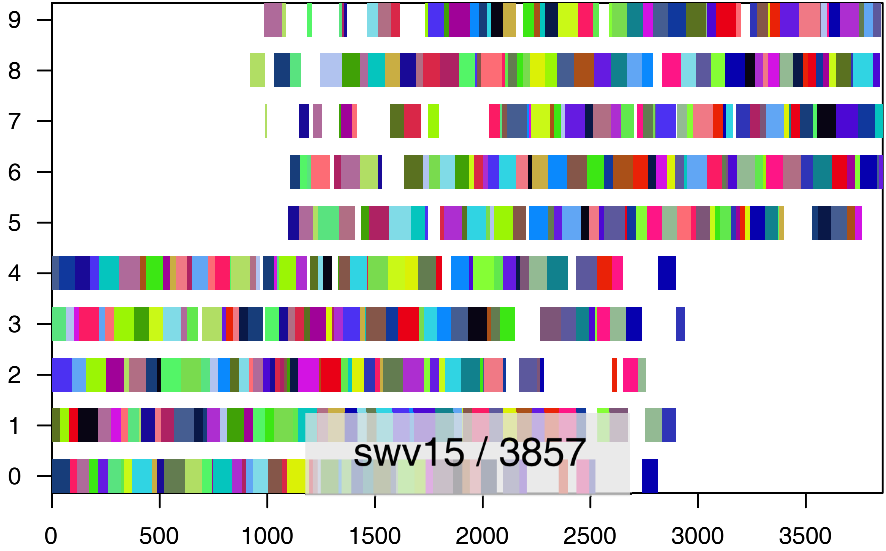


So what do we get?



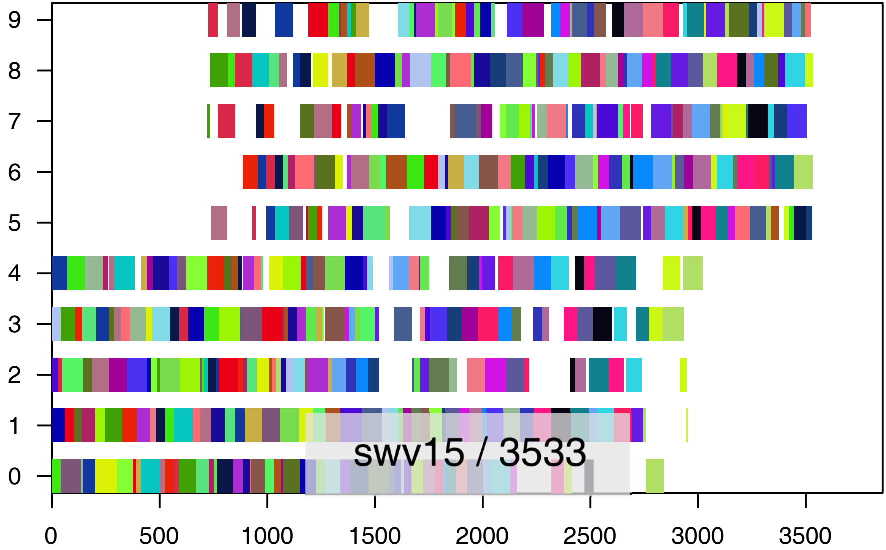
So what do we get?

`hcr_256+5%_1swap`: HC with restarts after 256+5% non-improv steps



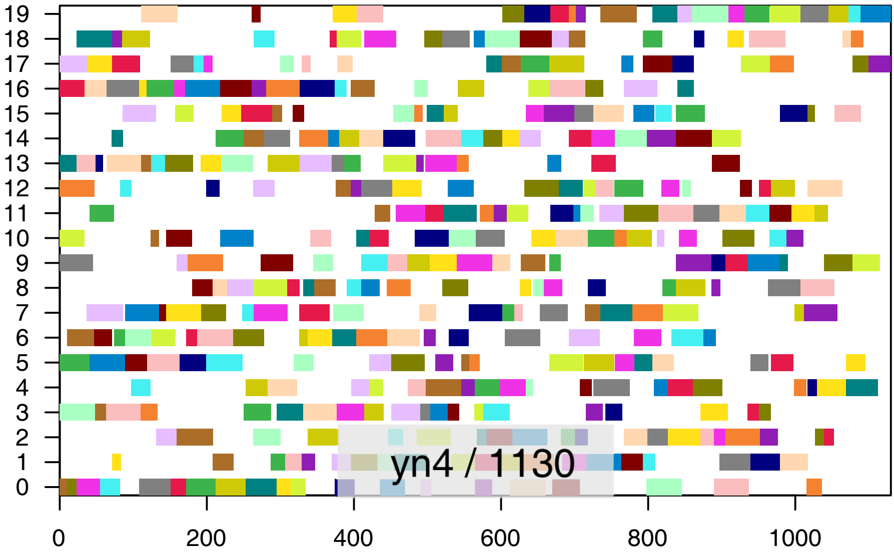
So what do we get?

ea4096_1swap: $(4096 + 4096)$ EA with 0% crossover

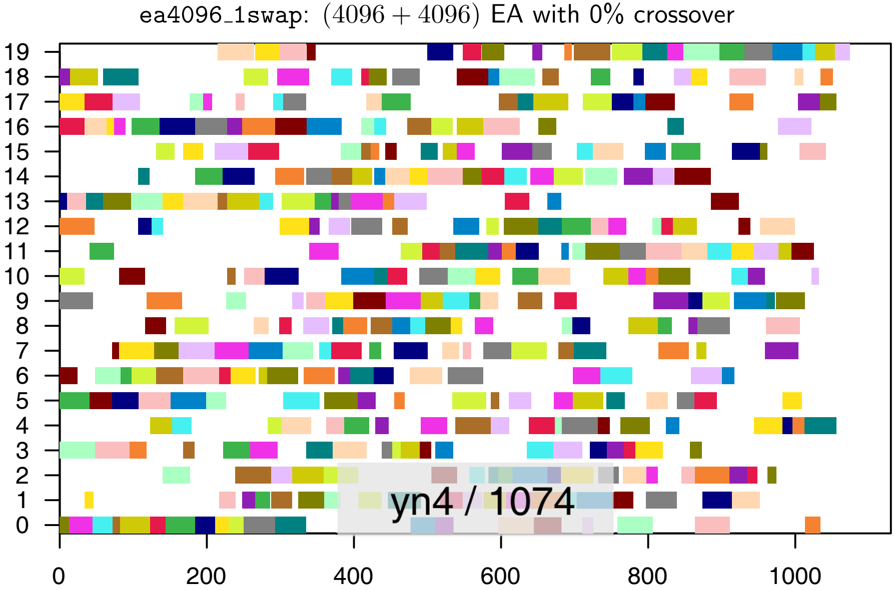


So what do we get?

hcr_256+5%_1swap: HC with restarts after 256+5% non-improv steps



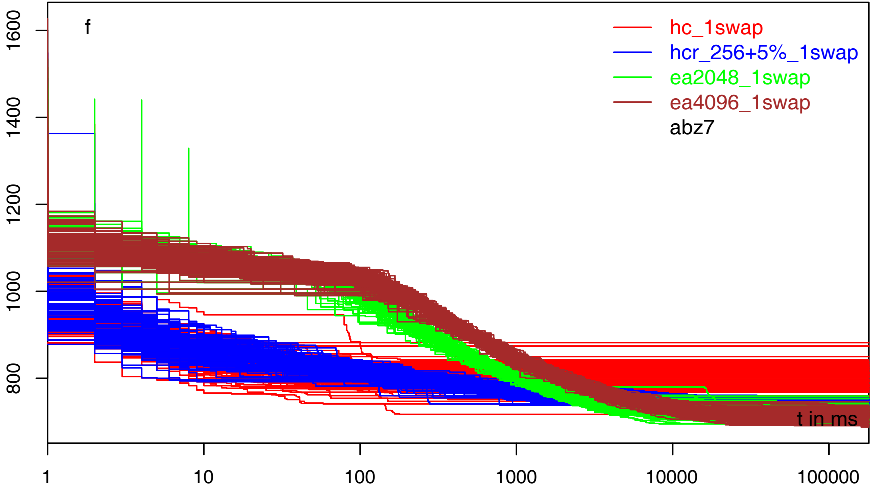
So what do we get?



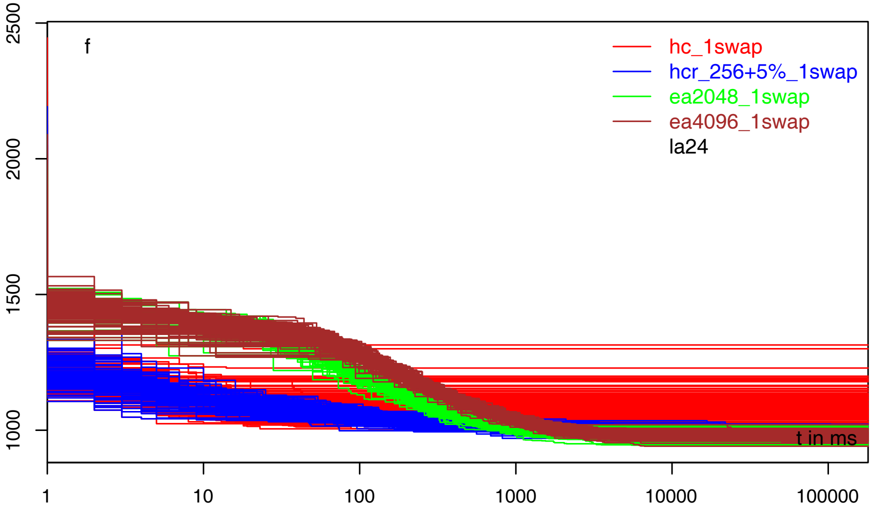
Progress over Time

What progress does the algorithm make over time?

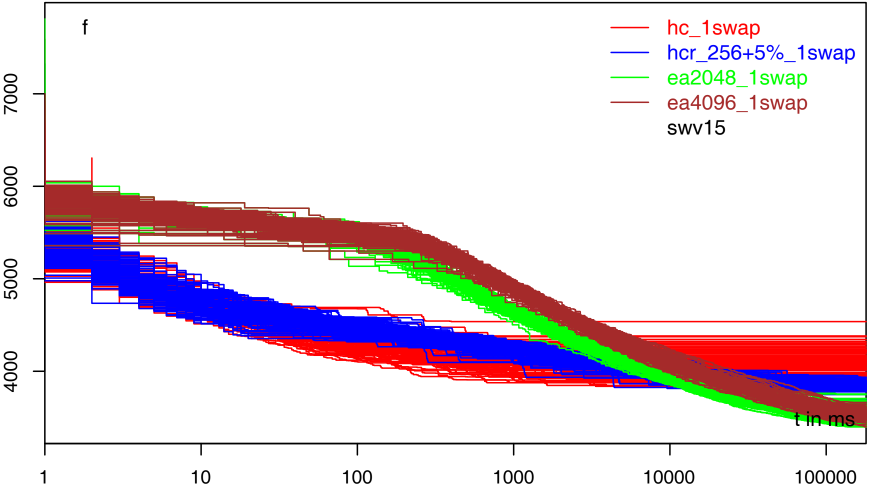
What progress does the algorithm make over time?



