合肥学院
HEFEI UNIVERSITY

Optimization Algorithms

5. Stochastic Hill Climbing

Thomas Weise · 汤卫思
tweise@hfuu.edu.cn · http://iao.hfuu.edu.cn/5

Institute of Applied Optimization (IAO) | 应用优化研究所
School of Artificial Intelligence and Big Data | 人工智能与大数据学院
Hefei University | 合肥学院
Hefei, Anhui, China | 中国安徽省合肥市

Outline

Introduction

Information from Good Solutions

- Our first algorithm, random sampling, was not very efficient.

Information from Good Solutions

- Our first algorithm, random sampling, was not very efficient.
- It does not make any use of the information it "sees" during the optimization process.

Information from Good Solutions

- Our first algorithm, random sampling, was not very efficient.
- It does not make any use of the information it "sees" during the optimization process.
- Each search step consists of creating an entirely new, entirely random candidate solution.

Information from Good Solutions

- Our first algorithm, random sampling, was not very efficient.
- It does not make any use of the information it “sees” during the optimization process.
- Each search step consists of creating an entirely new, entirely random candidate solution.
- Each search step is thus independent of all prior steps.

Information from Good Solutions

- Our first algorithm, random sampling, was not very efficient.
- It does not make any use of the information it "sees" during the optimization process.
- Each search step consists of creating an entirely new, entirely random candidate solution.
- Each search step is thus independent of all prior steps.
- Is this a good idea?

Information from Good Solutions

- Our first algorithm, random sampling, was not very efficient.
- It does not make any use of the information it “sees” during the optimization process.
- Each search step consists of creating an entirely new, entirely random candidate solution.
- Each search step is thus independent of all prior steps.
- Is this a good idea?
- Probably not.

Information from Good Solutions

- Our first algorithm, random sampling, was not very efficient.
- It does not make any use of the information it "sees" during the optimization process.
- Each search step consists of creating an entirely new, entirely random candidate solution.
- Each search step is thus independent of all prior steps.
- Is this a good idea?
- Probably not.
- In almost all practical scenarios, good solutions are somewhat similar to other good solutions.

Information from Good Solutions

- Our first algorithm, random sampling, was not very efficient.
- It does not make any use of the information it “sees” during the optimization process.
- Each search step consists of creating an entirely new, entirely random candidate solution.
- Each search step is thus independent of all prior steps.
- Is this a good idea?
- Probably not.
- In almost all practical scenarios, good solutions are somewhat similar to other good solutions.
- In other words, every good solution we see is some useful information.

Basic Idea

- So how we can make use of the information we have seen during the search?

Basic Idea

- So how we can make use of the information we have seen during the search?

Basic Idea

- So how we can make use of the information we have seen during the search?
- Instead of generating a completely random new candidate solution in each step. . .

Basic Idea

- So how we can make use of the information we have seen during the search?
- Instead of generating a completely random new candidate solution in each step...
- ...why can't we try to iteratively improve the best solution we have seen so far?

Algorithm Concept

Stochastic Hill Climbing

- This is the concept of Local Search[2–5] and its simplest realization is Stochastic Hill Climbing[2].

Stochastic Hill Climbing

- This is the concept of Local Search[2–5] and its simplest realization is Stochastic Hill Climbing[2].
- Simple Concept

Stochastic Hill Climbing

- This is the concept of Local Search[2–5] and its simplest realization is Stochastic Hill Climbing[2].
- Simple Concept:
    1. create random initial solution

Stochastic Hill Climbing

- This is the concept of Local Search[2–5] and its simplest realization is Stochastic Hill Climbing[2].
- Simple Concept:
  1. create random initial solution
  2. make a modified copy of best-so-far solution

Stochastic Hill Climbing

- This is the concept of Local Search[2–5] and its simplest realization is Stochastic Hill Climbing[2].
- Simple Concept:
    1. create random initial solution
    2. make a modified copy of best-so-far solution
    3. if it is better, it becomes the new best-so-far solution (if it is not better, discard it).

Stochastic Hill Climbing

- This is the concept of Local Search[2–5] and its simplest realization is Stochastic Hill Climbing[2].
- Simple Concept:
  1. create random initial solution
  2. make a modified copy of best-so-far solution
  3. if it is better, it becomes the new best-so-far solution (if it is not better, discard it).
  4. go back to 2. (until the time is up)

Implementation of the Stochastic Hill Climber

```
package aitoa.algorithms;

public class HillClimber<X, Y> {
// unnecessary stuff omitted here...
//
//
//
//

//
//

//
//
//
//
//
//
//
//
//
}
```

Implementation of the Stochastic Hill Climber

```
package aitoa.algorithms;

public class HillClimber<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
//
//
//

//
//

//
//
//
//
//
//
//
//
  }
}
```

Implementation of the Stochastic Hill Climber

```
package aitoa.algorithms;

public class HillClimber<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
      X xCur  = process.getSearchSpace().create();
//
//

//
//

//
//
//
//
//
//
//
//
  }
}
```

Implementation of the Stochastic Hill Climber

```
package aitoa.algorithms;

public class HillClimber<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
      X xCur  = process.getSearchSpace().create();
      X xBest = process.getSearchSpace().create();
//

//
//

//
//
//
//
//
//
//
//
  }
}
```

Implementation of the Stochastic Hill Climber

```
package aitoa.algorithms;

public class HillClimber<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
      X xCur  = process.getSearchSpace().create();
      X xBest = process.getSearchSpace().create();
      Random random = process.getRandom();

//
//

//
//
//
//
//
//
//
//
  }
}
```

Implementation of the Stochastic Hill Climber

```
package aitoa.algorithms;

public class HillClimber<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
      X xCur  = process.getSearchSpace().create();
      X xBest = process.getSearchSpace().create();
      Random random = process.getRandom();

      this.nullary.apply(xBest, random);
//

//
//
//
//
//
//
//
//
  }
}
```

Implementation of the Stochastic Hill Climber

```
package aitoa.algorithms;

public class HillClimber<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
     X xCur  = process.getSearchSpace().create();
     X xBest = process.getSearchSpace().create();
     Random random = process.getRandom();

     this.nullary.apply(xBest, random);
     double fBest = process.evaluate(xBest);

//
//
//
//
//
//
//
//
  }
}
```

Implementation of the Stochastic Hill Climber

```
package aitoa.algorithms;

public class HillClimber<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
     X xCur  = process.getSearchSpace().create();
     X xBest = process.getSearchSpace().create();
     Random random = process.getRandom();

     this.nullary.apply(xBest, random);
     double fBest = process.evaluate(xBest);

//
     this.unary.apply(xBest, xCur, random);
//
//
//
//
//
//
  }
}
```

Implementation of the Stochastic Hill Climber

```
package aitoa.algorithms;

public class HillClimber<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
     X xCur  = process.getSearchSpace().create();
     X xBest = process.getSearchSpace().create();
     Random random = process.getRandom();

     this.nullary.apply(xBest, random);
     double fBest = process.evaluate(xBest);

//
     this.unary.apply(xBest, xCur, random);
     double fCur = process.evaluate(xCur);
//
//
//
//
//
  }
}
```

Implementation of the Stochastic Hill Climber

```java
package aitoa.algorithms;

public class HillClimber<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
      X xCur  = process.getSearchSpace().create();
      X xBest = process.getSearchSpace().create();
      Random random = process.getRandom();

      this.nullary.apply(xBest, random);
      double fBest = process.evaluate(xBest);

//
      this.unary.apply(xBest, xCur, random);
      double fCur = process.evaluate(xCur);
      if (fCur < fBest) {
//
//
      }
//
  }
}
```

Implementation of the Stochastic Hill Climber

```
package aitoa.algorithms;

public class HillClimber<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
      X xCur  = process.getSearchSpace().create();
      X xBest = process.getSearchSpace().create();
      Random random = process.getRandom();

      this.nullary.apply(xBest, random);
      double fBest = process.evaluate(xBest);

//
      this.unary.apply(xBest, xCur, random);
      double fCur = process.evaluate(xCur);
      if (fCur < fBest) {
        fBest = fCur;
//
      }
//
  }
}
```

Implementation of the Stochastic Hill Climber

```java
package aitoa.algorithms;

public class HillClimber<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
      X xCur  = process.getSearchSpace().create();
      X xBest = process.getSearchSpace().create();
      Random random = process.getRandom();

      this.nullary.apply(xBest, random);
      double fBest = process.evaluate(xBest);

//
      this.unary.apply(xBest, xCur, random);
      double fCur = process.evaluate(xCur);
      if (fCur < fBest) {
        fBest = fCur;
        process.getSearchSpace().copy(xCur, xBest);
      }
//
  }
}
```

Implementation of the Stochastic Hill Climber

```
package aitoa.algorithms;

public class HillClimber<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
      X xCur  = process.getSearchSpace().create();
      X xBest = process.getSearchSpace().create();
      Random random = process.getRandom();

      this.nullary.apply(xBest, random);
      double fBest = process.evaluate(xBest);

      while (!process.shouldTerminate()) {
        this.unary.apply(xBest, xCur, random);
        double fCur = process.evaluate(xCur);
        if (fCur < fBest) {
          fBest = fCur;
          process.getSearchSpace().copy(xCur, xBest);
        }
      }
  }
}
```

Causality

- Local searches like hill climbers exploit a property of many optimization problems called causality[6–9].

Causality

- Local searches like hill climbers exploit a property of many optimization problems called causality[6–9].
- Causality means that small changes in the features of an object (or candidate solution) also lead to small changes in its behavior (or objective value).

Causality

- Local searches like hill climbers exploit a property of many optimization problems called causality[6–9].
- Causality means that small changes in the features of an object (or candidate solution) also lead to small changes in its behavior (or objective value).
- If an optimization problem exhibits causality, then there should be good solutions that are similar to other good solutions.

Causality

- Local searches like hill climbers exploit a property of many optimization problems called causality[6–9].
- Causality means that small changes in the features of an object (or candidate solution) also lead to small changes in its behavior (or objective value).
- If an optimization problem exhibits causality, then there should be good solutions that are similar to other good solutions.
- The idea is that if we have a good candidate solution, then there may exist similar solutions which are better.

Causality

- Local searches like hill climbers exploit a property of many optimization problems called causality[6–9].
- Causality means that small changes in the features of an object (or candidate solution) also lead to small changes in its behavior (or objective value).
- If an optimization problem exhibits causality, then there should be good solutions that are similar to other good solutions.
- The idea is that if we have a good candidate solution, then there may exist similar solutions which are better.
- We hope to find one of them and then continue trying to do the same from there.

Ingredient: Unary Search Operator

Unary Search Operator

- Our hill climber must be able to make modified copies of an existing point $x \in \mathbb{X}$ in order to find these better points.

Unary Search Operator

- Our hill climber must be able to make modified copies of an existing point $x \in \mathbb{X}$ in order to find these better points.
- A unary search operator accepts on existing point $x \in \mathbb{X}$ and creates a modified copy of it.

Unary Search Operator

- Our hill climber must be able to make modified copies of an existing point $x \in \mathbb{X}$ in order to find these better points.
- A unary search operator accepts on existing point $x \in \mathbb{X}$ and creates a modified copy of it.
- It must make sure that the modified copy is still a valid element of $\mathbb{X}$.

Unary Search Operator

- Our hill climber must be able to make modified copies of an existing point $x \in \mathbb{X}$ in order to find these better points.
- A unary search operator accepts on existing point $x \in \mathbb{X}$ and creates a modified copy of it.
- It must make sure that the modified copy is still a valid element of $\mathbb{X}$.
- It should ideally be randomized, i.e., applying it twice to the same point x should yield different results.

Unary Search Operator

- Our hill climber must be able to make modified copies of an existing point $x \in \mathbb{X}$ in order to find these better points.
- A unary search operator accepts on existing point $x \in \mathbb{X}$ and creates a modified copy of it.
- It must make sure that the modified copy is still a valid element of $\mathbb{X}$.
- It should ideally be randomized, i.e., applying it twice to the same point x should yield different results.

```
package aitoa.structure;

public interface IUnarySearchOperator<X> {

  void apply(X x, X dest, Random random);

}
```

Unary Search Operator

- Our hill climber must be able to make modified copies of an existing point $x \in \mathbb{X}$ in order to find these better points.
- A unary search operator accepts on existing point $x \in \mathbb{X}$ and creates a modified copy of it.
- It must make sure that the modified copy is still a valid element of $\mathbb{X}$.
- It should ideally be randomized, i.e., applying it twice to the same point x should yield different results.
- How can we implement this for our JSSP scenario?

Unary Search Operator

- Our hill climber must be able to make modified copies of an existing point $x \in \mathbb{X}$ in order to find these better points.
- A unary search operator accepts on existing point $x \in \mathbb{X}$ and creates a modified copy of it.
- It must make sure that the modified copy is still a valid element of $\mathbb{X}$.
- It should ideally be randomized, i.e., applying it twice to the same point x should yield different results.
- How can we implement this for our JSSP scenario?
- Easy: Just swap two (different) job IDs in the string!

Unary Search Operator

- Our hill climber must be able to make modified copies of an existing point $x \in \mathbb{X}$ in order to find these better points.
- A unary search operator accepts on existing point $x \in \mathbb{X}$ and creates a modified copy of it.
- It must make sure that the modified copy is still a valid element of $\mathbb{X}$.
- It should ideally be randomized, i.e., applying it twice to the same point x should yield different results.
- How can we implement this for our JSSP scenario?
- Easy: Just swap two (different) job IDs in the string!
- Since the numbers of occurrences of the IDs will not change, the new strings will be valid.