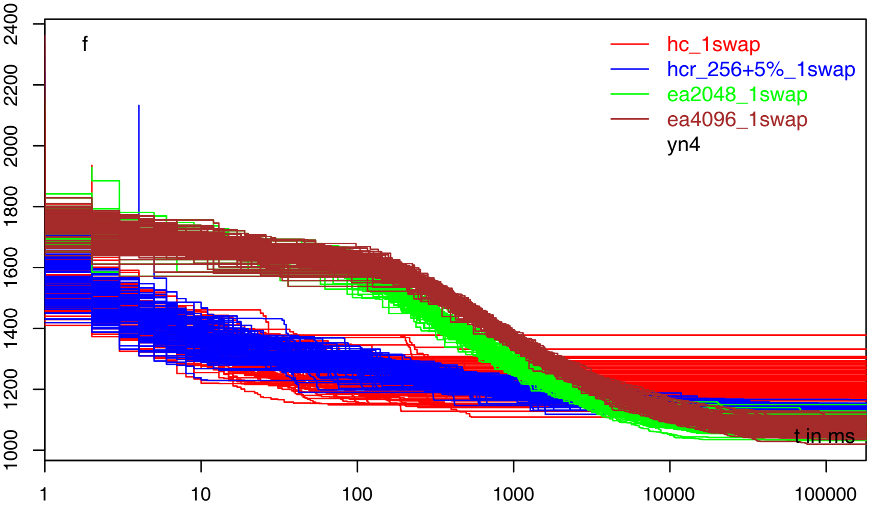
What progress does the algorithm make over time?



What progress does the algorithm make over time?



What progress does the algorithm make over time?



Relationship EA / Hill Climber / Random Sampling

- We can imagine an EA to be a generalized version of a hill climber

Relationship EA / Hill Climber / Random Sampling

- We can imagine an EA to be a generalized version of a hill climber
- Or the other way around: A hill climber is a $(1+1)$ EA

- We can imagine an EA to be a generalized version of a hill climber
- Or the other way around: A hill climber is a $(1+1)$ EA, i.e., an EA where we always remember the $\mu = 1$ best solutions

Relationship EA / Hill Climber / Random Sampling

- We can imagine an EA to be a generalized version of a hill climber
- Or the other way around: A hill climber is a $(1 + 1)$ EA, i.e., an EA where we always remember the $\mu = 1$ best solutions and use them as parents for $\lambda = 1$ new solutions

Relationship EA / Hill Climber / Random Sampling

- We can imagine an EA to be a generalized version of a hill climber
- Or the other way around: A hill climber is a $(1+1)$ EA, i.e., an EA where we always remember the $\mu=1$ best solutions and use them as parents for $\lambda=1$ new solutions, which we create using the unary modification operator as modified copy of the $\mu=1$ parent.

Relationship EA / Hill Climber / Random Sampling

- We can imagine an EA to be a generalized version of a hill climber
- Or the other way around: A hill climber is a $(1+1)$ EA, i.e., an EA where we always remember the $\mu = 1$ best solutions and use them as parents for $\lambda = 1$ new solutions, which we create using the unary modification operator as modified copy of the $\mu = 1$ parent.
- On the other hand: For the first $\mu + \lambda$ (random) solutions, the EA always behaves exactly like a random sampling algorithm.

Relationship EA / Hill Climber / Random Sampling

- We can imagine an EA to be a generalized version of a hill climber
- Or the other way around: A hill climber is a $(1+1)$ EA, i.e., an EA where we always remember the $\mu = 1$ best solutions and use them as parents for $\lambda = 1$ new solutions, which we create using the unary modification operator as modified copy of the $\mu = 1$ parent.
- On the other hand: For the first $\mu + \lambda$ (random) solutions, the EA always behaves exactly like a random sampling algorithm.
- If $\mu + \lambda \to +\infty$, the EA *becomes* a random sampling algorithm.

Relationship EA / Hill Climber / Random Sampling

- We can imagine an EA to be a generalized version of a hill climber
- Or the other way around: A hill climber is a $(1+1)$ EA, i.e., an EA where we always remember the $\mu = 1$ best solutions and use them as parents for $\lambda = 1$ new solutions, which we create using the unary modification operator as modified copy of the $\mu = 1$ parent.
- On the other hand: For the first $\mu + \lambda$ (random) solutions, the EA always behaves exactly like a random sampling algorithm.
- Actually, for $\mu + \lambda \geq \eta$, with an η large enough to completely exhaust our computational budget (here: 3min), the EA *is* a random sampling algorithm.

Relationship EA / Hill Climber / Random Sampling

- We can imagine an EA to be a generalized version of a hill climber
- Or the other way around: A hill climber is a $(1+1)$ EA, i.e., an EA where we always remember the $\mu = 1$ best solutions and use them as parents for $\lambda = 1$ new solutions, which we create using the unary modification operator as modified copy of the $\mu = 1$ parent.
- On the other hand: For the first $\mu + \lambda$ (random) solutions, the EA always behaves exactly like a random sampling algorithm.
- Actually, for $\mu + \lambda \geq \eta$, with an η large enough to completely exhaust our computational budget (here: 3min), the EA *is* a random sampling algorithm.
- **An EA is a way to choose an algorithm behavior in between random sampling and hill climbing!**

- **An EA is a way to choose an algorithm behavior in between random sampling and hill climbing!**

Exploration versus Exploitation

- **An EA is a way to choose an algorithm behavior in between random sampling and hill climbing!**
- The parameter μ basically allows us to "tune" between these two behaviors [8]

Exploration versus Exploitation

- **An EA is a way to choose an algorithm behavior in between random sampling and hill climbing!**
- The parameter μ basically allows us to "tune" between these two behaviors [8]
- If we pick it small, our algorithm becomes more "greedy".

- **An EA is a way to choose an algorithm behavior in between random sampling and hill climbing!**
- The parameter μ basically allows us to “tune” between these two behaviors [8]
- If we pick it small, our algorithm becomes more “greedy”.
- It will investigate (exploit) the neighborhood current best solutions more eagerly.

Exploration versus Exploitation

- **An EA is a way to choose an algorithm behavior in between random sampling and hill climbing!**
- The parameter μ basically allows us to "tune" between these two behaviors [8]
- If we pick it small, our algorithm becomes more "greedy".
- It will investigate (exploit) the neighborhood current best solutions more eagerly, which means that it will trace down local optima faster.

- **An EA is a way to choose an algorithm behavior in between random sampling and hill climbing!**
- The parameter μ basically allows us to "tune" between these two behaviors [8]
- If we pick it small, our algorithm becomes more "greedy".
- It will investigate (exploit) the neighborhood current best solutions more eagerly, which means that it will trace down local optima faster but be trapped more easily in local optima as well.

- **An EA is a way to choose an algorithm behavior in between random sampling and hill climbing!**
- The parameter μ basically allows us to "tune" between these two behaviors [8]
- If we pick it small, our algorithm becomes more "greedy".
- It will investigate (exploit) the neighborhood current best solutions more eagerly, which means that it will trace down local optima faster but be trapped more easily in local optima as well.
- The bigger μ, the more points in the search space are maintained and the more likely are we do to good "global" search, we do more exploration.

Exploration versus Exploitation

- **An EA is a way to choose an algorithm behavior in between random sampling and hill climbing!**
- The parameter μ basically allows us to "tune" between these two behaviors [8]
- If we pick it small, our algorithm becomes more "greedy".
- It will investigate (exploit) the neighborhood current best solutions more eagerly, which means that it will trace down local optima faster but be trapped more easily in local optima as well.
- The bigger μ, the more points in the search space are maintained and the more likely are we do to good "global" search, we do more exploration. We pay for that by a slower exploitation (investigation) of the current best solution (because we always work on all μ points, not just one).

- **An EA is a way to choose an algorithm behavior in between random sampling and hill climbing!**
- The parameter μ basically allows us to "tune" between these two behaviors [8]
- If we pick it small, our algorithm becomes more "greedy".
- It will investigate (exploit) the neighborhood current best solutions more eagerly, which means that it will trace down local optima faster but be trapped more easily in local optima as well.
- The bigger μ, the more points in the search space are maintained and the more likely are we do to good "global" search, we do more exploration. We pay for that by a slower exploitation (investigation) of the current best solution (because we always work on all μ points, not just one).
- This is dilemma of "Exploration versus Exploitation" [2, 9–11].

- We now have more than one candidate solution in our "population"

Binary Search Operator

- We now have more than one candidate solution in our "population"
- But we only use one existing point from $\mathbb{X}$ as "blueprint" when we create a new one.

- We now have more than one candidate solution in our “population”
- But we only use one existing point from $\mathbb{X}$ as “blueprint” when we create a new one.
- Why can’t we use *two* instead?

Binary Search Operator

- We now have more than one candidate solution in our "population"
- But we only use one existing point from $\mathbb{X}$ as "blueprint" when we create a new one.
- Why can't we use *two* instead?
  - If two candidate solutions have been selected, they are probably good.

- We now have more than one candidate solution in our "population"
- But we only use one existing point from $\mathbb{X}$ as "blueprint" when we create a new one.
- Why can't we use *two* instead?
  - If two candidate solutions have been selected, they are probably good.
  - If two candidate solutions are good, they may have different positive characteristics.

Binary Search Operator

- We now have more than one candidate solution in our “population”
- But we only use one existing point from $\mathbb{X}$ as “blueprint” when we create a new one.
- Why can't we use *two* instead?
  - If two candidate solutions have been selected, they are probably good.
  - If two candidate solutions are good, they may have different positive characteristics.
  - Let's try to create a new “offspring” solution which inherits characteristics from both “parents”.

Binary Search Operator

- We now have more than one candidate solution in our "population"
- But we only use one existing point from $\mathbb{X}$ as "blueprint" when we create a new one.
- Why can't we use *two* instead?
  - If two candidate solutions have been selected, they are probably good.
  - If two candidate solutions are good, they may have different positive characteristics.
  - Let's try to create a new "offspring" solution which inherits characteristics from both "parents".
  - It could maybe inherit the positiv traits and combine them...

Binary Search Operator

- We now have more than one candidate solution in our "population"
- But we only use one existing point from $\mathbb{X}$ as "blueprint" when we create a new one.
- Why can't we use *two* instead?
  - If two candidate solutions have been selected, they are probably good.
  - If two candidate solutions are good, they may have different positive characteristics.
  - Let's try to create a new "offspring" solution which inherits characteristics from both "parents".
  - It could maybe inherit the positiv traits and combine them...
- This is the idea of the *crossover* or *recombination* operator in Evolutionary Algorithms. [2, 3, 7]

- The $(\mu + \lambda)$ EAs with recombination work as follows:

- The $(\mu + \lambda)$ EAs with recombination work as follows:
  1. Generate a population of $\mu + \lambda$ random points in the search space (and map them to solutions and evaluate them).

$(\mu + \lambda)$ EA with Recombination

- The $(\mu + \lambda)$ EAs with recombination work as follows:
  1. Generate a population of $\mu + \lambda$ random points in the search space (and map them to solutions and evaluate them).
  2. From the complete population, select the μ best points as "parents" for the next "generation", discard the remaining λ points.