Example for our `1swap` Operator

$\mathbb{X}$ (2,0,1,0,1,1,2,3,2,3,
2,0,0,1,3,3,2,3,1,0)

Example for our `1swap` Operator

$\mathbb{X}$ (2,0,1,0,1,1,2,3,2,3,
2,0,0,1,3,3,2,3,1,0)

$\downarrow \gamma$

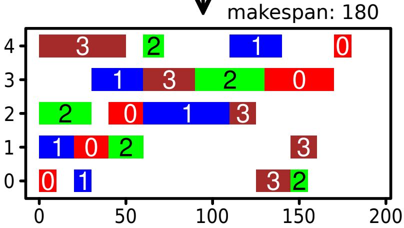
$\mathbb{Y}$

Example for our `1swap` Operator

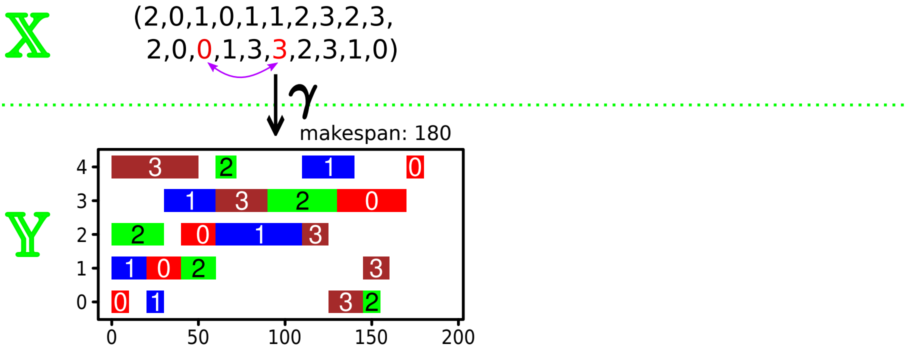
$\mathbb{X}$ (2,0,1,0,1,1,2,3,2,3,
2,0,0,1,3,3,2,3,1,0)

$\downarrow \gamma$

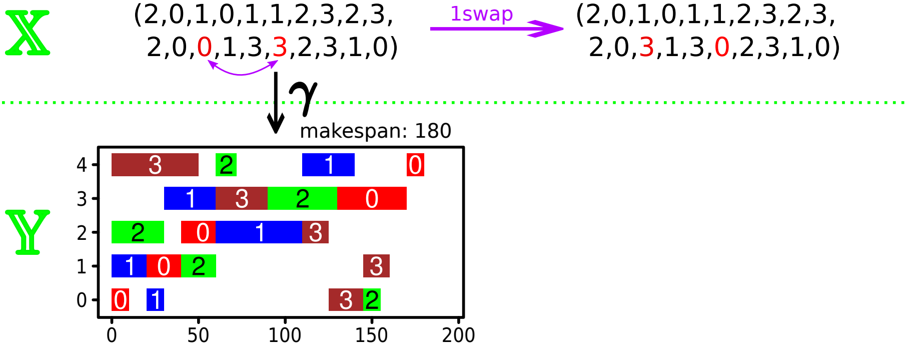
$\mathbb{Y}$



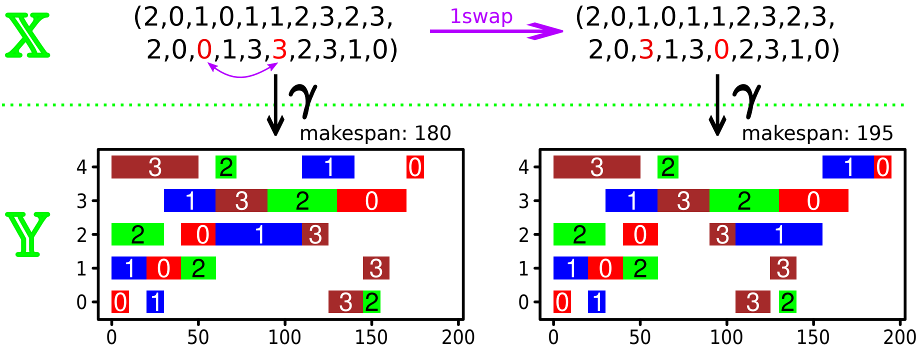
Example for our `1swap` Operator



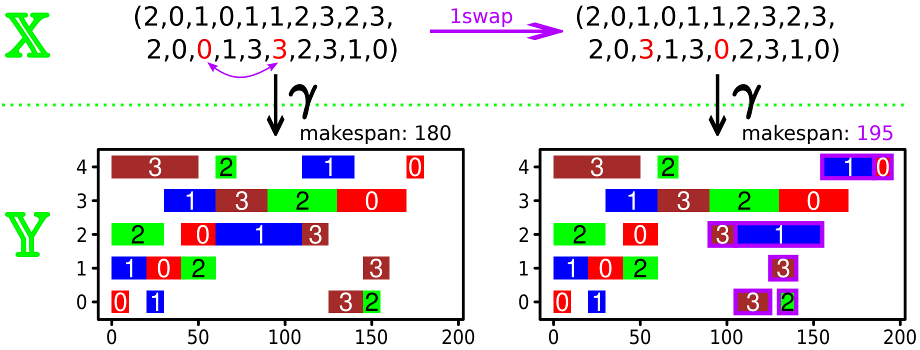
Example for our `1swap` Operator



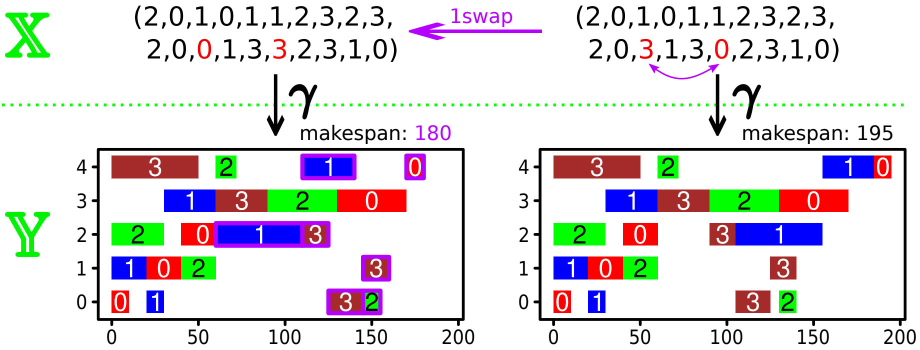
Example for our `1swap` Operator



Example for our `1swap` Operator



Example for our `1swap` Operator



```java
package aitoa.examples.jssp;

public class JSSPUnaryOperator1Swap {
//
// unnecessary stuff omitted here...
//
//
//

//
//
//

//
//
//
//
//
//
//
//
//
//
}
```

```java
package aitoa.examples.jssp;

public class JSSPUnaryOperator1Swap implements
    IUnarySearchOperator<int[]> {
// unnecessary stuff omitted here...
//
//
//

//
//
//

//
//
//
//
//
//
//
//
//
//
}
```

```
package aitoa.examples.jssp;

public class JSSPUnaryOperator1Swap implements
    IUnarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] x, int[] dest, Random random) {
//
//

//
//
//

//
//
//
//
//
//
//
//
//
  }
}
```

```java
package aitoa.examples.jssp;

public class JSSPUnaryOperator1Swap implements
    IUnarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] x, int[] dest, Random random) {
// copy the source point in search space to the dest
    System.arraycopy(x, 0, dest, 0, x.length);

//
//
//

//
//
//
//
//
//
//
//
//
  }
}
```

```java
package aitoa.examples.jssp;

public class JSSPUnaryOperator1Swap implements
    IUnarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] x, int[] dest, Random random) {
// copy the source point in search space to the dest
    System.arraycopy(x, 0, dest, 0, x.length);

// choose the index of the first operation to swap
    int i    = random.nextInt(dest.length);
    int jobI = dest[i]; // remember job id

//
//
//
//
//
//
//
//
//
  }
}
```

```java
package aitoa.examples.jssp;

public class JSSPUnaryOperator1Swap implements
    IUnarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] x, int[] dest, Random random) {
// copy the source point in search space to the dest
    System.arraycopy(x, 0, dest, 0, x.length);

// choose the index of the first operation to swap
    int i    = random.nextInt(dest.length);
    int jobI = dest[i]; // remember job id

// choose index of second operation to swap
    int j = random.nextInt(dest.length);
    int jobJ = dest[j];
//
//
//
//
//
//
  }
}
```

```java
package aitoa.examples.jssp;

public class JSSPUnaryOperator1Swap implements
    IUnarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] x, int[] dest, Random random) {
// copy the source point in search space to the dest
    System.arraycopy(x, 0, dest, 0, x.length);

// choose the index of the first operation to swap
    int i     = random.nextInt(dest.length);
    int jobI = dest[i]; // remember job id

// choose index of second operation to swap
    int j = random.nextInt(dest.length);
    int jobJ = dest[j];
//
    dest[i] = jobJ;
    dest[j] = jobI;    // then we swap the values
//
//
//
  }
}
```

```java
package aitoa.examples.jssp;

public class JSSPUnaryOperator1Swap implements
    IUnarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] x, int[] dest, Random random) {
// copy the source point in search space to the dest
    System.arraycopy(x, 0, dest, 0, x.length);

// choose the index of the first operation to swap
    int i    = random.nextInt(dest.length);
    int jobI = dest[i]; // remember job id

// choose index of second operation to swap
    int j = random.nextInt(dest.length);
    int jobJ = dest[j];
    if (jobI != jobJ) { // we found two locations with two
      dest[i] = jobJ;   // different values
      dest[j] = jobI;   // then we swap the values
//
    }
//
  }
}
```

```java
package aitoa.examples.jssp;

public class JSSPUnaryOperator1Swap implements
    IUnarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] x, int[] dest, Random random) {
// copy the source point in search space to the dest
    System.arraycopy(x, 0, dest, 0, x.length);

// choose the index of the first sub-job to swap
    int i    = random.nextInt(dest.length);
    int jobI = dest[i]; // remember job id

    for (;;) { // try to find a location j with a different job
      int j = random.nextInt(dest.length);
      int jobJ = dest[j];
      if (jobI != jobJ) { // we found two locations with two
        dest[i] = jobJ;   // different values
        dest[j] = jobI;   // then we swap the values
        return;           // and are done
      }
    }
  }
}
```

Experiment and Analysis

So what do we get?

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

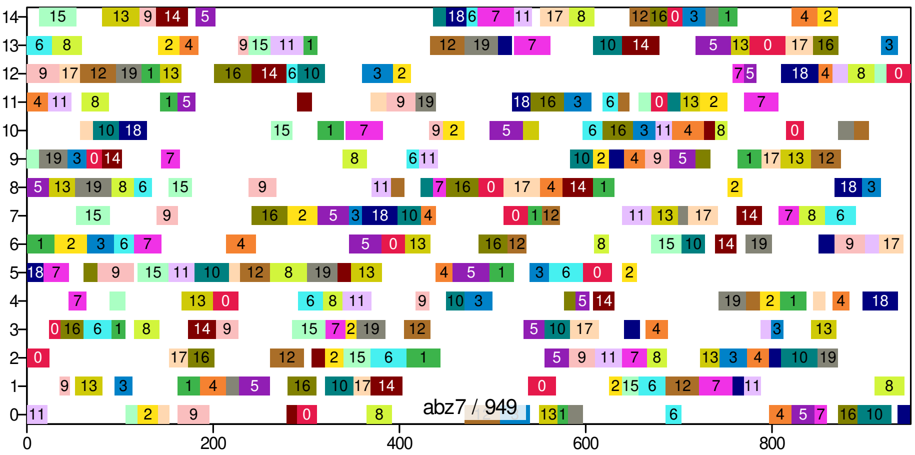
So what do we get?

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

		makespan				last improvement	
$\mathcal{I}$	algo	best	mean	med	sd	med(t)	med(FEs)
`abz7`	`rs`	895	947	949	**12**	85s	6'512'505
	`hc_1swap`	**717**	**800**	**798**	28	0s	16'978
`la24`	`rs`	1153	1206	1208	**15**	82s	15'902'911
	`hc_1swap`	**999**	**1095**	**1086**	56	0s	6'612
`swv15`	`rs`	4988	5166	5172	**50**	87s	5'559'124
	`hc_1swap`	**3837**	**4108**	**4108**	137	1s	104'598
`yn4`	`rs`	1460	1498	1499	**15**	76s	4'814'914
	`hc_1swap`	**1109**	**1222**	**1220**	48	0s	31'789

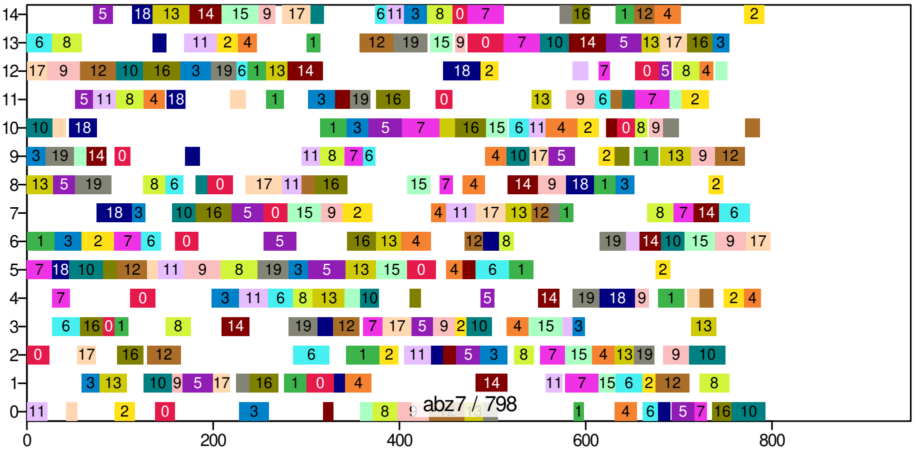
So what do we get?

`rs`: median result of 3 min of random sampling



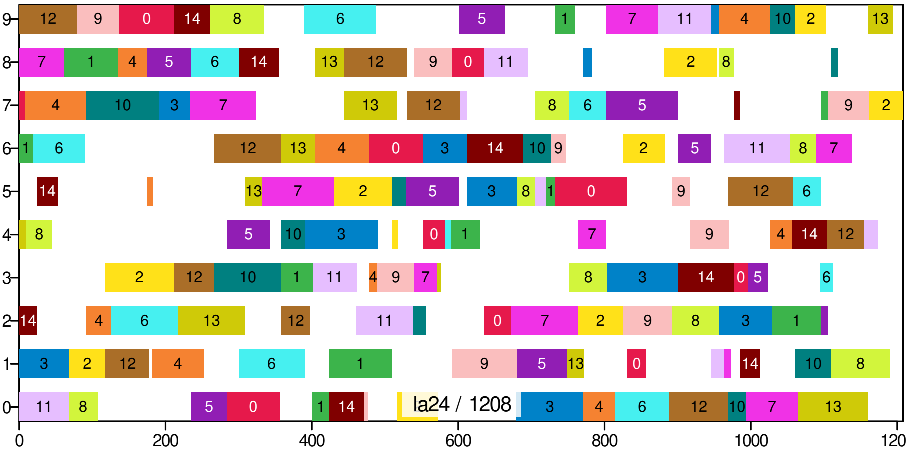
So what do we get?

hc_1swap: median result of 3 min of hill climber



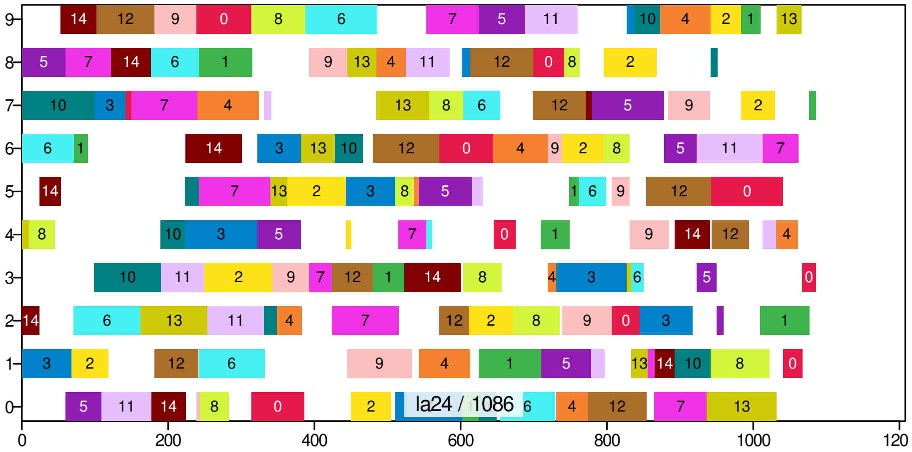
So what do we get?