- The $(\mu + \lambda)$ EAs with recombination work as follows:
  1. Generate a population of $\mu + \lambda$ random points in the search space (and map them to solutions and evaluate them).
  2. From the complete population, select the μ best points as "parents" for the next "generation", discard the remaining λ points.
  3. Generate λ new "offspring" points

$(\mu + \lambda)$ EA with Recombination

- The $(\mu + \lambda)$ EAs with recombination work as follows:
  1. Generate a population of $\mu + \lambda$ random points in the search space (and map them to solutions and evaluate them).
  2. From the complete population, select the μ best points as “parents” for the next “generation”, discard the remaining λ points.
  3. Generate λ new “offspring” points by
     1. applying a binary recombination operator which combines two existing parents to one new offspring

- The $(\mu + \lambda)$ EAs with recombination work as follows:
  1. Generate a population of $\mu + \lambda$ random points in the search space (and map them to solutions and evaluate them).
  2. From the complete population, select the μ best points as "parents" for the next "generation", discard the remaining λ points.
  3. Generate λ new "offspring" points by
     1. applying a binary recombination operator which combines two existing parents to one new offspring with probability cr

$(\mu + \lambda)$ EA with Recombination

- The $(\mu + \lambda)$ EAs with recombination work as follows:
  1. Generate a population of $\mu + \lambda$ random points in the search space (and map them to solutions and evaluate them).
  2. From the complete population, select the μ best points as "parents" for the next "generation", discard the remaining λ points.
  3. Generate λ new "offspring" points by either
     1. applying a binary recombination operator which combines two existing parents to one new offspring with probability cr or
     2. applying a unary search operator which creates a randomly modified copy from a parent as offspring.

$(\mu + \lambda)$ EA with Recombination

- The $(\mu + \lambda)$ EAs with recombination work as follows:
  1. Generate a population of $\mu + \lambda$ random points in the search space (and map them to solutions and evaluate them).
  2. From the complete population, select the μ best points as "parents" for the next "generation", discard the remaining λ points.
  3. Generate λ new "offspring" points by either
     1. applying a binary recombination operator which combines two existing parents to one new offspring with probability cr or
     2. applying a unary search operator which creates a randomly modified copy from a parent as offspring.
  4. Evaluate the λ offsprings, add them to the population, and go back to step 2.

Recombination for our Representation: Idea

Recombination for our Representation: Idea

1. Repeat until new point in search space is completely constructed

Recombination for our Representation: Idea

1. Repeat until new point in search space is completely constructed
    1. Randomly choose one of the two input points $x1$ or $x2$ with equal probability as source x for the next job id.

1. Repeat until new point in search space is completely constructed
   1. Randomly choose one of the two input points $x1$ or $x2$ with equal probability as source x for the next job id.
   2. Select the first job id J in x that has not yet been used.

Recombination for our Representation: Idea

1. Repeat until new point in search space is completely constructed
   1. Randomly choose one of the two input points $x1$ or $x2$ with equal probability as source x for the next job id.
   2. Select the first job id J in x that has not yet been used.
   3. Append the job id J of this sub-job to the new point

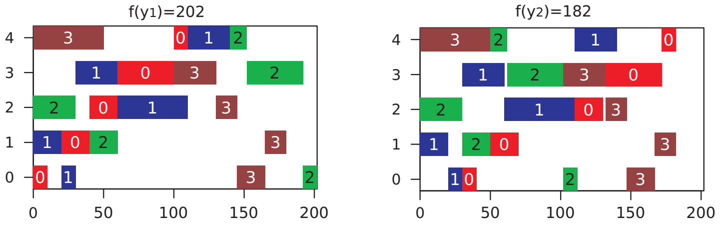
Recombination for our Representation: Idea

1. Repeat until new point in search space is completely constructed
    1. Randomly choose one of the two input points $x1$ or $x2$ with equal probability as source x for the next job id.
    2. Select the first job id J in x that has not yet been used.
    3. Append the job id J of this sub-job to the new point
    4. Mark the first unmarked occurrence of J as "already used" in $x1$.

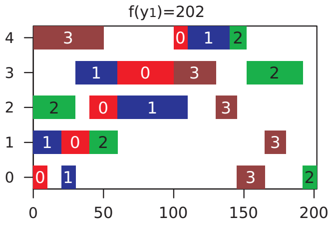
Recombination for our Representation: Idea

1. Repeat until new point in search space is completely constructed
    1. Randomly choose one of the two input points $x1$ or $x2$ with equal probability as source x for the next job id.
    2. Select the first job id J in x that has not yet been used.
    3. Append the job id J of this sub-job to the new point
    4. Mark the first unmarked occurrence of J as “already used” in $x1$.
    5. Mark the first unmarked occurrence of J as “already used” in $x2$.

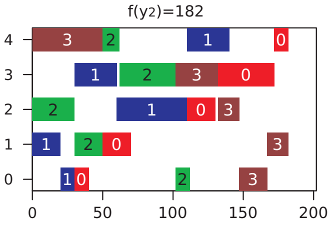
Example for Sequence Recombination



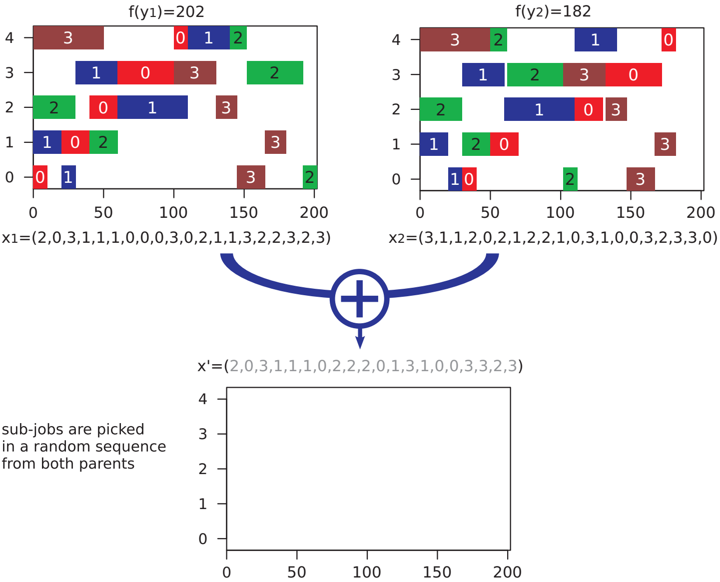
Example for Sequence Recombination



x1=(2,0,3,1,1,1,0,0,0,3,0,2,1,1,3,2,2,3,2,3)

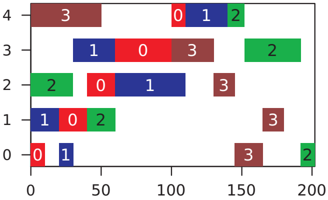


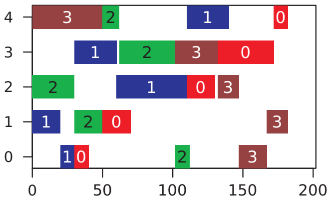
x2=(3,1,1,2,0,2,1,2,2,1,0,3,1,0,0,3,2,3,3,0)



sub-jobs are picked
in a random sequence
from both parents

Example for Sequence Recombination



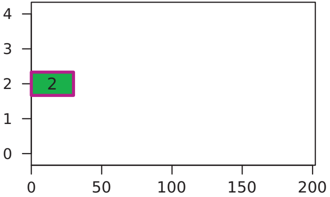


x1=(2,0,3,1,1,1,0,0,0,3,0,2,1,1,3,2,2,3,2,3) x2=(3,1,1,~~2~~,0,2,1,2,2,1,0,3,1,0,0,3,2,3,3,0)

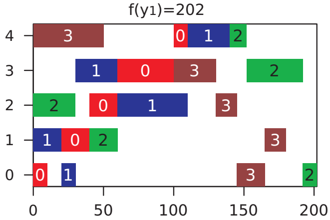
x'=(2,0,3,1,1,1,0,2,2,2,0,1,3,1,0,0,3,3,2,3)

random sequence in which the sub-jobs are picked:

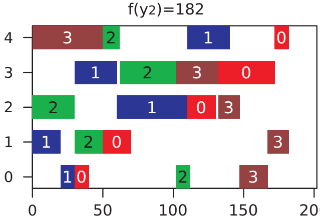
x1



Example for Sequence Recombination



x_1=(2,0,3,1,1,1,0,0,0,3,0,2,1,1,3,2,2,3,2,3)

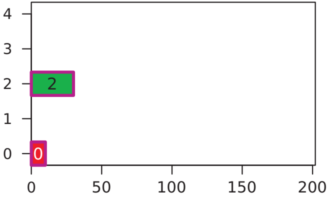


x_2=(3,1,1,~~2~~,~~0~~,2,1,2,2,1,0,3,1,0,0,3,2,3,3,0)

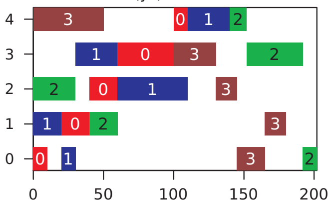
x'=(2,0,3,1,1,1,0,2,2,2,0,1,3,1,0,0,3,3,2,3)

random sequence in which the sub-jobs are picked:

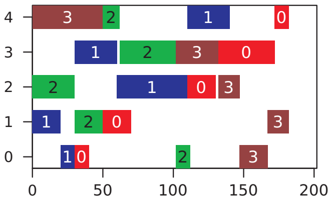
x_1, x_1



Example for Sequence Recombination



$x_1=(2,0,3,1,1,1,0,0,0,3,0,2,1,1,3,2,2,3,2,3)$

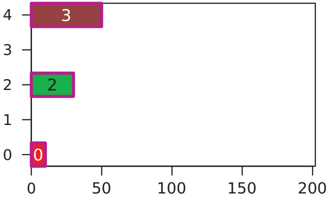


$x_2=(\cancel{3},1,1,\cancel{2},\cancel{0},2,1,2,2,1,0,3,1,0,0,3,2,3,3,0)$

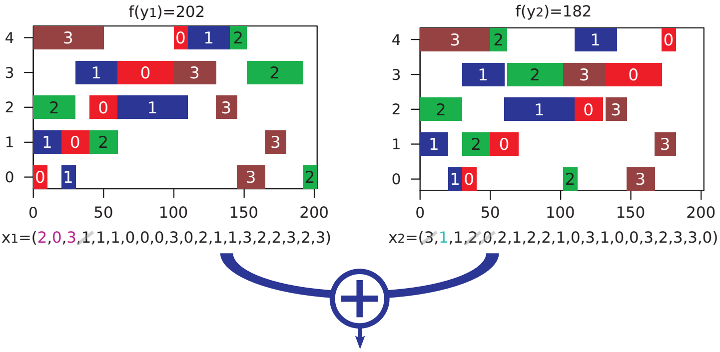
$x'=(2,0,3,1,1,1,0,2,2,2,0,1,3,1,0,0,3,3,2,3)$