`rs`: median result of 3 min of random sampling



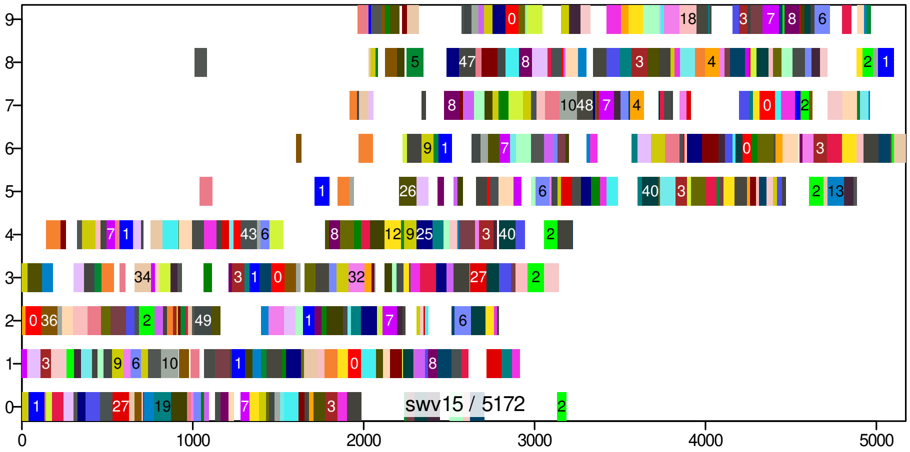
So what do we get?

`hc_1swap`: median result of 3 min of hill climber



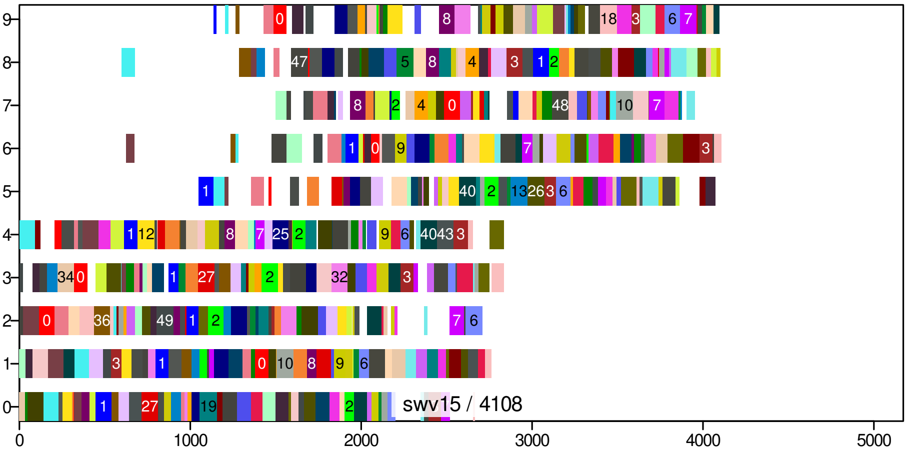
So what do we get?

`rs`: median result of 3 min of random sampling



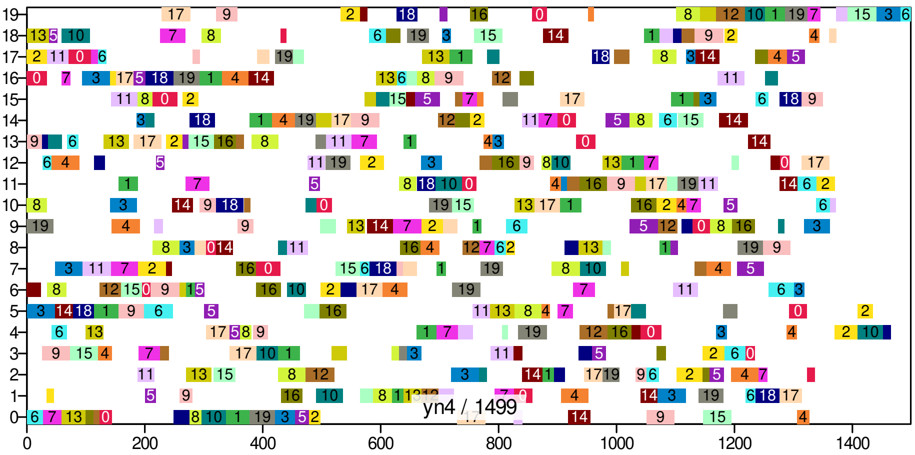
So what do we get?

hc_1swap: median result of 3 min of hill climber

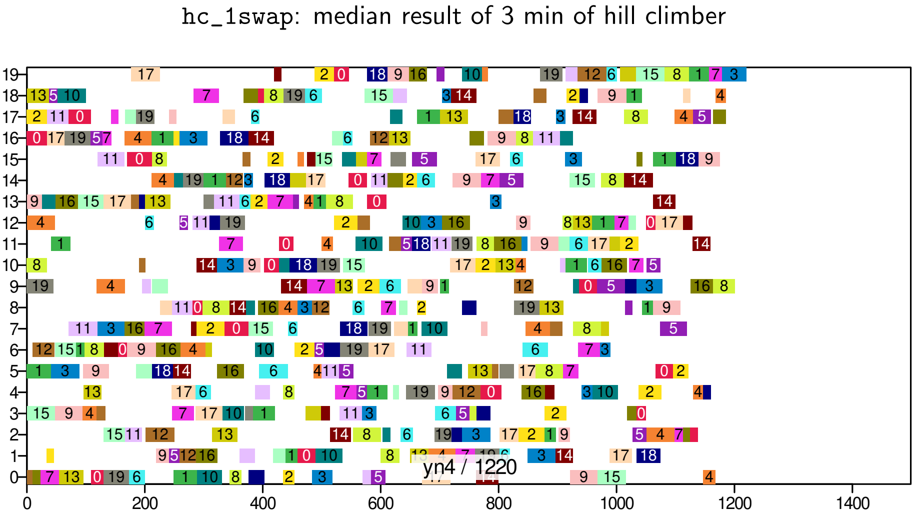


So what do we get?

`rs`: median result of 3 min of random sampling



So what do we get?

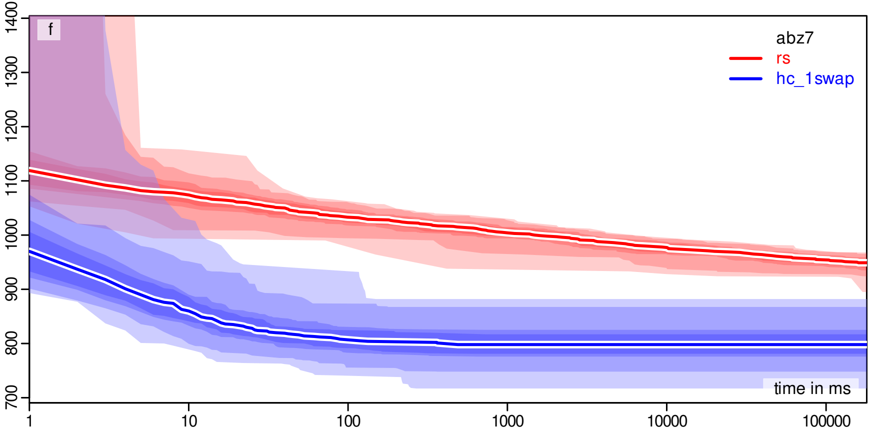


Progress over Time

What progress does the algorithm make over time?

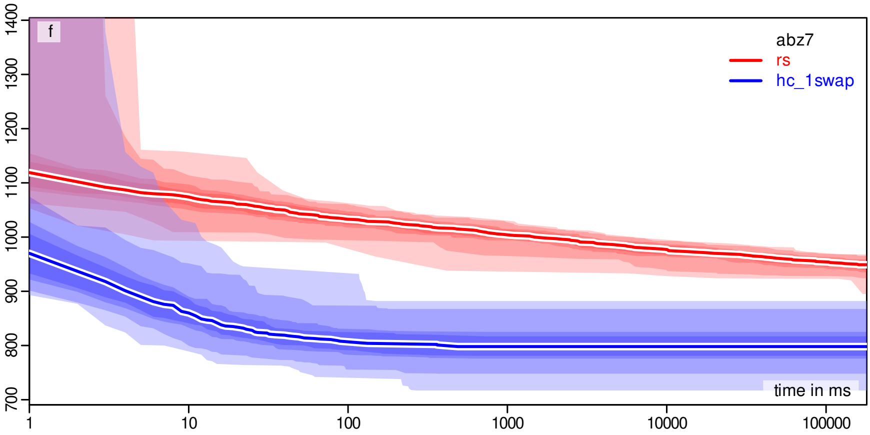
Progress over Time

What progress does the algorithm make over time?



Progress over Time

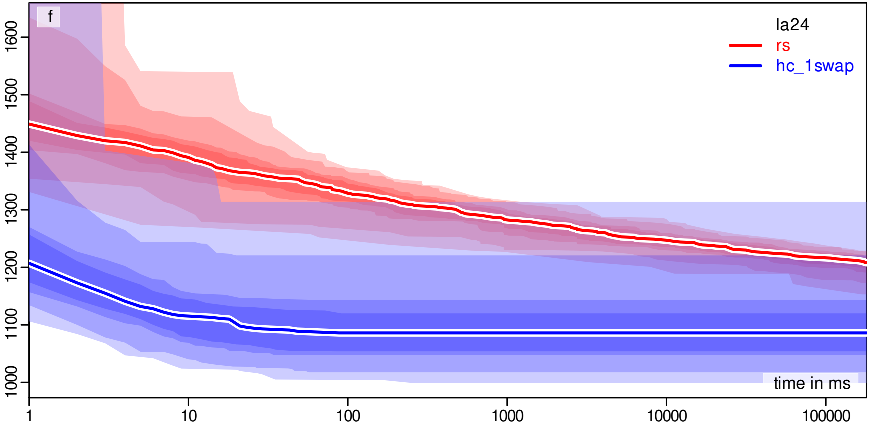
What progress does the algorithm make over time?



First we have much progress. . .

Progress over Time

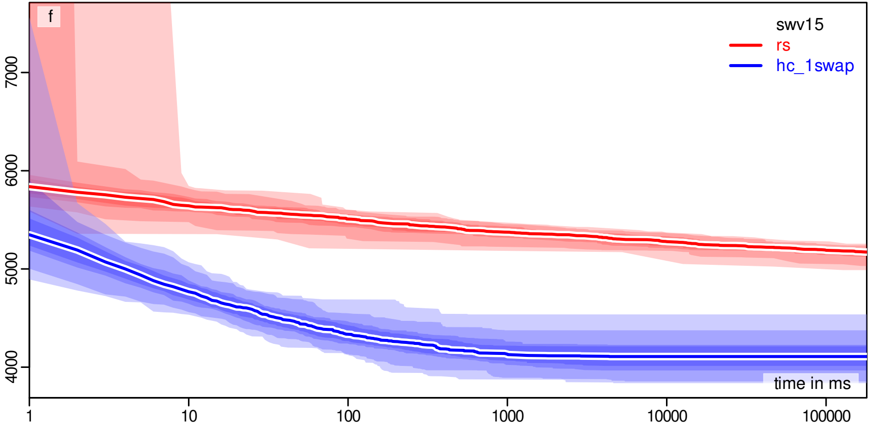
What progress does the algorithm make over time?



First we have much progress...

Progress over Time

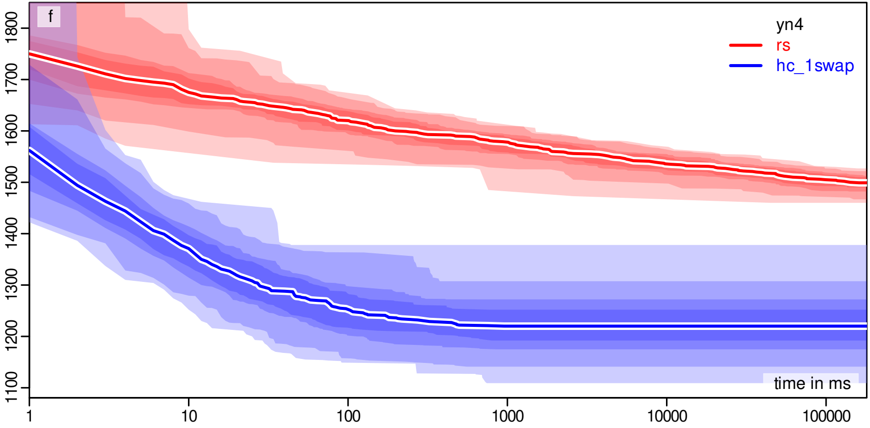
What progress does the algorithm make over time?



First we have much progress. . .

Progress over Time

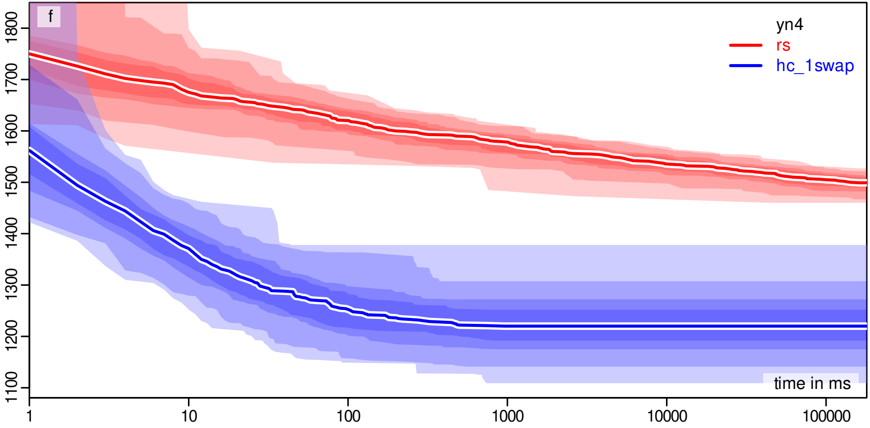
What progress does the algorithm make over time?



First we have much progress. . .

Progress over Time

What progress does the algorithm make over time?



First we have much progress. . .

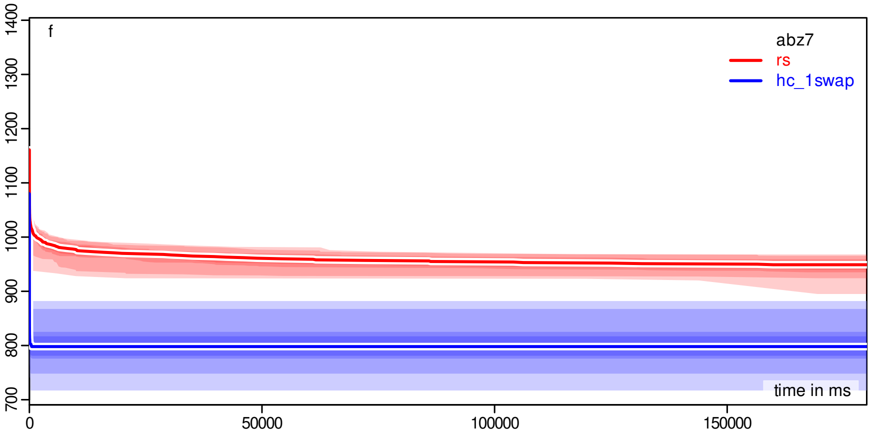
. . . but then the hill climber stagnates!

But we waste time. . .

What if we look at this without log-scaling the time axis?

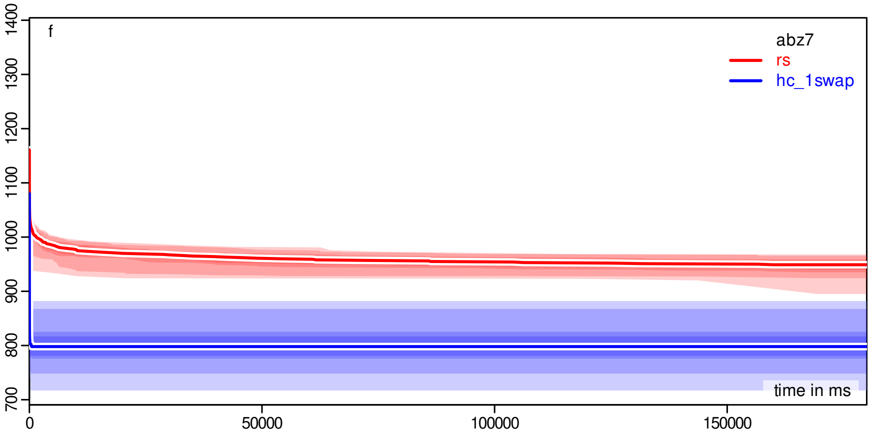
But we waste time. . .

What if we look at this without log-scaling the time axis?



But we waste time...

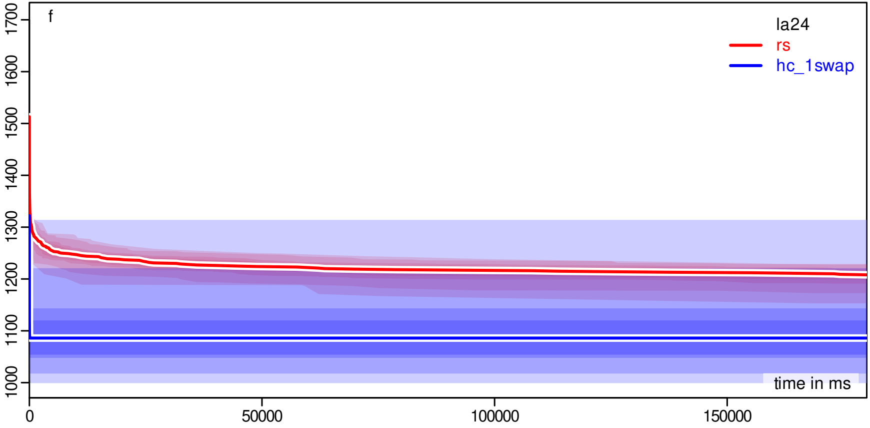
What if we look at this without log-scaling the time axis?



Then it looks even much worse!

But we waste time. . .

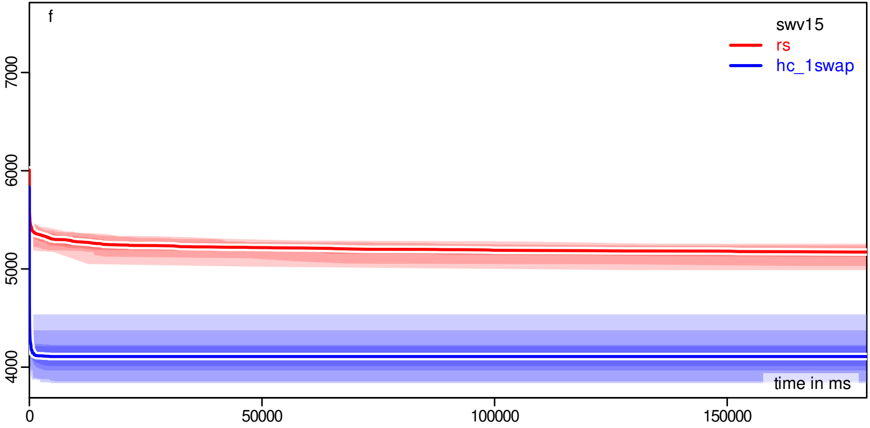
What if we look at this without log-scaling the time axis?



Then it looks even much worse!

But we waste time...

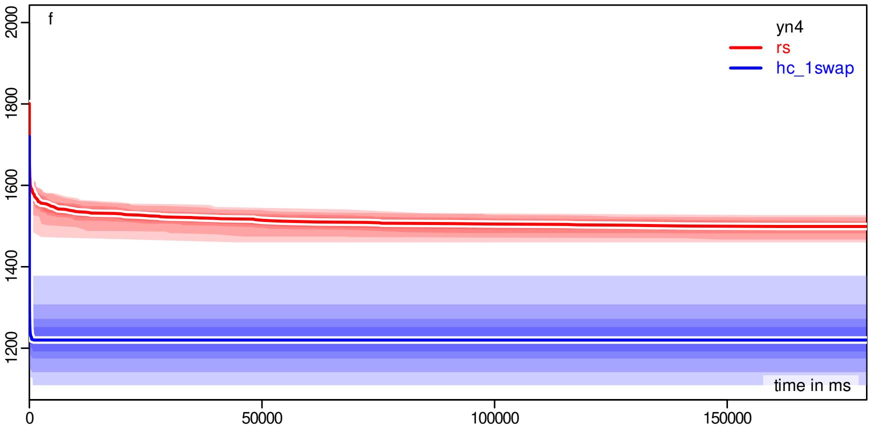
What if we look at this without log-scaling the time axis?



Then it looks even much worse!

But we waste time...

What if we look at this without log-scaling the time axis?



Then it looks even much worse!

Indeed, we waste time!

		makespan				last improvement	
$\mathcal{I}$	algo	best	mean	med	sd	med(t)	med(FEs)
abz7	rs	895	947	949	**12**	85s	6'512'505
	hc_1swap	**717**	**800**	**798**	28	0s	16'978
la24	rs	1153	1206	1208	**15**	82s	15'902'911
	hc_1swap	**999**	**1095**	**1086**	56	0s	6'612
swv15	rs	4988	5166	5172	**50**	87s	5'559'124
	hc_1swap	**3837**	**4108**	**4108**	137	1s	104'598
yn4	rs	1460	1498	1499	**15**	76s	4'814'914
	hc_1swap	**1109**	**1222**	**1220**	48	0s	31'789

Indeed, we waste time!

		makespan				last improvement	
$\mathcal{I}$	algo	best	mean	med	sd	med(t)	med(FEs)
abz7	rs	895	947	949	**12**	85s	6'512'505
	hc_1swap	**717**	**800**	**798**	28	0s	16'978
la24	rs	1153	1206	1208	**15**	82s	15'902'911
	hc_1swap	**999**	**1095**	**1086**	56	0s	6'612
swv15	rs	4988	5166	5172	**50**	87s	5'559'124
	hc_1swap	**3837**	**4108**	**4108**	137	1s	104'598
yn4	rs	1460	1498	1499	**15**	76s	4'814'914
	hc_1swap	**1109**	**1222**	**1220**	48	0s	31'789

Indeed, we waste time!

		makespan				last improvement	
$\mathcal{I}$	algo	best	mean	med	sd	med(t)	med(FEs)
abz7	rs	895	947	949	**12**	85s	6'512'505
	hc_1swap	**717**	**800**	**798**	28	0s	16'978
la24	rs	1153	1206	1208	**15**	82s	15'902'911
	hc_1swap	**999**	**1095**	**1086**	56	0s	6'612
swv15	rs	4988	5166	5172	**50**	87s	5'559'124
	hc_1swap	**3837**	**4108**	**4108**	137	1s	104'598
yn4	rs	1460	1498	1499	**15**	76s	4'814'914
	hc_1swap	**1109**	**1222**	**1220**	48	0s	31'789

- We have three minutes but after about 1 second, our `hc_1swap` algorithm stops improving!

Premature Convergence

- Our algorithm makes most of its progress early during the search.

Premature Convergence

- Our algorithm makes most of its progress early during the search.
- Later, it basically stagnates and cannot improve.

Premature Convergence

- Our algorithm makes most of its progress early during the search.
- Later, it basically stagnates and cannot improve.
- Why is that?

Premature Convergence

- Our algorithm makes most of its progress early during the search.
- Later, it basically stagnates and cannot improve.
- Why is that?
- The search operator `1swap` defines a neighborhood $N(x) \subset \mathbb{X}$ around a point x.

Premature Convergence

- Our algorithm makes most of its progress early during the search.
- Later, it basically stagnates and cannot improve.
- Why is that?
- The search operator `1swap` defines a neighborhood $N(x) \subset \mathbb{X}$ around a point x.
- The hill climber can only find solutions which are in the neighborhood of the current best solution.

Premature Convergence

- Our algorithm makes most of its progress early during the search.
- Later, it basically stagnates and cannot improve.
- Why is that?
- The search operator `1swap` defines a neighborhood $N(x) \subset \mathbb{X}$ around a point x.
- The hill climber can only find solutions which are in the neighborhood of the current best solution.
- Only the schedules that I can reach by swapping two operations of two different jobs.

Premature Convergence

- Our algorithm makes most of its progress early during the search.
- Later, it basically stagnates and cannot improve.
- Why is that?
- The search operator `1swap` defines a neighborhood $N(x) \subset \mathbb{X}$ around a point x.
- The hill climber can only find solutions which are in the neighborhood of the current best solution.
- Only the schedules that I can reach by swapping two operations of two different jobs.
- Clearly $|N(x)| \lll |\mathbb{X}|$!

Premature Convergence

- Our algorithm makes most of its progress early during the search.
- Later, it basically stagnates and cannot improve.
- Why is that?
- The search operator `1swap` defines a neighborhood $N(x) \subset \mathbb{X}$ around a point x.
- The hill climber can only find solutions which are in the neighborhood of the current best solution.
- Only the schedules that I can reach by swapping two operations of two different jobs.
- Clearly $|N(x)| \lll |\mathbb{X}|$!
- What happens if $f(\gamma(x^{\times})) \leq f(\gamma(x)) \forall x \in N(x^{\times})$ but $x^{\times}$ is not the global optimum?

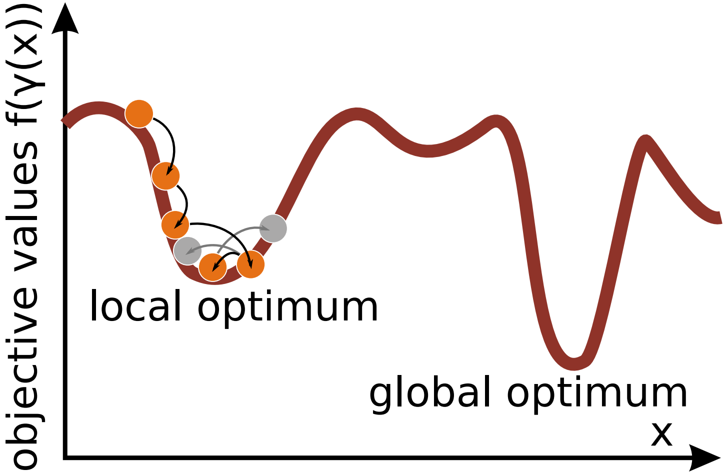
Premature Convergence

- Our algorithm makes most of its progress early during the search.
- Later, it basically stagnates and cannot improve.
- Why is that?
- The search operator `1swap` defines a neighborhood $N(x) \subset \mathbb{X}$ around a point x.
- The hill climber can only find solutions which are in the neighborhood of the current best solution.
- Only the schedules that I can reach by swapping two operations of two different jobs.
- Clearly $|N(x)| \lll |\mathbb{X}|$!
- What happens if $f(\gamma(x^\times)) \leq f(\gamma(x)) \forall x \in N(x^\times)$ but $x^\times$ is not the global optimum?
- Our algorithm gets trapped in the local optimum $x^\times$ and cannot escape!

Premature Convergence

- Our algorithm makes most of its progress early during the search.
- Later, it basically stagnates and cannot improve.
- Why is that?
- The search operator `1swap` defines a neighborhood $N(x) \subset \mathbb{X}$ around a point x.
- The hill climber can only find solutions which are in the neighborhood of the current best solution.
- Only the schedules that I can reach by swapping two operations of two different jobs.
- Clearly $|N(x)| \lll |\mathbb{X}|$!
- What happens if $f(\gamma(x^{\times})) \leq f(\gamma(x)) \forall x \in N(x^{\times})$ but $x^{\times}$ is not the global optimum?
- Our algorithm gets trapped in the local optimum $x^{\times}$ and cannot escape!
- This is called Premature Convergence.[8] [9]

Premature Convergence



Improved Algorithm Concept 1

Stochastic Hill Climber with Restarts

- Idea: We have seen that the results of the hill climber exhibit a relatively high standard deviation.

		makespan				last improvement	
$\mathcal{I}$	algo	best	mean	med	sd	med(t)	med(FEs)
abz7	rs	895	947	949	12	85s	6'512'505
	hc_1swap	717	800	798	28	0s	16'978
la24	rs	1153	1206	1208	15	82s	15'902'911
	hc_1swap	999	1095	1086	56	0s	6'612
swv15	rs	4988	5166	5172	50	87s	5'559'124
	hc_1swap	3837	4108	4108	137	1s	104'598
yn4	rs	1460	1498	1499	15	76s	4'814'914
	hc_1swap	1109	1222	1220	48	0s	31'789

Stochastic Hill Climber with Restarts

- Idea: We have seen that the results of the hill climber exhibit a relatively high standard deviation.
- At the same time, a single *run* of the algorithm converges quickly.

		makespan				last improvement	
$\mathcal{I}$	algo	best	mean	med	sd	med(t)	med(FEs)
abz7	rs	895	947	949	12	85s	6'512'505
	hc_1swap	717	800	798	28	0s	16'978
la24	rs	1153	1206	1208	15	82s	15'902'911
	hc_1swap	999	1095	1086	56	0s	6'612
swv15	rs	4988	5166	5172	50	87s	5'559'124
	hc_1swap	3837	4108	4108	137	1s	104'598
yn4	rs	1460	1498	1499	15	76s	4'814'914
	hc_1swap	1109	1222	1220	48	0s	31'789

Stochastic Hill Climber with Restarts

- Idea: We have seen that the results of the hill climber exhibit a relatively high standard deviation.
- At the same time, a single *run* of the algorithm converges quickly.
- Let us exploit this variance!