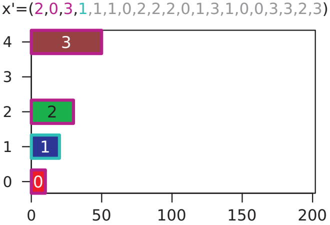
random sequence in which the sub-jobs are picked:

x_1, x_1, x_1



Example for Sequence Recombination

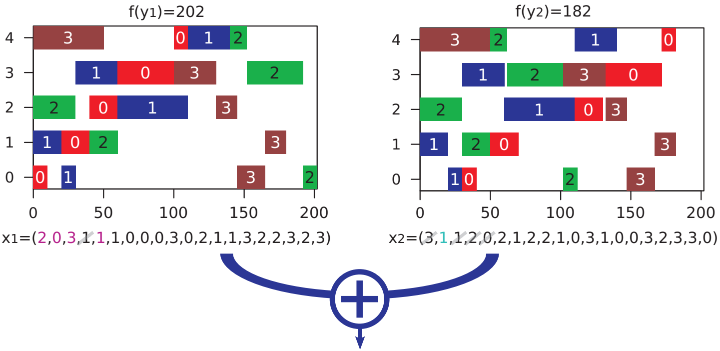




random sequence in which the sub-jobs are picked:

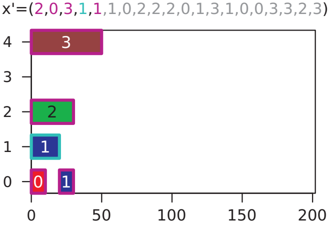
x1, x1, x1, x2

Example for Sequence Recombination



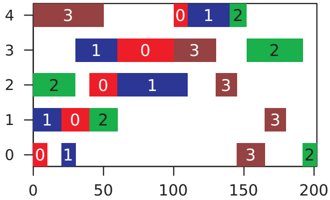
random sequence in which the sub-jobs are picked:

x1, x1, x1, x2, x1



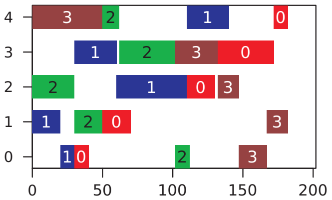
Example for Sequence Recombination

f(y_1)=202



x1=(2,0,3,~~1~~,1,1,0,0,0,3,0,2,1,1,3,2,2,3,2,3)

f(y_2)=182

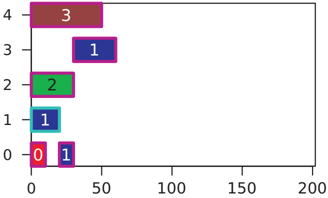


x2=(~~3~~,1,~~1~~,~~2~~,~~0~~,2,~~1~~,2,2,1,0,3,1,0,0,3,2,3,3,0)

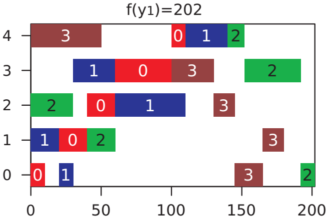
x'=(2,0,3,1,1,1,0,2,2,2,0,1,3,1,0,0,3,3,2,3)

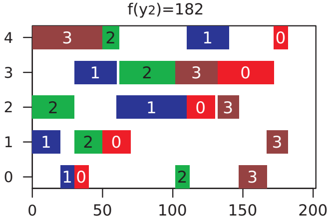
random sequence in which the sub-jobs are picked:

x1, x1, x1, x2, x1, x1



Example for Sequence Recombination

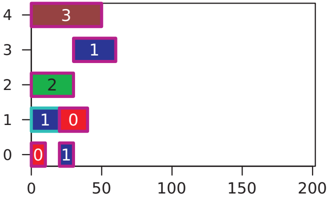




x1=(2,0,3,~~1~~,1,1,0,0,0,3,0,2,1,1,3,2,2,3,2,3)

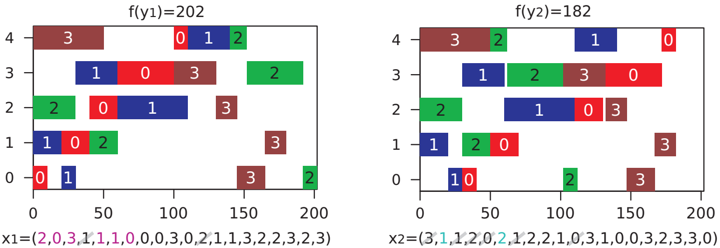
x2=(~~3~~,1,~~1~~,~~2~~,~~0~~,2,~~1~~,2,2,1,~~0~~,3,1,0,0,3,2,3,3,0)

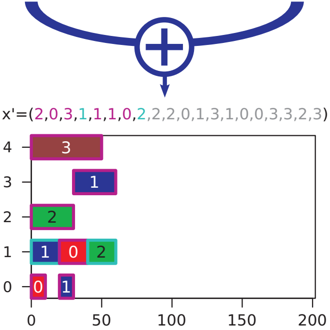
x'=(2,0,3,1,1,1,0,2,2,2,0,1,3,1,0,0,3,3,2,3)



random sequence in which the sub-jobs are picked:

x1, x1, x1, x2, x1, x1, x1

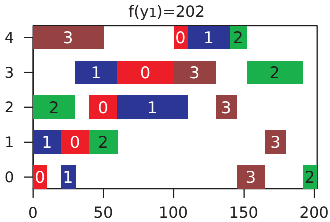


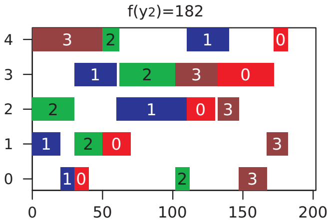


random sequence in which the sub-jobs are picked:

x1, x1, x1, x2, x1, x1, x1, x2

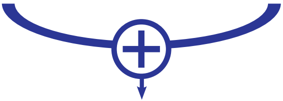
Example for Sequence Recombination



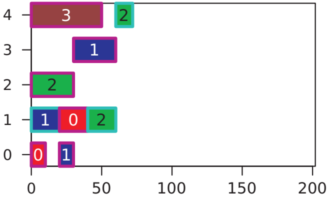


$x_1=(2,0,3,\cancel{1},1,1,0,0,0,3,0,\cancel{2},1,1,3,\cancel{2},2,3,2,3)$

$x_2=(\cancel{3},1,\cancel{1},\cancel{2},\cancel{0},2,\cancel{1},2,2,1,\cancel{0},3,1,0,0,3,2,3,3,0)$



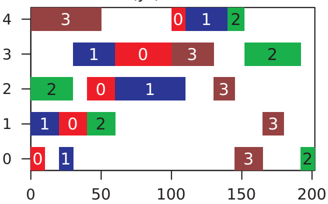
$x'=(2,0,3,1,1,1,0,2,2,2,0,1,3,1,0,0,3,3,2,3)$



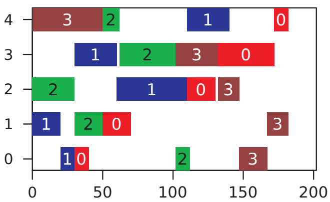
random sequence in which the sub-jobs are picked:

x_1, x_1, x_1, x_2, x_1, x_1, x_1, x_2, x_2

Example for Sequence Recombination



x1=(2,0,3,~~1~~,1,1,0,0,0,3,0,~~2~~,1,1,3,~~2~~,~~2~~,3,2,3)

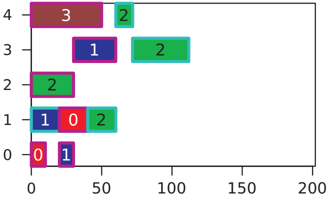


x2=(~~3~~,1,~~1~~,~~2~~,~~0~~,2,~~1~~,2,2,1,~~0~~,3,1,0,0,3,2,3,3,0)

x'=(2,0,3,1,1,1,0,2,2,2,0,1,3,1,0,0,3,3,2,3)

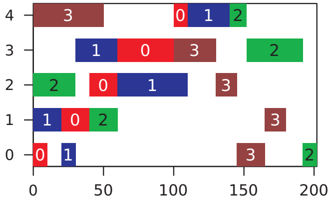
random sequence in which the sub-jobs are picked:

x1, x1, x1, x2, x1, x1, x1, x2, x2, x2

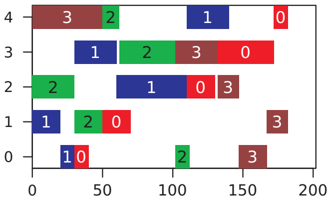


Example for Sequence Recombination

f(y1)=202



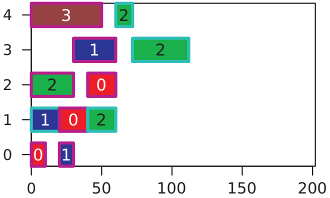
f(y2)=182



x1=(2,0,3,~~1~~,1,1,0,0,0,3,0,~~2~~,1,1,3,~~2~~,~~2~~,3,2,3)

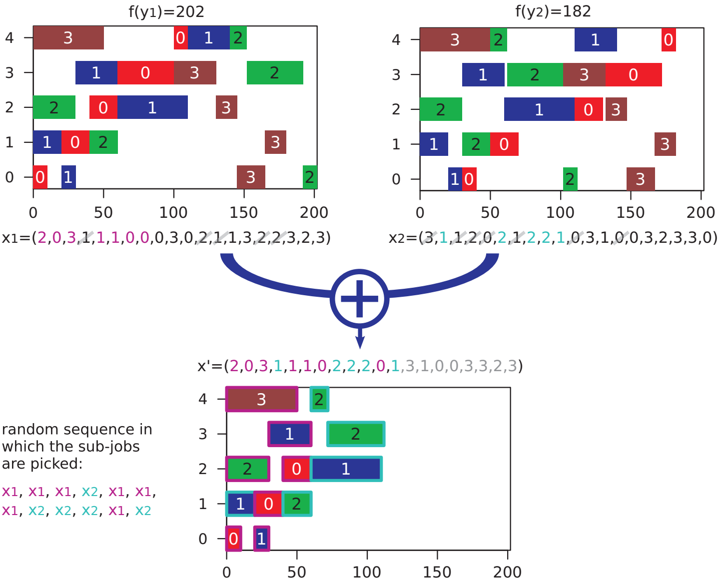
x2=(~~3~~,1,~~1~~,~~2~~,~~0~~,2,~~1~~,2,2,1,~~0~~,3,1,~~0~~,0,3,2,3,3,0)

x'=(2,0,3,1,1,1,0,2,2,2,0,1,3,1,0,0,3,3,2,3)



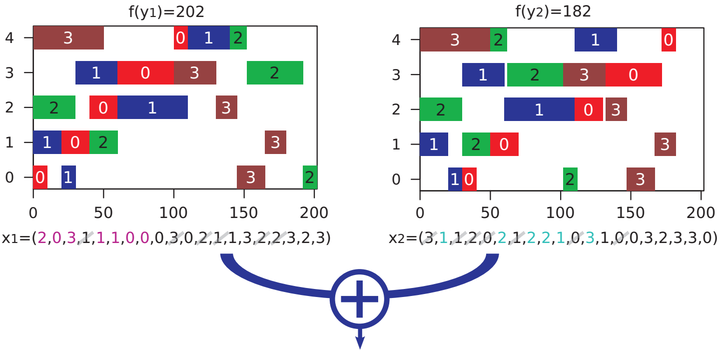
random sequence in which the sub-jobs are picked:

x1, x1, x1, x2, x1, x1, x1, x2, x2, x2, x1



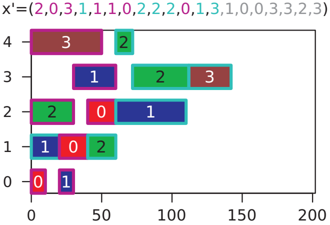
random sequence in which the sub-jobs are picked:

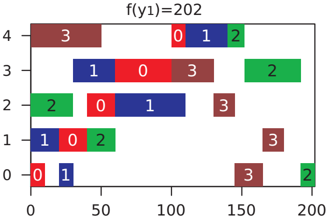
x_1, x_1, x_1, x_2, x_1, x_1, x_1, x_2, x_2, x_2, x_1, x_2

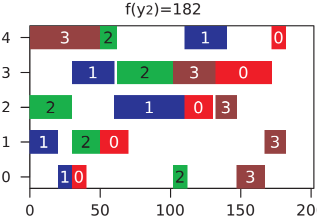


random sequence in which the sub-jobs are picked:

x1, x1, x1, x2, x1, x1, x1, x2, x2, x2, x1, x2, x2





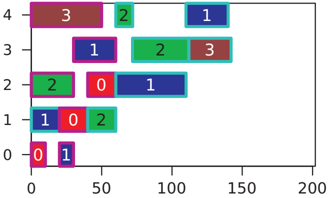


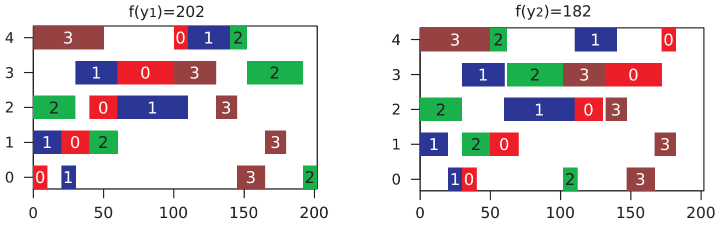
x1=(2,0,3,1,1,1,0,0,0,3,0,2,1,1,3,2,2,3,2,3) x2=(3,1,1,2,0,2,1,2,2,1,0,3,1,0,0,3,2,3,3,0)

x'=(2,0,3,1,1,1,0,2,2,2,0,1,3,1,0,0,3,3,2,3)

random sequence in which the sub-jobs are picked:

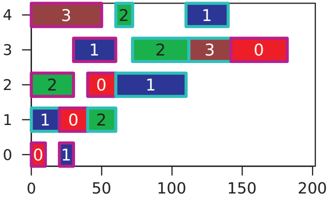
x1, x1, x1, x2, x1, x1, x1, x2, x2, x2, x1, x2, x2, x2





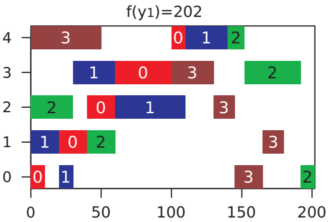
x_1=(2,0,3,1,1,1,0,0,0,3,0,2,1,1,3,2,2,3,2,3) x_2=(3,1,1,2,0,2,1,2,2,1,0,3,1,0,0,3,2,3,3,0)

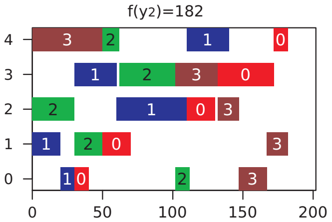
x'=(2,0,3,1,1,1,0,2,2,2,0,1,3,1,0,0,3,3,2,3)



random sequence in which the sub-jobs are picked:

x_1, x_1, x_1, x_2, x_1, x_1, x_1, x_2, x_2, x_2, x_1, x_2, x_2, x_2, x_1

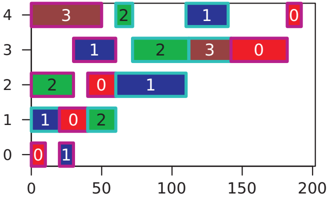




x1=(2,0,3,1,1,1,0,0,0,3,0,2,1,1,3,2,2,3,2,3)

x2=(3,1,1,2,0,2,1,2,2,1,0,3,1,0,0,3,2,3,3,0)

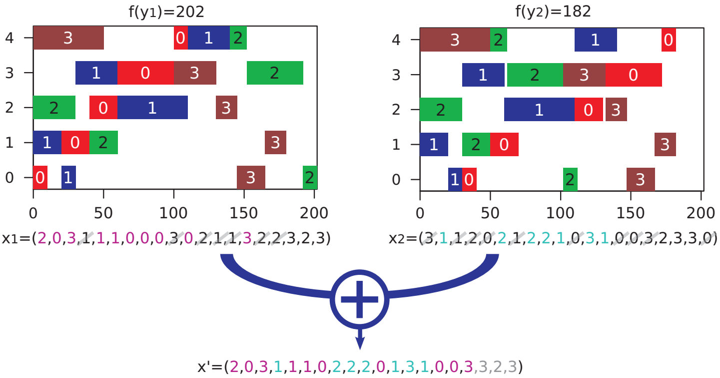
x'=(2,0,3,1,1,1,0,2,2,2,0,1,3,1,0,0,3,3,2,3)

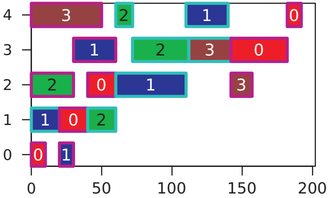


random sequence in which the sub-jobs are picked:

x1, x1, x1, x2, x1, x1, x1, x2, x2, x2, x1, x2, x2, x2, x1, x1

Example for Sequence Recombination

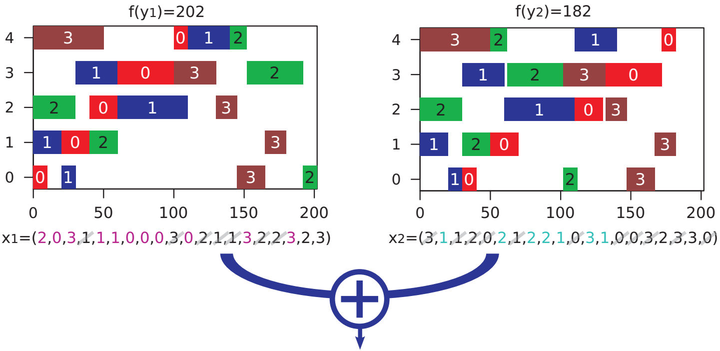




random sequence in which the sub-jobs are picked:

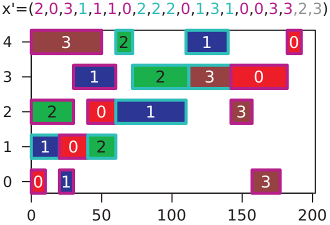
x1, x1, x1, x2, x1, x1, x1, x2, x2, x2, x1, x2, x2, x2, x1, x1, x1

Example for Sequence Recombination

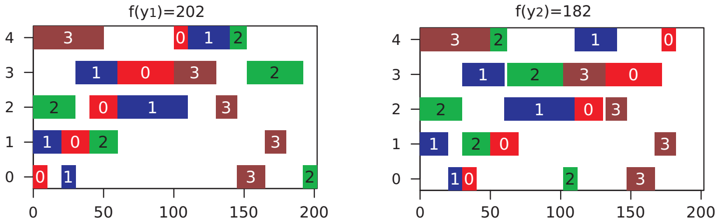


random sequence in which the sub-jobs are picked:

x1, x1, x1, x2, x1, x1, x1, x2, x2, x2, x1, x2, x2, x2, x1, x1, x1, x1



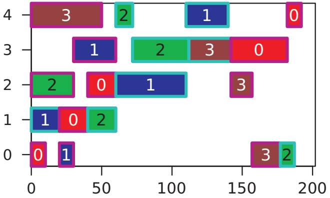
Example for Sequence Recombination



x_1=(2,0,3,~~1~~,1,1,0,0,0,~~3~~,0,~~2~~,~~1~~,~~1~~,3,~~2~~,~~2~~,3,~~2~~,3)

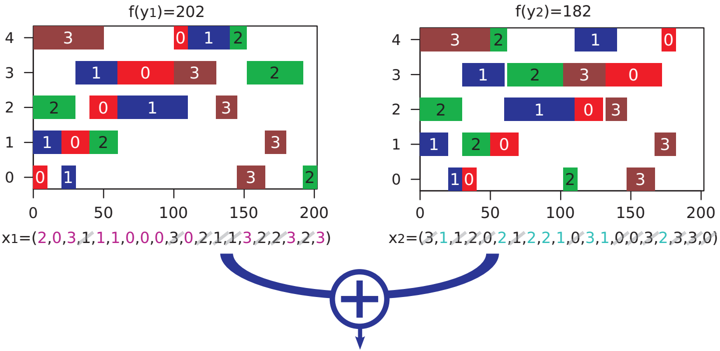
x_2=(~~3~~,1,~~1~~,~~2~~,~~0~~,2,~~1~~,2,2,1,~~0~~,3,1,~~0~~,~~0~~,~~3~~,2,~~3~~,3,~~0~~)

x'=(2,0,3,1,1,1,0,2,2,2,0,1,3,1,0,0,3,3,2,3)



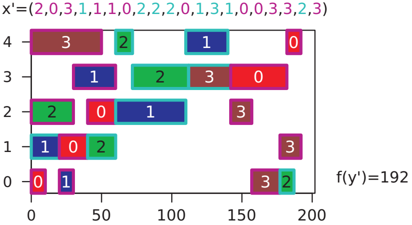
random sequence in which the sub-jobs are picked:

x_1, x_1, x_1, x_2, x_1, x_1, x_1, x_2, x_2, x_2, x_1, x_2, x_2, x_2, x_1, x_1, x_1, x_1, x_2