Stochastic Hill Climber with Restarts

- Idea: We have seen that the results of the hill climber exhibit a relatively high standard deviation.
- At the same time, a single *run* of the algorithm converges quickly.
- Let us exploit this variance!
- Idea: If we did not make any progress for a number L of algorithm steps, we simply restart at a new random solution.

Stochastic Hill Climber with Restarts

- Idea: We have seen that the results of the hill climber exhibit a relatively high standard deviation.
- At the same time, a single *run* of the algorithm converges quickly.
- Let us exploit this variance!
- Idea: If we did not make any progress for a number L of algorithm steps, we simply restart at a new random solution.
- Of course, we will always remember the overall best solution we ever had (in another variable).

Stochastic Hill Climbing Algorithm with Restarts

```
package aitoa.algorithms;

public class HillClimberWithRestarts<X, Y> {
// unnecessary stuff omitted here...
//
//
//
//

//
//
//
//

//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
}
```

Stochastic Hill Climbing Algorithm with Restarts

```
package aitoa.algorithms;

public class HillClimberWithRestarts<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
//
//
//

//
//
//
//

//
//
//
//
//
//
//
//
//
//
//
//
//
//
  }
}
```

Stochastic Hill Climbing Algorithm with Restarts

```
package aitoa.algorithms;

public class HillClimberWithRestarts<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
    X xCur  = process.getSearchSpace().create();
//
//

//
//
//
//

//
//
//
//
//
//
//
//
//
//
//
//
//
//
  }
}
```

Stochastic Hill Climbing Algorithm with Restarts

```
package aitoa.algorithms;

public class HillClimberWithRestarts<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
    X xCur  = process.getSearchSpace().create();
    X xBest = process.getSearchSpace().create();
//

//
//
//
//

//
//
//
//
//
//
//
//
//
//
//
//
//
//
  }
}
```

Stochastic Hill Climbing Algorithm with Restarts

```
package aitoa.algorithms;

public class HillClimberWithRestarts<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
    X xCur  = process.getSearchSpace().create();
    X xBest = process.getSearchSpace().create();
    Random random = process.getRandom();

//
//
//
//

//
//
//
//
//
//
//
//
//
//
//
//
//
//
  }
}
```

Stochastic Hill Climbing Algorithm with Restarts

```java
package aitoa.algorithms;

public class HillClimberWithRestarts<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
    X xCur  = process.getSearchSpace().create();
    X xBest = process.getSearchSpace().create();
    Random random = process.getRandom();

//
    this.nullary.apply(xBest, random); // sample random solution
//
//

//
//
//
//
//
//
//
//
//
//
//
//
//
//
  }
}
```

Stochastic Hill Climbing Algorithm with Restarts

```
package aitoa.algorithms;

public class HillClimberWithRestarts<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
    X xCur  = process.getSearchSpace().create();
    X xBest = process.getSearchSpace().create();
    Random random = process.getRandom();

//
    this.nullary.apply(xBest, random); // sample random solution
    double fBest       = process.evaluate(xBest); // evaluate it
//

//
//
//
//
//
//
//
//
//
//
//
//
//
//
  } // process has stored best-so-far result
}
```

Stochastic Hill Climbing Algorithm with Restarts

```
package aitoa.algorithms;

public class HillClimberWithRestarts<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
    X xCur  = process.getSearchSpace().create();
    X xBest = process.getSearchSpace().create();
    Random random = process.getRandom();

//
    this.nullary.apply(xBest, random); // sample random solution
    double fBest       = process.evaluate(xBest); // evaluate it
//

//
    this.unary.apply(xBest, xCur, random); // try to improve
//
//
//
//
//
//
//
//
//
//
//
//
  } // process has stored best-so-far result
}
```

Stochastic Hill Climbing Algorithm with Restarts

```
package aitoa.algorithms;

public class HillClimberWithRestarts<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
    X xCur  = process.getSearchSpace().create();
    X xBest = process.getSearchSpace().create();
    Random random = process.getRandom();

//
    this.nullary.apply(xBest, random); // sample random solution
    double fBest       = process.evaluate(xBest); // evaluate it
//

//
    this.unary.apply(xBest, xCur, random); // try to improve
    double fCur = process.evaluate(xCur); // evaluate
//
//
//
//
//
//
//
//
//
//
//
  } // process has stored best-so-far result
}
```

Stochastic Hill Climbing Algorithm with Restarts

```
package aitoa.algorithms;

public class HillClimberWithRestarts<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
    X xCur  = process.getSearchSpace().create();
    X xBest = process.getSearchSpace().create();
    Random random = process.getRandom();

//
    this.nullary.apply(xBest, random); // sample random solution
    double fBest       = process.evaluate(xBest); // evaluate it
//

//
    this.unary.apply(xBest, xCur, random); // try to improve
    double fCur = process.evaluate(xCur); // evaluate
    if (fCur < fBest) { // we found a better solution
//
//
//
    }
//
//
//
//
//
//
  } // process has stored best-so-far result
}
```

Stochastic Hill Climbing Algorithm with Restarts

```
package aitoa.algorithms;

public class HillClimberWithRestarts<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
    X xCur  = process.getSearchSpace().create();
    X xBest = process.getSearchSpace().create();
    Random random = process.getRandom();

//
    this.nullary.apply(xBest, random); // sample random solution
    double fBest       = process.evaluate(xBest); // evaluate it
//

//
    this.unary.apply(xBest, xCur, random); // try to improve
    double fCur = process.evaluate(xCur); // evaluate
    if (fCur < fBest) { // we found a better solution
      fBest = fCur; // remember best quality
//
//
    }
//
//
//
//
//
//
  } // process has stored best-so-far result
}
```

Stochastic Hill Climbing Algorithm with Restarts

```java
package aitoa.algorithms;

public class HillClimberWithRestarts<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
    X xCur  = process.getSearchSpace().create();
    X xBest = process.getSearchSpace().create();
    Random random = process.getRandom();

//
    this.nullary.apply(xBest, random); // sample random solution
    double fBest       = process.evaluate(xBest); // evaluate it
//

//
    this.unary.apply(xBest, xCur, random); // try to improve
    double fCur = process.evaluate(xCur); // evaluate
    if (fCur < fBest) { // we found a better solution
      fBest = fCur; // remember best quality
      process.getSearchSpace().copy(xCur, xBest); // copy
//
    }
//
//
//
//
//
//
  } // process has stored best-so-far result
}
```

Stochastic Hill Climbing Algorithm with Restarts

```
package aitoa.algorithms;

public class HillClimberWithRestarts<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
    X xCur  = process.getSearchSpace().create();
    X xBest = process.getSearchSpace().create();
    Random random = process.getRandom();

//
    this.nullary.apply(xBest, random); // sample random solution
    double fBest       = process.evaluate(xBest); // evaluate it
//

    while (!(process.shouldTerminate())) { // inner loop
      this.unary.apply(xBest, xCur, random); // try to improve
      double fCur = process.evaluate(xCur); // evaluate
      if (fCur < fBest) { // we found a better solution
        fBest = fCur; // remember best quality
        process.getSearchSpace().copy(xCur, xBest); // copy
//
      }
//
//
//
//
    } // inner loop
//
  } // process has stored best-so-far result
}
```

Stochastic Hill Climbing Algorithm with Restarts

```java
package aitoa.algorithms;

public class HillClimberWithRestarts<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
    X xCur  = process.getSearchSpace().create();
    X xBest = process.getSearchSpace().create();
    Random random = process.getRandom();

    while (!(process.shouldTerminate())) { // outer loop: restart
      this.nullary.apply(xBest, random); // sample random solution
      double fBest       = process.evaluate(xBest); // evaluate it
//

      while (!(process.shouldTerminate())) { // inner loop
        this.unary.apply(xBest, xCur, random); // try to improve
        double fCur = process.evaluate(xCur); // evaluate
        if (fCur < fBest) { // we found a better solution
          fBest = fCur; // remember best quality
          process.getSearchSpace().copy(xCur, xBest); // copy
//
        }
//
//
//
//
      } // inner loop
    } // outer loop
  } // process has stored best-so-far result
}
```

Stochastic Hill Climbing Algorithm with Restarts

```
package aitoa.algorithms;

public class HillClimberWithRestarts<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
    X xCur  = process.getSearchSpace().create();
    X xBest = process.getSearchSpace().create();
    Random random = process.getRandom();

    while (!(process.shouldTerminate())) { // outer loop: restart
      this.nullary.apply(xBest, random); // sample random solution
      double fBest       = process.evaluate(xBest); // evaluate it
//

      while (!(process.shouldTerminate())) { // inner loop
        this.unary.apply(xBest, xCur, random); // try to improve
        double fCur = process.evaluate(xCur); // evaluate
        if (fCur < fBest) { // we found a better solution
          fBest = fCur; // remember best quality
          process.getSearchSpace().copy(xCur, xBest); // copy
//
        }
//
//
//
//
      } // inner loop
    } // outer loop
  } // process has stored best-so-far result
}
```

Stochastic Hill Climbing Algorithm with Restarts

```
package aitoa.algorithms;

public class HillClimberWithRestarts<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
    X xCur  = process.getSearchSpace().create();
    X xBest = process.getSearchSpace().create();
    Random random = process.getRandom();

    while (!(process.shouldTerminate())) { // outer loop: restart
      this.nullary.apply(xBest, random); // sample random solution
      double fBest       = process.evaluate(xBest); // evaluate it
      long   failCounter = 0L; // initialize counters

      while (!(process.shouldTerminate())) { // inner loop
        this.unary.apply(xBest, xCur, random); // try to improve
        double fCur = process.evaluate(xCur); // evaluate
        if (fCur < fBest) { // we found a better solution
          fBest = fCur; // remember best quality
          process.getSearchSpace().copy(xCur, xBest); // copy
//
        }
//
//
//
//
      } // inner loop
    } // outer loop
  } // process has stored best-so-far result
}
```

Stochastic Hill Climbing Algorithm with Restarts

```java
package aitoa.algorithms;

public class HillClimberWithRestarts<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
    X xCur  = process.getSearchSpace().create();
    X xBest = process.getSearchSpace().create();
    Random random = process.getRandom();

    while (!(process.shouldTerminate())) { // outer loop: restart
      this.nullary.apply(xBest, random); // sample random solution
      double fBest       = process.evaluate(xBest); // evaluate it
      long   failCounter = 0L; // initialize counters

      while (!(process.shouldTerminate())) { // inner loop
        this.unary.apply(xBest, xCur, random); // try to improve
        double fCur = process.evaluate(xCur); // evaluate
        if (fCur < fBest) { // we found a better solution
          fBest = fCur; // remember best quality
          process.getSearchSpace().copy(xCur, xBest); // copy
          failCounter = 0L; // reset number of unsuccessful steps
        }
//
//
//
//
      } // inner loop
    } // outer loop
  } // process has stored best-so-far result
}
```

Stochastic Hill Climbing Algorithm with Restarts

```java
package aitoa.algorithms;

public class HillClimberWithRestarts<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
    X xCur  = process.getSearchSpace().create();
    X xBest = process.getSearchSpace().create();
    Random random = process.getRandom();

    while (!(process.shouldTerminate())) { // outer loop: restart
      this.nullary.apply(xBest, random); // sample random solution
      double fBest       = process.evaluate(xBest); // evaluate it
      long   failCounter = 0L; // initialize counters

      while (!(process.shouldTerminate())) { // inner loop
        this.unary.apply(xBest, xCur, random); // try to improve
        double fCur = process.evaluate(xCur); // evaluate
        if (fCur < fBest) { // we found a better solution
          fBest = fCur; // remember best quality
          process.getSearchSpace().copy(xCur, xBest); // copy
          failCounter = 0L; // reset number of unsuccessful steps
        } else { // ok, we did not find an improvement
//
//
//
        } // failure
      } // inner loop
    } // outer loop
  } // process has stored best-so-far result
}
```

Stochastic Hill Climbing Algorithm with Restarts

```java
package aitoa.algorithms;

public class HillClimberWithRestarts<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
    X xCur  = process.getSearchSpace().create();
    X xBest = process.getSearchSpace().create();
    Random random = process.getRandom();

    while (!(process.shouldTerminate())) { // outer loop: restart
      this.nullary.apply(xBest, random); // sample random solution
      double fBest       = process.evaluate(xBest); // evaluate it
      long   failCounter = 0L; // initialize counters

      while (!(process.shouldTerminate())) { // inner loop
        this.unary.apply(xBest, xCur, random); // try to improve
        double fCur = process.evaluate(xCur); // evaluate
        if (fCur < fBest) { // we found a better solution
          fBest = fCur; // remember best quality
          process.getSearchSpace().copy(xCur, xBest); // copy
          failCounter = 0L; // reset number of unsuccessful steps
        } else { // ok, we did not find an improvement
          if ((++failCounter) >= this.failsBeforeRestart) {
//
          } // increase fail counter
        } // failure
      } // inner loop
    } // outer loop
  } // process has stored best-so-far result
}
```

Stochastic Hill Climbing Algorithm with Restarts

```
package aitoa.algorithms;

public class HillClimberWithRestarts<X, Y> extends Metaheuristic1<X, Y> {
// unnecessary stuff omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
    X xCur  = process.getSearchSpace().create();
    X xBest = process.getSearchSpace().create();
    Random random = process.getRandom();

    while (!(process.shouldTerminate())) { // outer loop: restart
      this.nullary.apply(xBest, random); // sample random solution
      double fBest       = process.evaluate(xBest); // evaluate it
      long   failCounter = 0L; // initialize counters

      while (!(process.shouldTerminate())) { // inner loop
        this.unary.apply(xBest, xCur, random); // try to improve
        double fCur = process.evaluate(xCur); // evaluate
        if (fCur < fBest) { // we found a better solution
          fBest = fCur; // remember best quality
          process.getSearchSpace().copy(xCur, xBest); // copy
          failCounter = 0L; // reset number of unsuccessful steps
        } else { // ok, we did not find an improvement
          if ((++failCounter) >= this.failsBeforeRestart) {
            break; // jump back to outer loop for restart
          } // increase fail counter
        } // failure
      } // inner loop
    } // outer loop
  } // process has stored best-so-far result
}
```

Experiment and Analysis

Configuring the Algorithm: Parameter L

- We now have an algorithm which, in theory, should be able to utilize some of the variance that we observe in the results of `hc_1swap`.

Configuring the Algorithm: Parameter L

- We now have an algorithm which, in theory, should be able to utilize some of the variance that we observe in the results of `hc_1swap`.
- We got one problem, though . . .

Configuring the Algorithm: Parameter L

- We now have an algorithm which, in theory, should be able to utilize some of the variance that we observe in the results of `hc_1swap`.
- We got one problem, though ... ...actually, it is not just one algorithm, it is an algorithm with a parameter L: `hcr_L_1swap`.

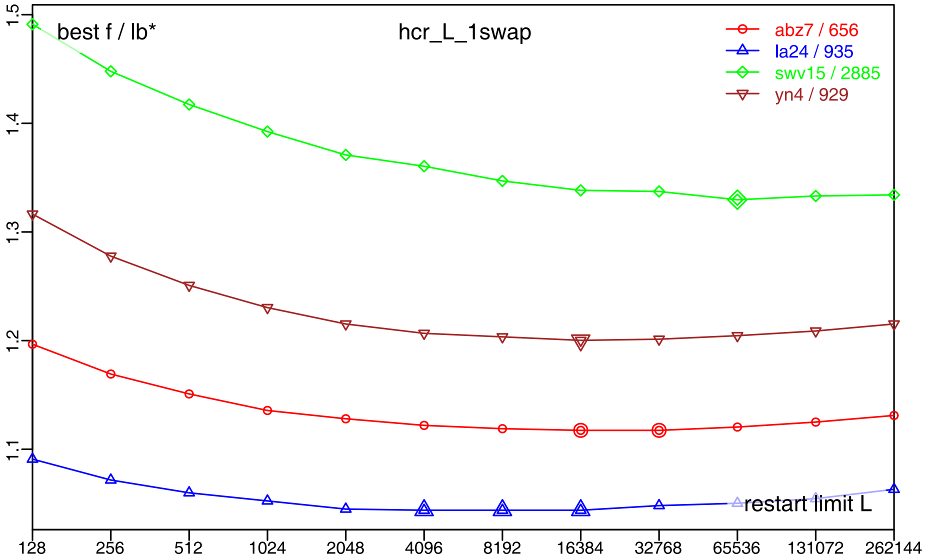
Configuring the Algorithm: Parameter L

- We now have an algorithm which, in theory, should be able to utilize some of the variance that we observe in the results of `hc_1swap`.
- We got one problem, though ... ...actually, it is not just one algorithm, it is an algorithm with a parameter L: `hcr_L_1swap`.
- What do we do with that?

Configuring the Algorithm: Parameter L

- We now have an algorithm which, in theory, should be able to utilize some of the variance that we observe in the results of `hc_1swap`.
- We got one problem, though ... ...actually, it is not just one algorithm, it is an algorithm with a parameter L: `hcr_L_1swap`.
- What do we do with that?
- Let's take a look.

Configuring the Algorithm: Parameter L



Configuring the Algorithm: Parameter L

- We now have an algorithm which, in theory, should be able to utilize some of the variance that we observe in the results of `hc_1swap`.
- We got one problem, though ... ...actually, it is not just one algorithm, it is an algorithm with a parameter L: `hcr_L_1swap`.
- What do we do with that?
- Let's take a look.
- If we choose L too small, we will restart the algorithm too early

Configuring the Algorithm: Parameter L

- We now have an algorithm which, in theory, should be able to utilize some of the variance that we observe in the results of `hc_1swap`.
- We got one problem, though ... ...actually, it is not just one algorithm, it is an algorithm with a parameter L: `hcr_L_1swap`.
- What do we do with that?
- Let's take a look.
- If we choose L too small, we will restart the algorithm too early, before it even arrives in a local optimum

Configuring the Algorithm: Parameter L

- We now have an algorithm which, in theory, should be able to utilize some of the variance that we observe in the results of `hc_1swap`.
- We got one problem, though ... ...actually, it is not just one algorithm, it is an algorithm with a parameter L: `hcr_L_1swap`.
- What do we do with that?
- Let's take a look.
- If we choose L too small, we will restart the algorithm too early, before it even arrives in a local optimum
- If we choose L too large, we will restart too late and thus waste time

Configuring the Algorithm: Parameter L

- We now have an algorithm which, in theory, should be able to utilize some of the variance that we observe in the results of `hc_1swap`.
- We got one problem, though ... ...actually, it is not just one algorithm, it is an algorithm with a parameter L: `hcr_L_1swap`.
- What do we do with that?
- Let's take a look.
- If we choose L too small, we will restart the algorithm too early, before it even arrives in a local optimum
- If we choose L too large, we will restart too late and thus waste time, that we could have used for more restarts

Configuring the Algorithm: Parameter L

- We now have an algorithm which, in theory, should be able to utilize some of the variance that we observe in the results of `hc_1swap`.
- We got one problem, though ... ...actually, it is not just one algorithm, it is an algorithm with a parameter L: `hcr_L_1swap`.
- What do we do with that?
- Let's take a look.
- If we choose L too small, we will restart the algorithm too early, before it even arrives in a local optimum
- If we choose L too large, we will restart too late and thus waste time, that we could have used for more restarts
- $L = 2^{14} = 16'384$ seems to be a reasonable choice.

So what do we get?

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

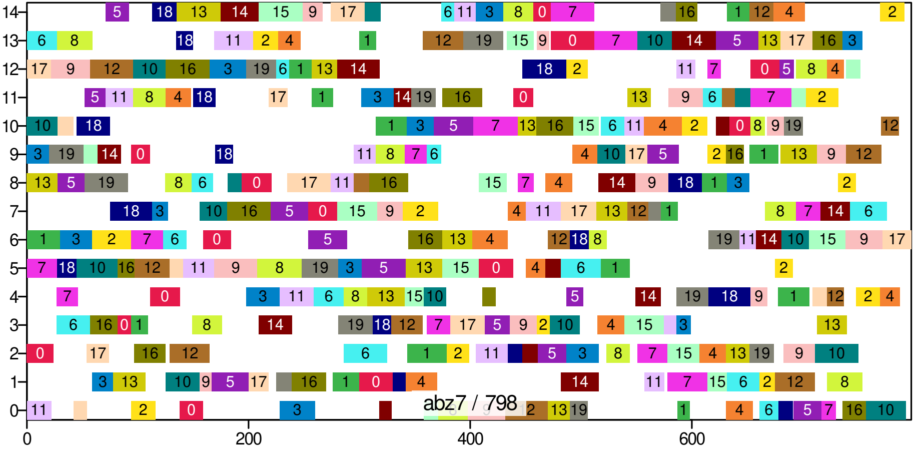
So what do we get?

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

		makespan				last improvement	
$\mathcal{I}$	algo	best	mean	med	sd	med(t)	med(FEs)
abz7	rs	895	947	949	12	85s	6'512'505
	hc_1swap	717	800	798	28	0s	16'978
	hcr_16384_1swap	**714**	**732**	**733**	**6**	91s	18'423'530
la24	rs	1153	1206	1208	15	82s	15'902'911
	hc_1swap	999	1095	1086	56	0s	6'612
	hcr_16384_1swap	**953**	**976**	**976**	**7**	80s	34'437'999
swv15	rs	4988	5166	5172	50	87s	5'559'124
	hc_1swap	3837	4108	4108	137	1s	104'598
	hcr_16384_1swap	**3752**	**3859**	**3861**	**42**	92s	11'756'497
yn4	rs	1460	1498	1499	15	76s	4'814'914
	hc_1swap	1109	1222	1220	48	0s	31'789
	hcr_16384_1swap	**1081**	**1115**	**1115**	**11**	91s	14'804'358

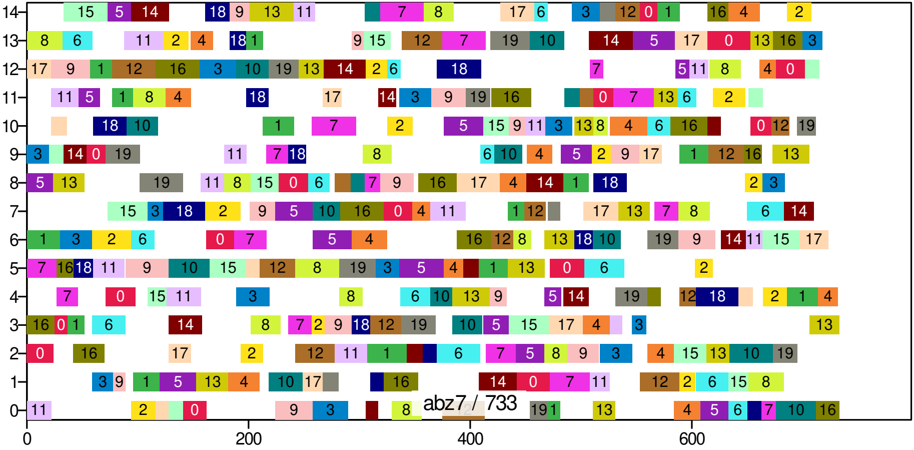
So what do we get?

hc_1swap: median result of 3 min of hill climber



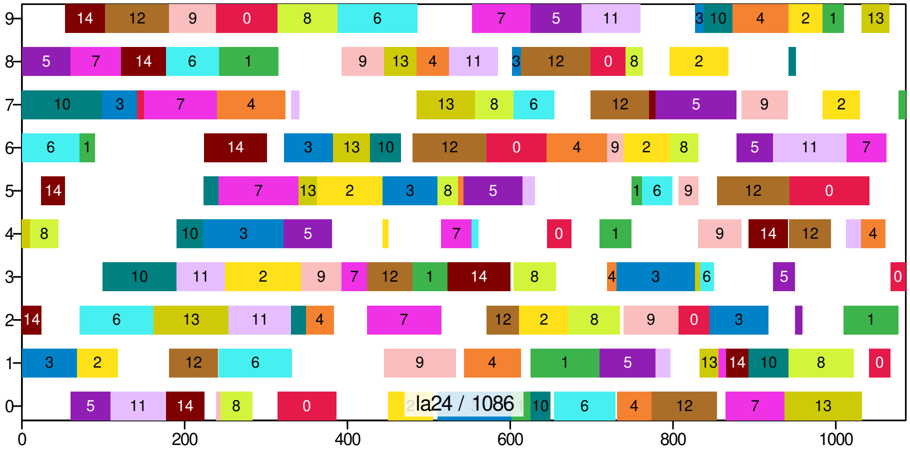
So what do we get?

`hcr_16384_1swap`: median result of 3 min of hill climber which restarts after $L = 16'384$ search steps without improvement



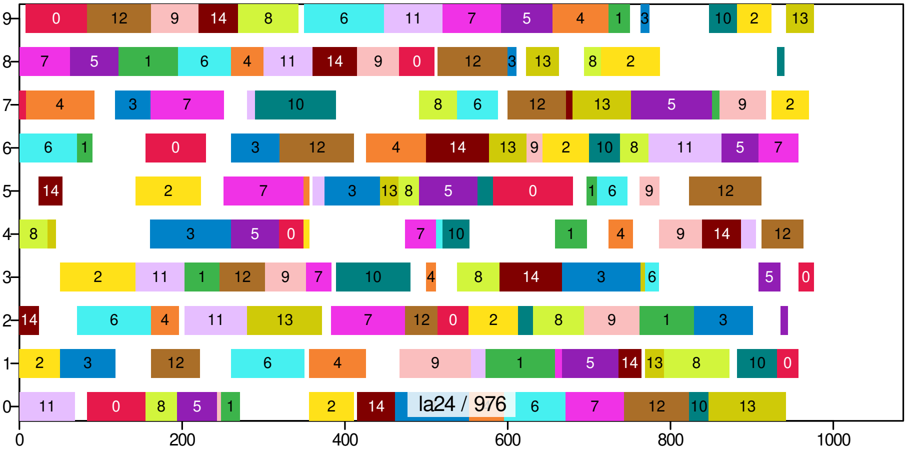
So what do we get?

`hc_1swap`: median result of 3 min of hill climber



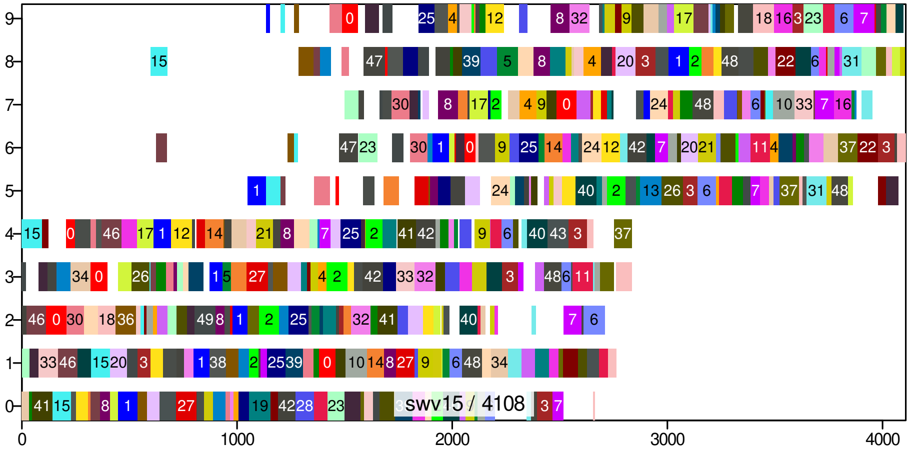
So what do we get?

`hcr_16384_1swap`: median result of 3 min of hill climber which restarts after $L = 16'384$ search steps without improvement



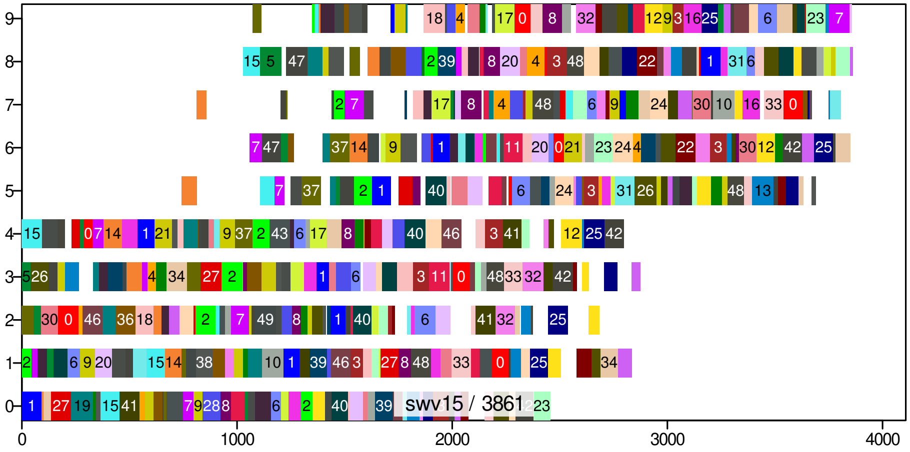
So what do we get?

`hc_1swap`: median result of 3 min of hill climber



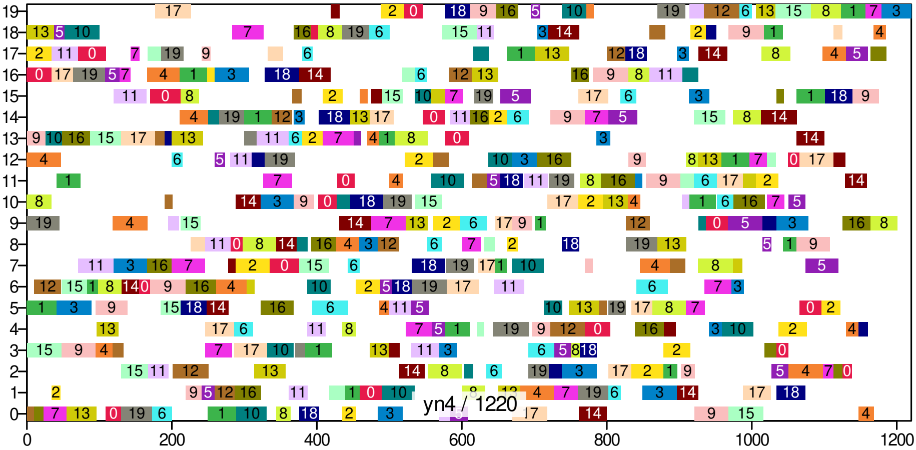
So what do we get?