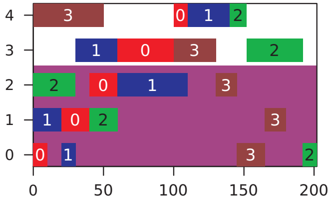


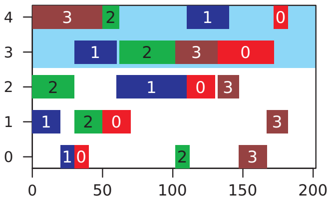
random sequence in which the sub-jobs are picked:

x1, x1, x1, x2, x1, x1, x1, x2, x2, x2, x1, x2, x2, x2, x1, x1, x1, x1, x2, x1



Example for Sequence Recombination





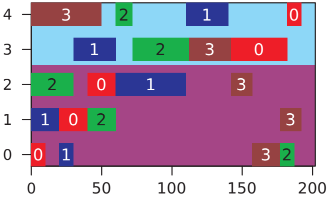
x1=(2,0,3,~~1~~,1,1,0,0,0,~~3~~,0,~~2~~,~~1~~,~~1~~,3,~~2~~,~~2~~,3,~~2~~,3)

x2=(~~3~~,1,~~1~~,~~2~~,~~0~~,2,~~1~~,2,2,1,~~0~~,3,1,~~0~~,~~0~~,~~3~~,2,~~3~~,~~3~~,~~0~~)

x'=(2,0,3,1,1,1,0,2,2,2,0,1,3,1,0,0,3,3,2,3)

random sequence in which the sub-jobs are picked:

x1, x1, x1, x2, x1, x1, x1, x2, x2, x2, x1, x2, x2, x2, x1, x1, x1, x1, x2, x1



f(y')=192

The Binary Operator in Java

Listing: Recombination for our Representation

```java
public class JSSPBinaryOperatorSequence implements IBinarySearchOperator<int[]> {

  public void apply(int[] x0, int[] x1, int[] dest, Random random) {
    boolean[] done_x0 = this.m_done_x0;
    Arrays.fill(done_x0, false);
    boolean[] done_x1 = this.m_done_x1;
    Arrays.fill(done_x1, false);

    int length = done_x0.length;
    int desti = 0, x0i = 0, x1i = 0;

    for (;;) {
      int add = random.nextBoolean() ? x0[x0i] : x1[x1i];
      dest[desti++] = add;
      if (desti >= length) return;

      for (int i = x0i;; i++) {
        if ((x0[i] == add) && (!done_x0[i])) {
          done_x0[i] = true; break;
        }
      }
      while (done_x0[x0i]) x0i++;

      for (int i = x1i;; i++) {
        if ((x1[i] == add) && (!done_x1[i])) {
          done_x1[i] = true; break;
        }
      }
      while (done_x1[x1i]) x1i++;
    }
  }
}
```

Section Outline

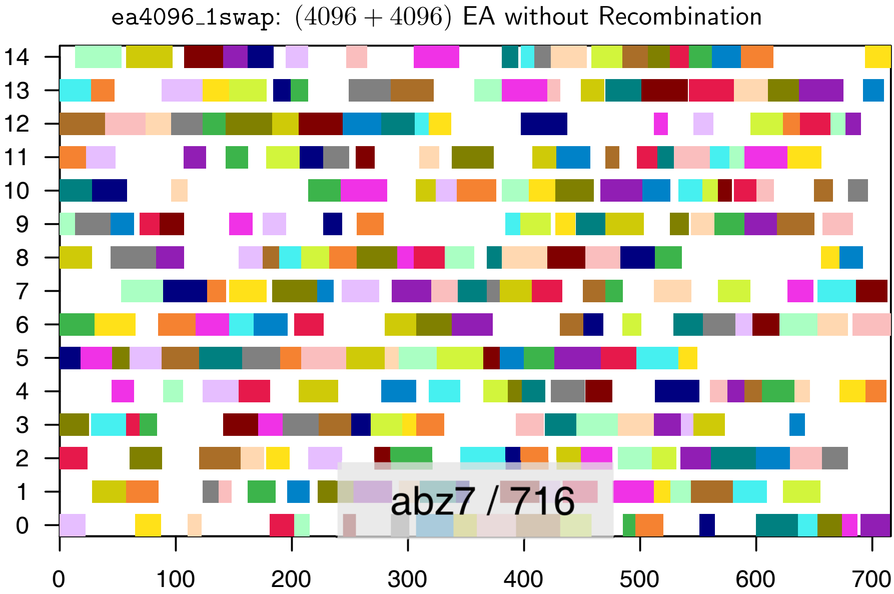
So what do we get?

- We now test the same EAs as before, but apply the binary operator at 5%

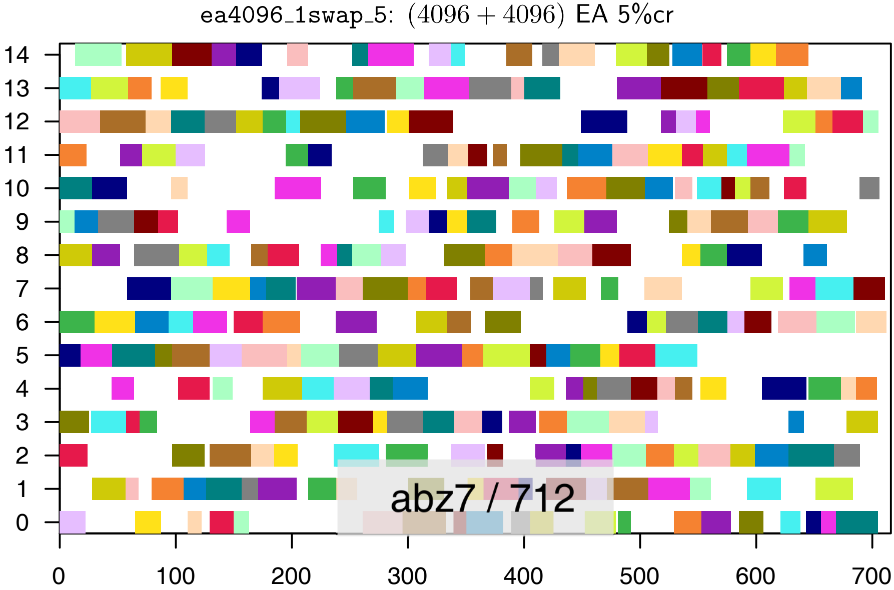
So what do we get?

		makespan				last improvement	
$\mathcal{I}$	algo	best	mean	med	sd	med(t)	med(FEs)
abz7	ea2048_1swap	695	719	718	13	11s	2'581'614
	ea4096_1swap	688	716	716	12	19s	4'416'129
	ea2048_1swap_5	689	713	**712**	11	12s	2'641'808
	ea4096_1swap_5	**680**	**712**	**712**	**10**	20s	4'145'924
la24	ea2048_1swap	945	983	983	16	2s	927'000
	ea4096_1swap	**941**	980	978	**14**	5s	1'897'387
	ea2048_1swap_5	948	980	982	15	2s	789'223
	ea4096_1swap_5	945	**976**	**975**	15	4s	1'601'925
swv15	ea2048_1swap	3395	3535	**3530**	78	128s	1'9290'521
	ea4096_1swap	3397	**3533**	3533	**54**	171s	25'073'630
	ea2048_1swap_5	**3390**	3545	3536	81	117s	15'999'092
	ea4096_1swap_5	3413	3543	3539	66	169s	22'266'887
yn4	ea2048_1swap	1032	1082	1082	22	26s	4'792'622
	ea4096_1swap	**1020**	1076	1074	21	39s	6'907'692
	ea2048_1swap_5	1027	1072	1072	19	19s	3'212'839
	ea4096_1swap_5	1034	**1068**	**1068**	**18**	37s	5'943'196

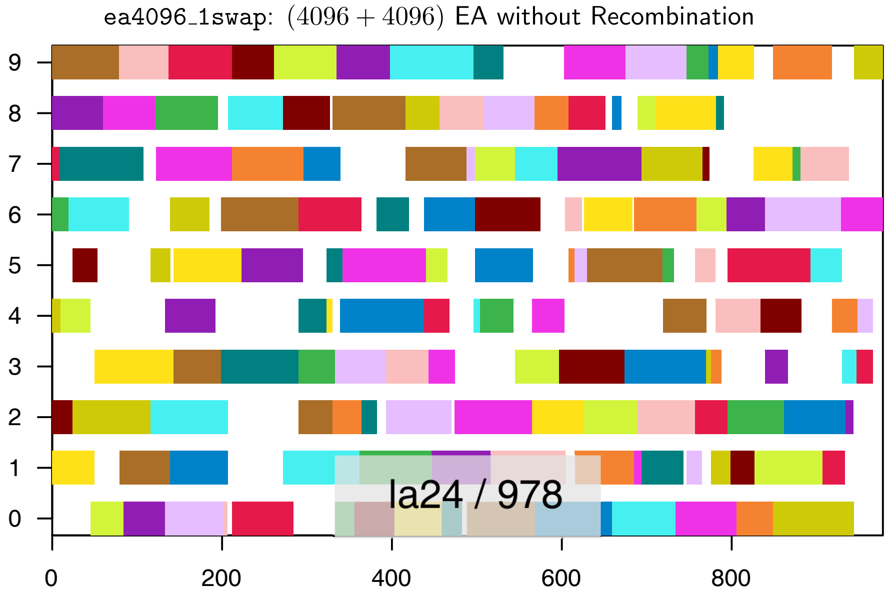
So what do we get?



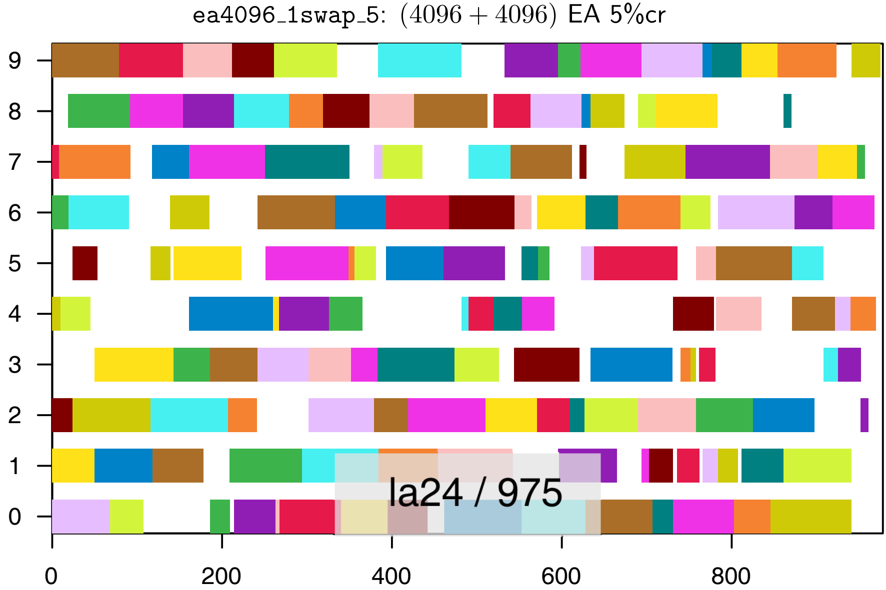
So what do we get?

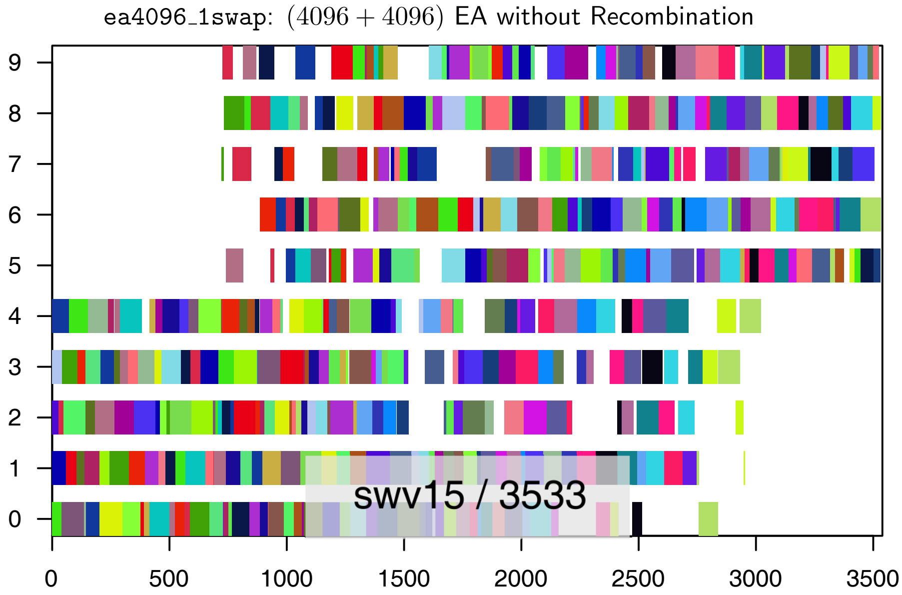


So what do we get?