`hcr_16384_1swap`: median result of 3 min of hill climber which restarts after $L = 16'384$ search steps without improvement



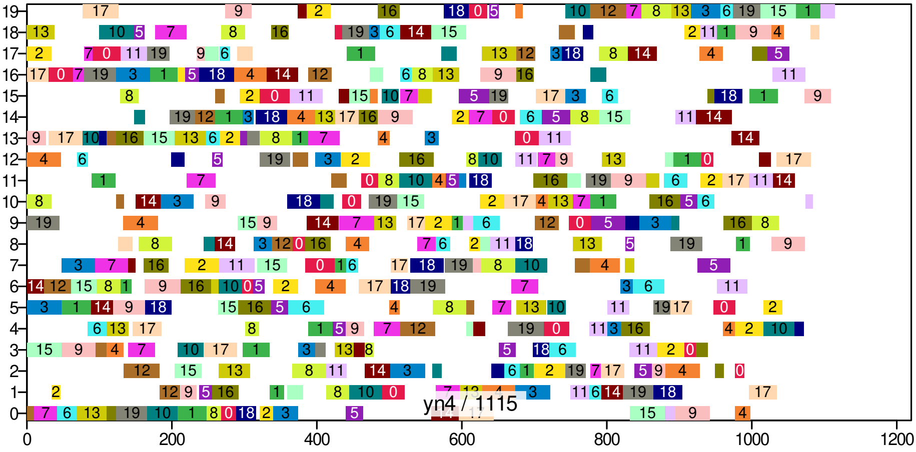
So what do we get?

hc_1swap: median result of 3 min of hill climber



So what do we get?

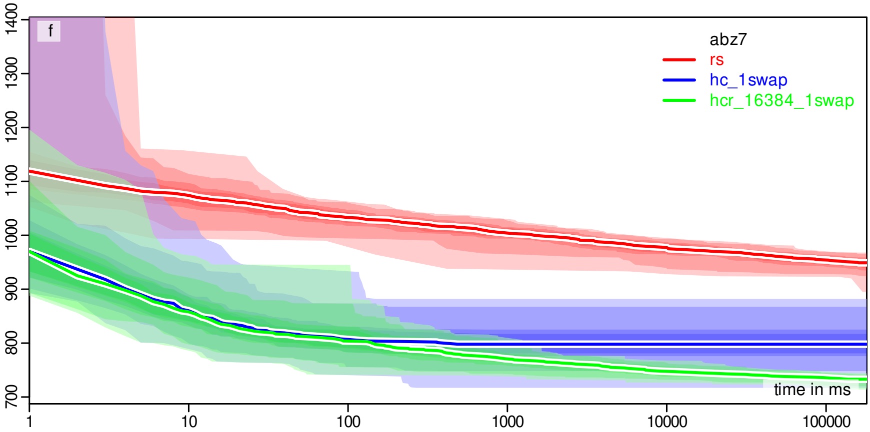
`hcr_16384_1swap`: median result of 3 min of hill climber which restarts after $L = 16'384$ search steps without improvement



Progress over Time

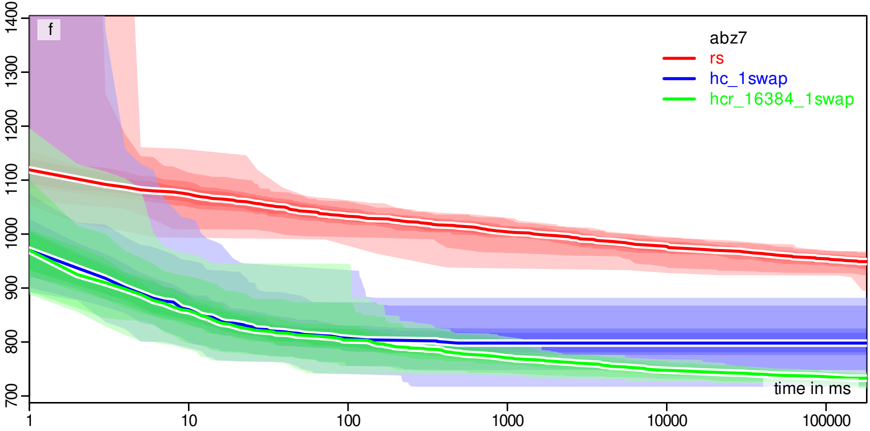
What progress does the algorithm make over time?

Progress over Time



What progress does the algorithm make over time?

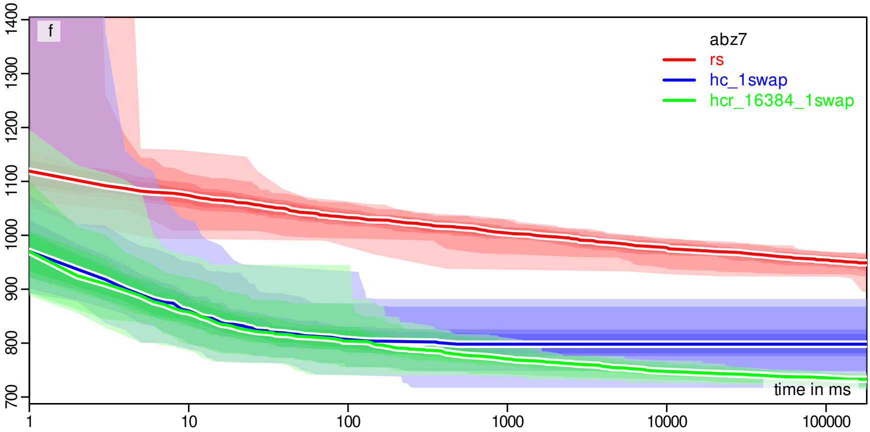
Progress over Time



What progress does the algorithm make over time?

- First it behaves like the normal hill climber

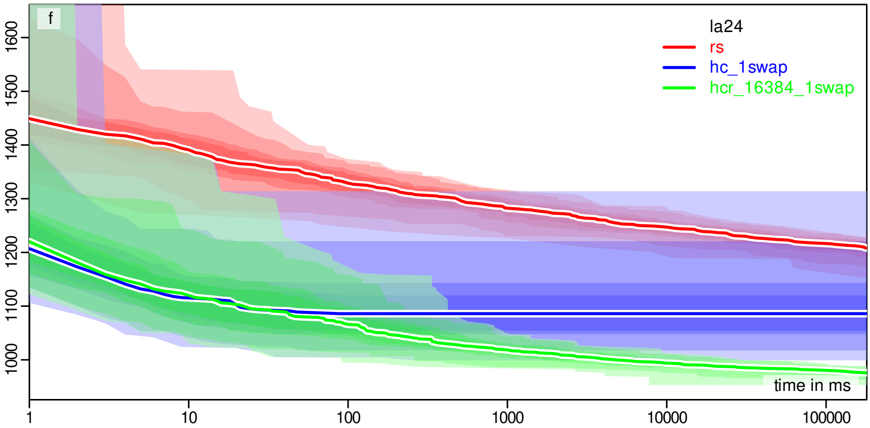
Progress over Time



What progress does the algorithm make over time?

- First it behaves like the normal hill climber
- But it keeps improving.

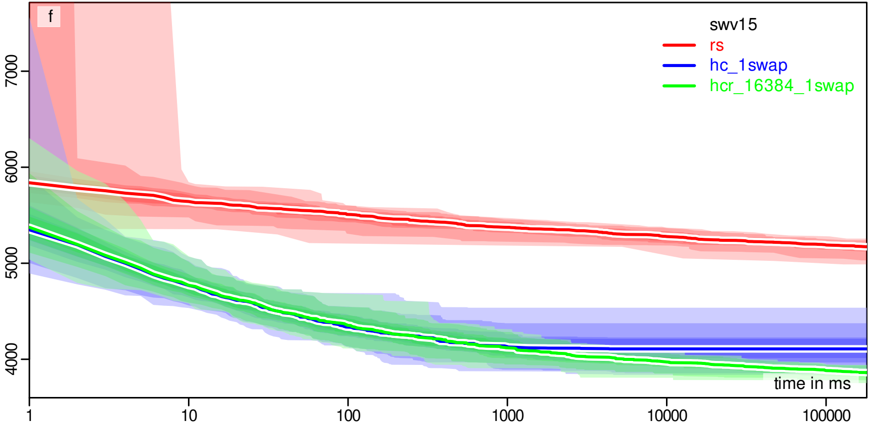
Progress over Time



What progress does the algorithm make over time?

- First it behaves like the normal hill climber
- But it keeps improving.

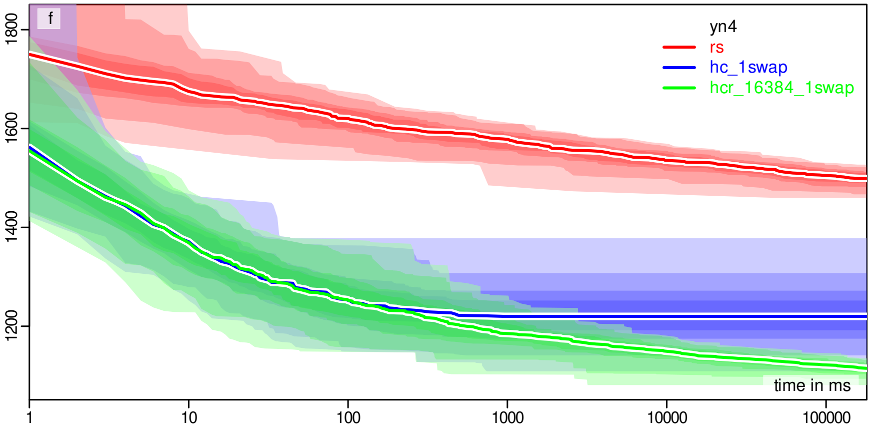
Progress over Time



What progress does the algorithm make over time?

- First it behaves like the normal hill climber
- But it keeps improving.

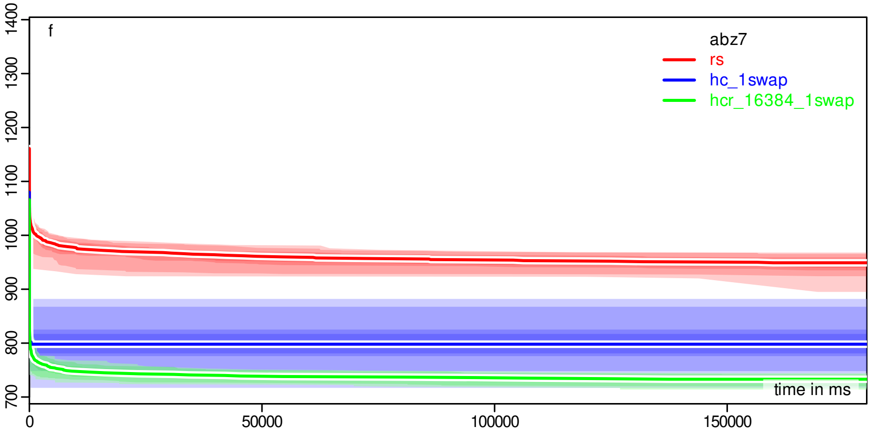
Progress over Time



What progress does the algorithm make over time?

- First it behaves like the normal hill climber
- But it keeps improving.

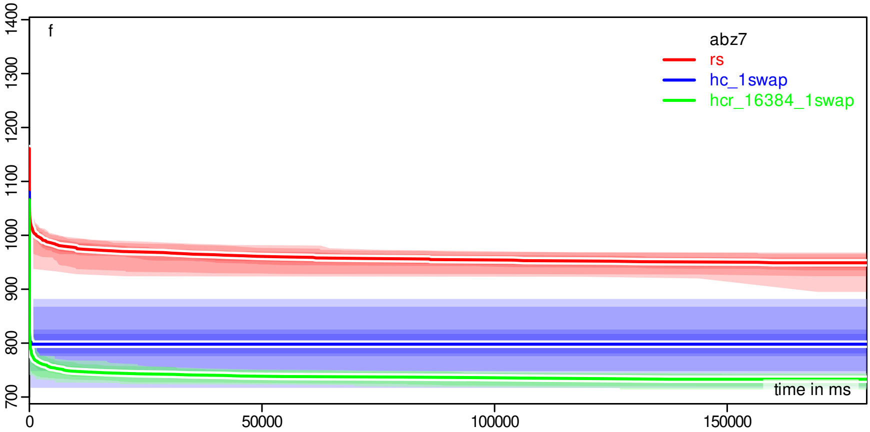
Progress over Time



What progress does the algorithm make over time?

- First it behaves like the normal hill climber
- But it keeps improving.

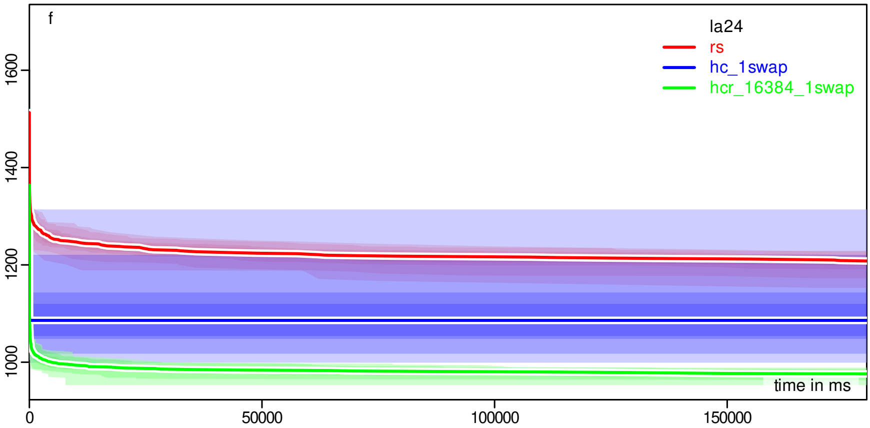
Progress over Time



What progress does the algorithm make over time?

- First it behaves like the normal hill climber
- But it keeps improving.
- Although we still do not use the available time very well...