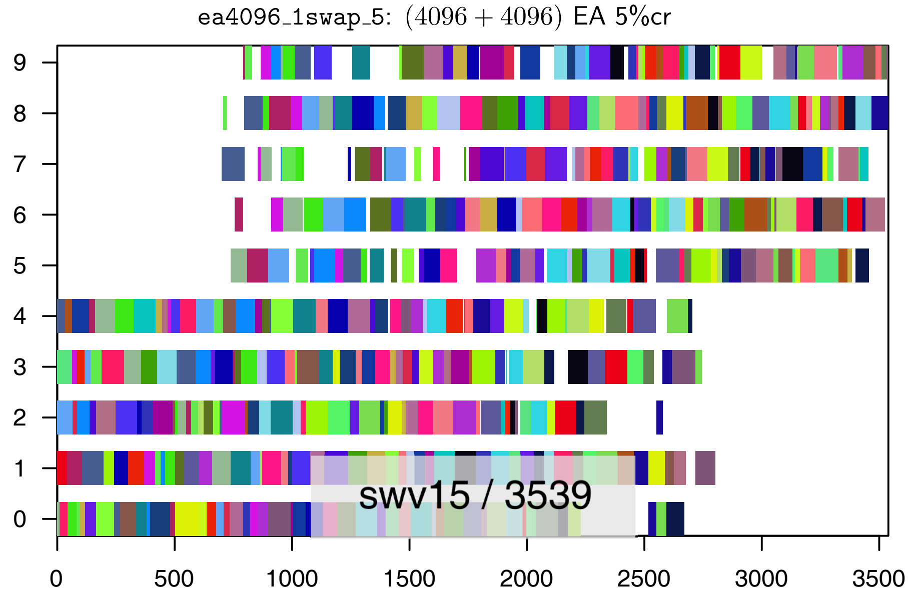


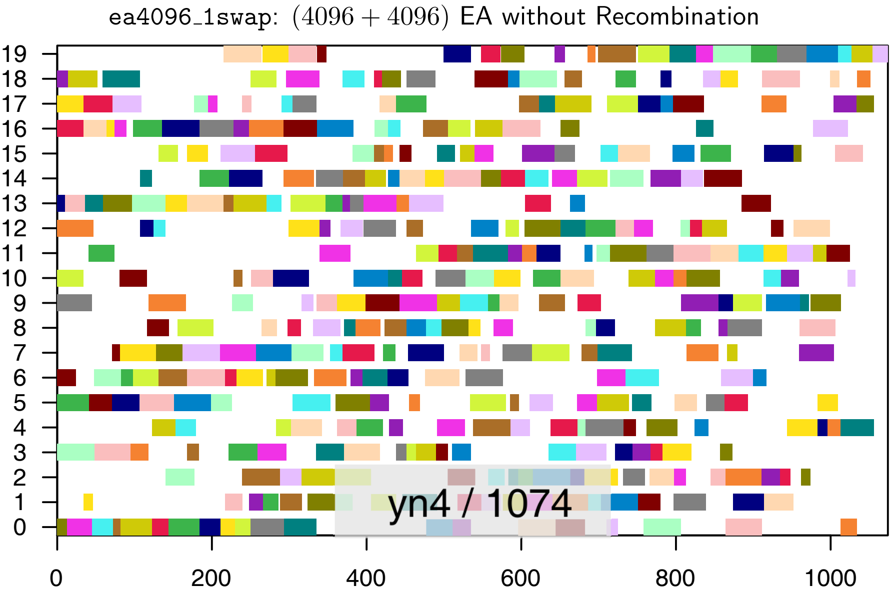
So what do we get?



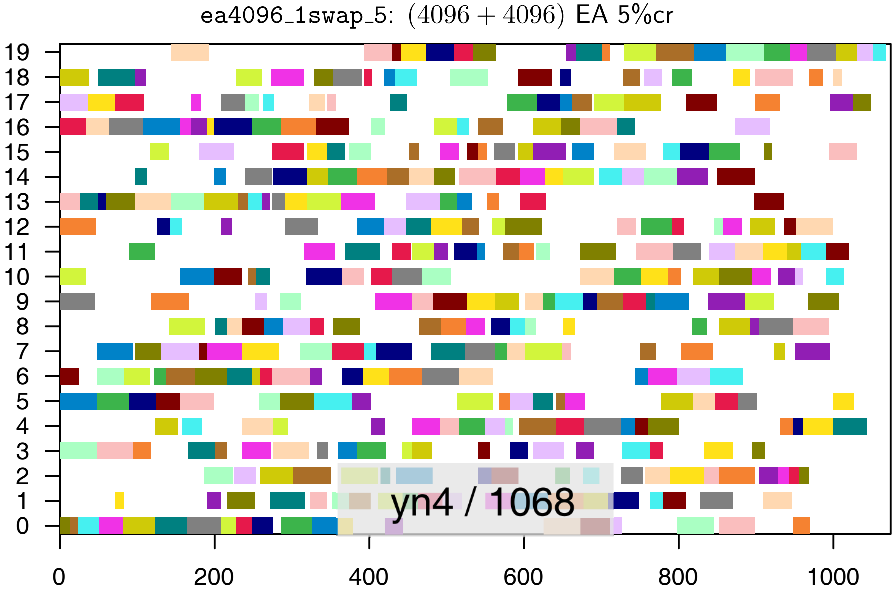

ea4096_1swap: $(4096+4096)$ EA without Recombination
9
8
7
6
5
4
3
2
1
0
0
500
1000
1500
2000
2500
3000
3500
swv15 / 3533

So what do we get?




ea4096_1swap: $(4096+4096)$ EA without Recombination
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
0
200
400
600
800
1000
yn4 / 1074

So what do we get?

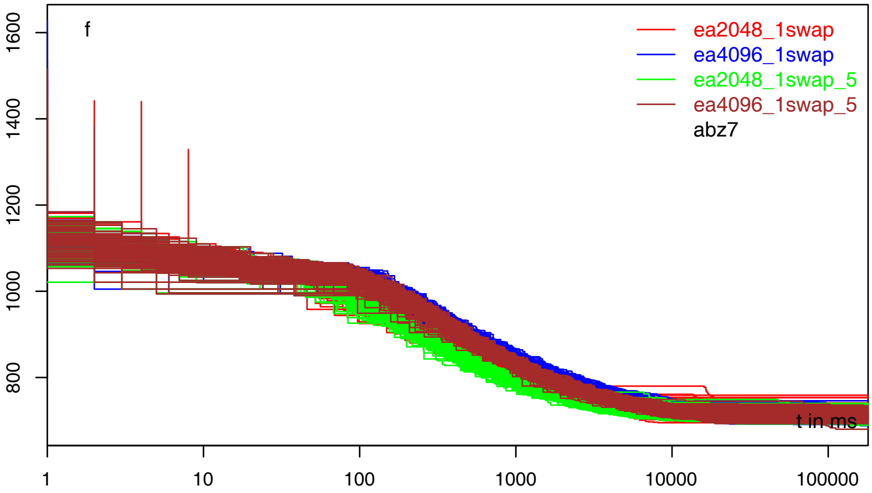


Progress over Time

What progress does the algorithm make over time?

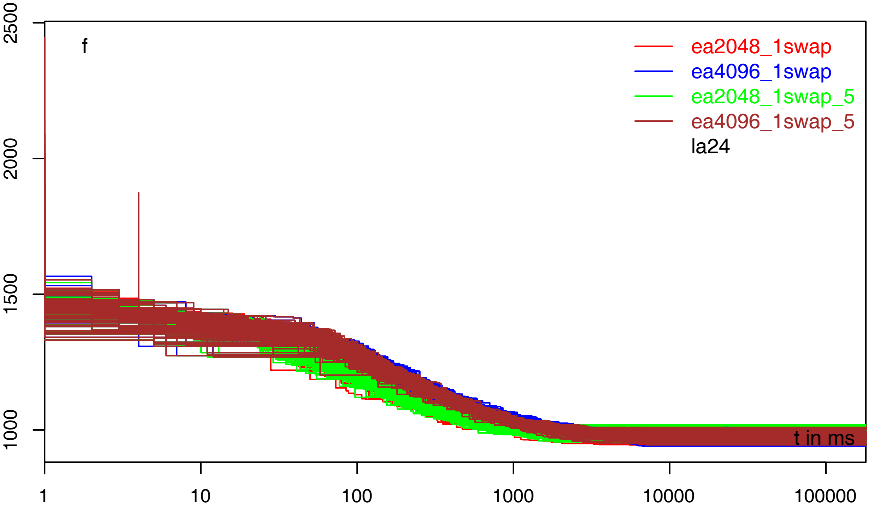
Progress over Time

What progress does the algorithm make over time?



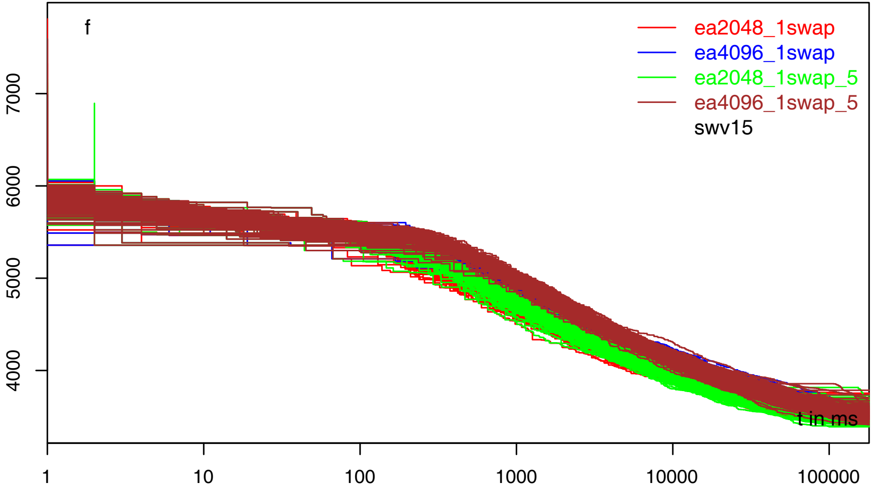
Progress over Time

What progress does the algorithm make over time?

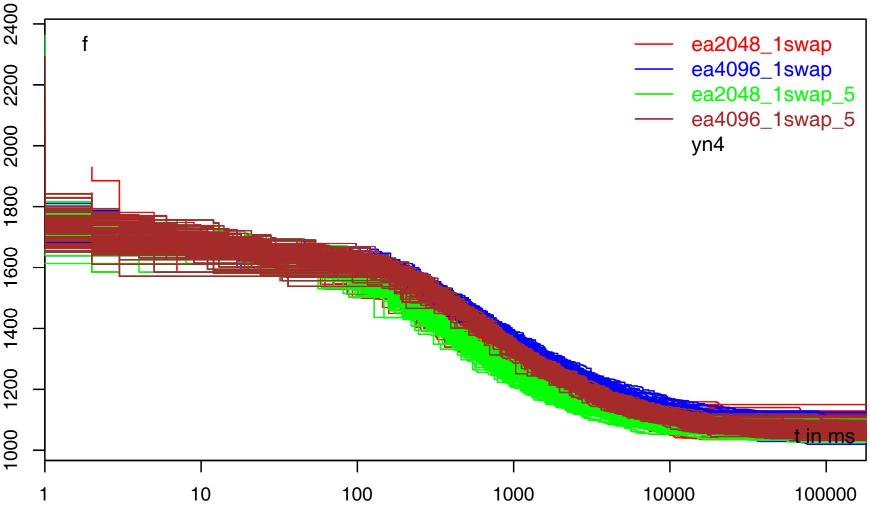


Progress over Time

What progress does the algorithm make over time?



What progress does the algorithm make over time?



Binary Operator Results

- The improvements that the binary operator offered us in this scenario are quite small.

Binary Operator Results

- The improvements that the binary operator offered us in this scenario are quite small.
- Nethertheless, in three out of four instances, it gives us better results.

Binary Operator Results

		makespan				last improvement	
$\mathcal{I}$	algo	best	mean	med	sd	med(t)	med(FEs)
abz7	ea2048_1swap	695	719	718	13	11s	2'581'614
	ea4096_1swap	688	716	716	12	19s	4'416'129
	ea2048_1swap_5	689	713	**712**	11	12s	2'641'808
	ea4096_1swap_5	**680**	**712**	**712**	**10**	20s	4'145'924
la24	ea2048_1swap	945	983	983	16	2s	927'000
	ea4096_1swap	**941**	980	978	**14**	5s	1'897'387
	ea2048_1swap_5	948	980	982	15	2s	789'223
	ea4096_1swap_5	945	**976**	**975**	15	4s	1'601'925
swv15	ea2048_1swap	3395	3535	**3530**	78	128s	1'9290'521
	ea4096_1swap	3397	**3533**	3533	**54**	171s	25'073'630
	ea2048_1swap_5	**3390**	3545	3536	81	117s	15'999'092
	ea4096_1swap_5	3413	3543	3539	66	169s	22'266'887
yn4	ea2048_1swap	1032	1082	1082	22	26s	4'792'622
	ea4096_1swap	**1020**	1076	1074	21	39s	6'907'692
	ea2048_1swap_5	1027	1072	1072	19	19s	3'212'839
	ea4096_1swap_5	1034	**1068**	**1068**	**18**	37s	5'943'196

- The improvements that the binary operator offered us in this scenario are quite small.
- Nethertheless, in three out of four instances, it gives us better results.
- In `swv15`, we already are close to the limit of the computational budget.

Binary Operator Results

- The improvements that the binary operator offered us in this scenario are quite small.
- Neththeless, in three out of four instances, it gives us better results.
- In `swv15`, we already are close to the limit of the computational budget, so it may be too late to reap any benefit from the binary operator.

Binary Operator Results

- The improvements that the binary operator offered us in this scenario are quite small.
- Nethertheless, in three out of four instances, it gives us better results.
- In `swv15`, we already are close to the limit of the computational budget, so it may be too late to reap any benefit from the binary operator.
- This is another lesson: Any statement about the result quality is only valid if accompanied by a statement about the consumed runtime!

Binary Operator Results

- The improvements that the binary operator offered us in this scenario are quite small.
- Nethertheless, in three out of four instances, it gives us better results.
- In `swv15`, we already are close to the limit of the computational budget, so it may be too late to reap any benefit from the binary operator.
- This is another lesson: Any statement about the result quality is only valid if accompanied by a statement about the consumed runtime!
- If I just let the algorithms run longer, I could probably report better results...

- The improvements that the binary operator offered us in this scenario are quite small.
- Nethertheless, in three out of four instances, it gives us better results.
- In `swv15`, we already are close to the limit of the computational budget, so it may be too late to reap any benefit from the binary operator.
- This is another lesson: Any statement about the result quality is only valid if accompanied by a statement about the consumed runtime!
- If I just let the algorithms run longer, I could probably report better results...
- ...which would be useless for our scenario, though.

Binary Operator Results

- The improvements that the binary operator offered us in this scenario are quite small.
- Nethertheless, in three out of four instances, it gives us better results.
- In `swv15`, we already are close to the limit of the computational budget, so it may be too late to reap any benefit from the binary operator.
- This is another lesson: Any statement about the result quality is only valid if accompanied by a statement about the consumed runtime!
- If I just let the algorithms run longer, I could probably report better results...
- ...which would be useless for our scenario, though.
- Anyway, overall, using the binary operator brings some gain.

Summary

- Population-based metaheuristics like Evolutionary Algorithms perform global search and can obtain better results than local searches like hill climbers.

Summary

- Population-based metaheuristics like Evolutionary Algorithms perform global search and can obtain better results than local searches like hill climbers.
- But they are also considerably slower.

Summary

- Population-based metaheuristics like Evolutionary Algorithms perform global search and can obtain better results than local searches like hill climbers.
- But they are also considerably slower.
- Sometimes, operators do not work as well as expected (e.g., the binary search operator here).

Summary

- Population-based metaheuristics like Evolutionary Algorithms perform global search and can obtain better results than local searches like hill climbers.
- But they are also considerably slower.
- Sometimes, operators do not work as well as expected (e.g., the binary search operator here).
- Sometimes, the reason may be that we just do not have enough time to benefit from it.

Summary

- Population-based metaheuristics like Evolutionary Algorithms perform global search and can obtain better results than local searches like hill climbers.
- But they are also considerably slower.
- Sometimes, operators do not work as well as expected (e.g., the binary search operator here).
- Sometimes, the reason may be that we just do not have enough time to benefit from it.
- This can be different for any optimization problem.

Summary

- Population-based metaheuristics like Evolutionary Algorithms perform global search and can obtain better results than local searches like hill climbers.
- But they are also considerably slower.
- Sometimes, operators do not work as well as expected (e.g., the binary search operator here).
- Sometimes, the reason may be that we just do not have enough time to benefit from it.
- This can be different for any optimization problem.
- Sometimes a different operator might work better.

Summary

- Population-based metaheuristics like Evolutionary Algorithms perform global search and can obtain better results than local searches like hill climbers.
- But they are also considerably slower.
- Sometimes, operators do not work as well as expected (e.g., the binary search operator here).
- Sometimes, the reason may be that we just do not have enough time to benefit from it.
- This can be different for any optimization problem.
- Sometimes a different operator might work better.
- This holds for *all* algorithm modules.

Summary

- Population-based metaheuristics like Evolutionary Algorithms perform global search and can obtain better results than local searches like hill climbers.
- But they are also considerably slower.
- Sometimes, operators do not work as well as expected (e.g., the binary search operator here).
- Sometimes, the reason may be that we just do not have enough time to benefit from it.
- This can be different for any optimization problem.
- Sometimes a different operator might work better.
- This holds for *all* algorithm modules.
- We always need to check whether the overall algorithm performs better with or without the module.

Summary

- Population-based metaheuristics like Evolutionary Algorithms perform global search and can obtain better results than local searches like hill climbers.
- But they are also considerably slower.
- Sometimes, operators do not work as well as expected (e.g., the binary search operator here).
- Sometimes, the reason may be that we just do not have enough time to benefit from it.
- This can be different for any optimization problem.
- Sometimes a different operator might work better.
- This holds for *all* algorithm modules.
- We always need to check whether the overall algorithm performs better with or without the module.
- ... but even small improvements might be worthwhile.

谢谢
Thank you

Thomas Weise [汤卫思]
tweise@hfuu.edu.cn
http://iao.hfuu.edu.cn

Hefei University, South Campus 2
Institute of Applied Optimization
Shushan District, Hefei, Anhui, China

Caspar David Friedrich, “Der Wanderer über dem Nebelmeer”, 1818
http://en.wikipedia.org/wiki/Wanderer_above_the_Sea_of_Fog

Bibliography

Bibliography I

1. Thomas Weise. *An Introduction to Optimization Algorithms*. Institute of Applied Optimization (IAO), Faculty of Computer Science and Technology, Hefei University, Hefei, Anhui, China, 2019-06-25 edition, 2018–2019. URL http://thomasweise.github.io/aitoa/. see also [2].
2. Thomas Weise. *Global Optimization Algorithms – Theory and Application*. it-weise.de (self-published), Germany, 2009. URL http://www.it-weise.de/projects/book.pdf.
3. John Henry Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. University of Michigan Press, Ann Arbor, MI, USA, 1975. ISBN 0-472-08460-7.
4. Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz, editors. *Handbook of Evolutionary Computation*. Computational Intelligence Library. Oxford University Press, Inc., New York, NY, USA, 1997. ISBN 0-7503-0392-1.
5. Zbigniew Michalewicz and David B. Fogel. *How to Solve It: Modern Heuristics*. Berlin/Heidelberg: Springer-Verlag, 2nd edition, 2004. ISBN 3-540-22494-7, 978-3-540-22494-5, and 978-3-642-06134-9. URL http://books.google.de/books?id=RJbV_-JlIUQC.
6. El-Ghazali Talbi. Population-based metaheuristics. In *Metaheuristics: From Design to Implementation*, chapter 3, pages 190–307. John Wiley & Sons, June 2009. ISBN 9780470278581. doi: 10.1002/9780470496916.ch3.
7. David Edward Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989. ISBN 0-201-15767-5.
8. Thomas Weise, Yuezhong Wu, Raymond Chiong, Ke Tang, and Jörg Lässig. Global versus local search: The impact of population sizes on evolutionary algorithm performance. *Journal of Global Optimization*, 66:511–534, November 2016. doi: 10.1007/s10898-016-0417-5.
9. Ágoston E. Eiben and C. A. Schippers. On evolutionary exploration and exploitation. *Fundamenta Informaticae – Annales Societatis Mathematicae Polonae, Series IV*, 35(1-2):35–50, July–August 1998. doi: 10.3233/FI-1998-35123403. URL http://www.cs.vu.nl/~gusz/papers/FunInf98-Eiben-Schippers.ps.
10. Thomas Weise, Raymond Chiong, and Ke Tang. Evolutionary optimization: Pitfalls and booby traps. *Journal of Computer Science and Technology (JCST)*, 27:907–936, September 2012. doi: 10.1007/s11390-012-1274-4.
11. Thomas Weise, Michael Zapf, Raymond Chiong, and Antonio Jesús Nebro Urbaneja. Why is optimization difficult? In Raymond Chiong, editor, *Nature-Inspired Algorithms for Optimisation*, volume 193/2009 of *Studies in Computational Intelligence (SCI)*, chapter 1, pages 1–50. Springer-Verlag, Berlin/Heidelberg, April 2009. ISBN 978-3-642-00266-3. doi: 10.1007/978-3-642-00267-0_1.