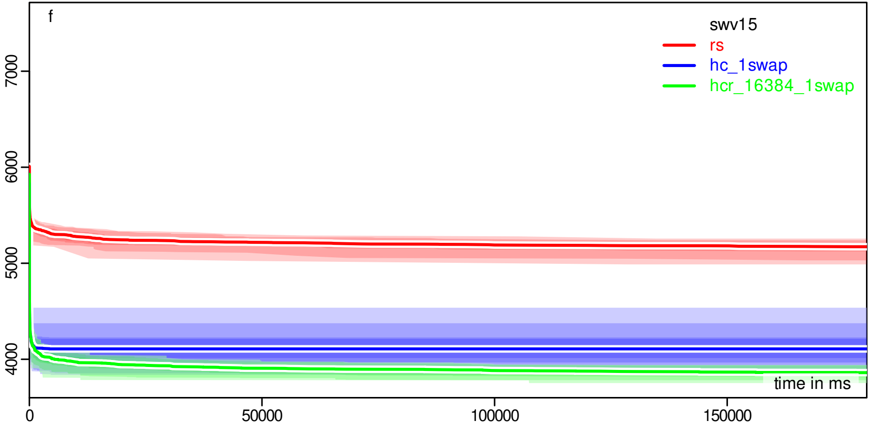
Progress over Time



What progress does the algorithm make over time?

- First it behaves like the normal hill climber
- But it keeps improving.
- Although we still do not use the available time very well. . .

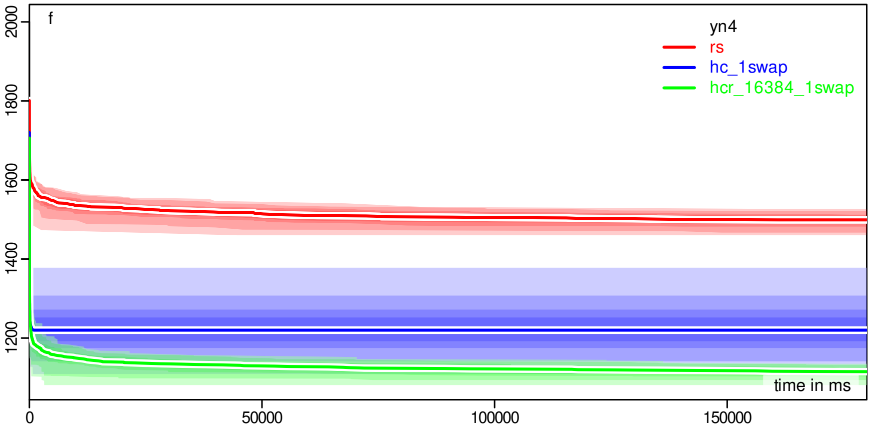
Progress over Time



What progress does the algorithm make over time?

- First it behaves like the normal hill climber
- But it keeps improving.
- Although we still do not use the available time very well. . .

Progress over Time



What progress does the algorithm make over time?

- First it behaves like the normal hill climber
- But it keeps improving.
- Although we still do not use the available time very well. . .

Improved Algorithm Concept 2

Drawbacks of Restarts

- A restarted algorithm is still the same algorithm, just restarted.

Drawbacks of Restarts

- A restarted algorithm is still the same algorithm, just restarted.
- If there are many more local optima than global optima, every restart will probably end again in a local optimum.

Drawbacks of Restarts

- A restarted algorithm is still the same algorithm, just restarted.
- If there are many more local optima than global optima, every restart will probably end again in a local optimum.
- If there are many more "bad" local optima than "good" local optima, every restart will probably end in a "bad" local optimum.

Drawbacks of Restarts

- A restarted algorithm is still the same algorithm, just restarted.
- If there are many more local optima than global optima, every restart will probably end again in a local optimum.
- If there are many more "bad" local optima than "good" local optima, every restart will probably end in a "bad" local optimum.
- While restarts improve the chance to find better solutions, they cannot solve the intrinsic shortcomings of an algorithm.

Drawbacks of Restarts

- A restarted algorithm is still the same algorithm, just restarted.
- If there are many more local optima than global optima, every restart will probably end again in a local optimum.
- If there are many more "bad" local optima than "good" local optima, every restart will probably end in a "bad" local optimum.
- While restarts improve the chance to find better solutions, they cannot solve the intrinsic shortcomings of an algorithm.

- Another problem is: With every restart we throw away all accumulated knowledge and information of the current run.

Drawbacks of Restarts

- A restarted algorithm is still the same algorithm, just restarted.
- If there are many more local optima than global optima, every restart will probably end again in a local optimum.
- If there are many more “bad” local optima than “good” local optima, every restart will probably end in a “bad” local optimum.
- While restarts improve the chance to find better solutions, they cannot solve the intrinsic shortcomings of an algorithm.

- Another problem is: With every restart we throw away all accumulated knowledge and information of the current run.
- Restarts are therefore also wasteful.

How else can we stop premature convergence?

- Our `hc_1swap` hill climber will stop improving if it can no longer finder better solutions.

How else can we stop premature convergence?

- Our `hc_1swap` hill climber will stop improving if it can no longer finder better solutions.
- This happens when it reaches a local optimum.

How else can we stop premature convergence?

- Our `hc_1swap` hill climber will stop improving if it can no longer finder better solutions.
- This happens when it reaches a local optimum.
- A local optimum is a point $x^{\times}$ in $\mathbb{X}$ where no `1swap`-move can yield any improvement.

How else can we stop premature convergence?

- Our `hc_1swap` hill climber will stop improving if it can no longer finder better solutions.
- This happens when it reaches a local optimum.
- A local optimum is a point $x^{\times}$ in $\mathbb{X}$ where no `1swap`-move can yield any improvement.
- It does not matter which two job ids I exchange in the current best string $x^{\times}$, the result is not better than $x^{\times}$.

How else can we stop premature convergence?

- Our `hc_1swap` hill climber will stop improving if it can no longer finder better solutions.
- This happens when it reaches a local optimum.
- A local optimum is a point $x^{\times}$ in $\mathbb{X}$ where no `1swap`-move can yield any improvement.
- It does not matter which two job ids I exchange in the current best string $x^{\times}$, the result is not better than $x^{\times}$.
- Notice: Whether or not a point x is a local optimum, is determined entirely by the unary search operator!

How else can we stop premature convergence?

- Our `hc_1swap` hill climber will stop improving if it can no longer finder better solutions.
- This happens when it reaches a local optimum.
- A local optimum is a point $x^{\times}$ in $\mathbb{X}$ where no `1swap`-move can yield any improvement.
- It does not matter which two job ids I exchange in the current best string $x^{\times}$, the result is not better than $x^{\times}$.
- Notice: Whether or not a point x is a local optimum, is determined entirely by the unary search operator!
- If we had a different operator with a bigger neighborhood, then maybe $x^{\times}$ would no longer be a local optimum and we could still improve the results after reaching it...

Making the neighborhood bigger

- Two solutions x_1 and x_2 are "neighbors" if I can reach x_2 by applying the search operator one time to x_1.

Making the neighborhood bigger

- Two solutions x_1 and x_2 are “neighbors” if I can reach x_2 by applying the search operator one time to x_1.
- The search operator determines which solutions are “neighbors”.

Making the neighborhood bigger

- Two solutions x_1 and x_2 are "neighbors" if I can reach x_2 by applying the search operator one time to x_1.
- The search operator determines which solutions are "neighbors".
- The neighborhood determines what a local optimum is.

Making the neighborhood bigger

- Two solutions x_1 and x_2 are "neighbors" if I can reach x_2 by applying the search operator one time to x_1.
- The search operator determines which solutions are "neighbors".
- The neighborhood determines what a local optimum is.
- Let's make it bigger.

Making the neighborhood bigger

- Two solutions x_1 and x_2 are "neighbors" if I can reach x_2 by applying the search operator one time to x_1.
- The search operator determines which solutions are "neighbors".
- The neighborhood determines what a local optimum is.
- Let's make it bigger.
- It always helps to think about the extreme cases first.

Making the neighborhood bigger

- Two solutions x_1 and x_2 are "neighbors" if I can reach x_2 by applying the search operator one time to x_1.
- The search operator determines which solutions are "neighbors".
- The neighborhood determines what a local optimum is.
- Let's make it bigger.
- It always helps to think about the extreme cases first.
- On one hand, we already have `1swap`, which swaps two jobs.

Making the neighborhood bigger

- Two solutions x_1 and x_2 are “neighbors” if I can reach x_2 by applying the search operator one time to x_1.
- The search operator determines which solutions are “neighbors”.
- The neighborhood determines what a local optimum is.
- Let’s make it bigger.
- It always helps to think about the extreme cases first.
- On one hand, we already have `1swap`, which swaps two jobs. This is the smallest step I can imagine.

Making the neighborhood bigger

- Two solutions x_1 and x_2 are “neighbors” if I can reach x_2 by applying the search operator one time to x_1.
- The search operator determines which solutions are “neighbors”.
- The neighborhood determines what a local optimum is.
- Let’s make it bigger.
- It always helps to think about the extreme cases first.
- On one hand, we already have `1swap`, which swaps two jobs. This is the smallest step I can imagine.
- On the other end of the spectrum, we could simply swap all jobs in our points x randomly.

Making the neighborhood bigger

- Two solutions x_1 and x_2 are "neighbors" if I can reach x_2 by applying the search operator one time to x_1.
- The search operator determines which solutions are "neighbors".
- The neighborhood determines what a local optimum is.
- Let's make it bigger.
- It always helps to think about the extreme cases first.
- On one hand, we already have `1swap`, which swaps two jobs. This is the smallest step I can imagine.
- On the other end of the spectrum, we could simply swap all jobs in our points x randomly. Is this a good idea?

Making the neighborhood bigger

- Two solutions x_1 and x_2 are "neighbors" if I can reach x_2 by applying the search operator one time to x_1.
- The search operator determines which solutions are "neighbors".
- The neighborhood determines what a local optimum is.
- Let's make it bigger.
- It always helps to think about the extreme cases first.
- On one hand, we already have `1swap`, which swaps two jobs. This is the smallest step I can imagine.
- On the other end of the spectrum, we could simply swap all jobs in our points x randomly. Is this a good idea?Probably not: It would turn our algorithm into random sampling!

Making the neighborhood bigger

- Two solutions x_1 and x_2 are "neighbors" if I can reach x_2 by applying the search operator one time to x_1.
- The search operator determines which solutions are "neighbors".
- The neighborhood determines what a local optimum is.
- Let's make it bigger.
- It always helps to think about the extreme cases first.
- On one hand, we already have `1swap`, which swaps two jobs. This is the smallest step I can imagine.
- On the other end of the spectrum, we could simply swap all jobs in our points x randomly. Is this a good idea?Probably not: It would turn our algorithm into random sampling!
- We should respect the causality: small changes to the solution cause small changes in the objective value – big changes will lead to unpredictable results.

Making the neighborhood bigger

- Idea: Let's most often swap 2 jobs

Making the neighborhood bigger

- Idea: Let's most often swap 2 jobs, but sometimes 3

Making the neighborhood bigger

- Idea: Let's most often swap 2 jobs, but sometimes 3, less often 4

Making the neighborhood bigger

- Idea: Let's most often swap 2 jobs, but sometimes 3, less often 4, from time to time 5

Making the neighborhood bigger

- Idea: Let's most often swap 2 jobs, but sometimes 3, less often 4, from time to time 5, rarely 6

Making the neighborhood bigger

- Idea: Let's most often swap 2 jobs, but sometimes 3, less often 4, from time to time 5, rarely 6, hardly ever 7

Making the neighborhood bigger

- Idea: Let's most often swap 2 jobs, but sometimes 3, less often 4, from time to time 5, rarely 6, hardly ever 7, ...

Making the neighborhood bigger

- Idea: Let's most often swap 2 jobs, but sometimes 3, less often 4, from time to time 5, rarely 6, hardly ever 7, ...
- `nswap` operator idea

Making the neighborhood bigger

- Idea: Let's most often swap 2 jobs, but sometimes 3, less often 4, from time to time 5, rarely 6, hardly ever 7, ...
- `nswap` operator idea:
    1. flip a coin: if it is heads (50% probability), we will swap 2 job ids (and stop).

Making the neighborhood bigger

- Idea: Let's most often swap 2 jobs, but sometimes 3, less often 4, from time to time 5, rarely 6, hardly ever 7, ...
- `nswap` operator idea:
  1. flip a coin: if it is heads (50% probability), we will swap 2 job ids (and stop).
  2. otherwise (it was tail), we again flip a coin.

Making the neighborhood bigger

- Idea: Let's most often swap 2 jobs, but sometimes 3, less often 4, from time to time 5, rarely 6, hardly ever 7, ...
- `nswap` operator idea:
    1. flip a coin: if it is heads (50% probability), we will swap 2 job ids (and stop).
    2. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 25% in total), we will swap 3 job ids (and stop).

Making the neighborhood bigger

- Idea: Let's most often swap 2 jobs, but sometimes 3, less often 4, from time to time 5, rarely 6, hardly ever 7, ...
- `nswap` operator idea:
  1. flip a coin: if it is heads (50% probability), we will swap 2 job ids (and stop).
  2. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 25% in total), we will swap 3 job ids (and stop).
  3. otherwise (it was tail), we again flip a coin.

Making the neighborhood bigger

- Idea: Let's most often swap 2 jobs, but sometimes 3, less often 4, from time to time 5, rarely 6, hardly ever 7, ...
- `nswap` operator idea:
    1. flip a coin: if it is heads (50% probability), we will swap 2 job ids (and stop).
    2. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 25% in total), we will swap 3 job ids (and stop).
    3. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 12.5% in total), we will swap 4 job ids (and stop).

Making the neighborhood bigger

- Idea: Let's most often swap 2 jobs, but sometimes 3, less often 4, from time to time 5, rarely 6, hardly ever 7, ...
- `nswap` operator idea:
  1. flip a coin: if it is heads (50% probability), we will swap 2 job ids (and stop).
  2. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 25% in total), we will swap 3 job ids (and stop).
  3. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 12.5% in total), we will swap 4 job ids (and stop).
  4. otherwise (it was tail), we again flip a coin.

Making the neighborhood bigger

- Idea: Let's most often swap 2 jobs, but sometimes 3, less often 4, from time to time 5, rarely 6, hardly ever 7, ...
- `nswap` operator idea:
  1. flip a coin: if it is heads (50% probability), we will swap 2 job ids (and stop).
  2. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 25% in total), we will swap 3 job ids (and stop).
  3. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 12.5% in total), we will swap 4 job ids (and stop).
  4. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 6.25% in total), we will swap 5 job ids (and stop).

Making the neighborhood bigger

- Idea: Let's most often swap 2 jobs, but sometimes 3, less often 4, from time to time 5, rarely 6, hardly ever 7, . . .
- `nswap` operator idea:
  1. flip a coin: if it is heads (50% probability), we will swap 2 job ids (and stop).
  2. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 25% in total), we will swap 3 job ids (and stop).
  3. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 12.5% in total), we will swap 4 job ids (and stop).
  4. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 6.25% in total), we will swap 5 job ids (and stop).
  5. otherwise (it was tail), we again flip a coin.

Making the neighborhood bigger

- Idea: Let's most often swap 2 jobs, but sometimes 3, less often 4, from time to time 5, rarely 6, hardly ever 7, ...
- `nswap` operator idea:
    1. flip a coin: if it is heads (50% probability), we will swap 2 job ids (and stop).
    2. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 25% in total), we will swap 3 job ids (and stop).
    3. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 12.5% in total), we will swap 4 job ids (and stop).
    4. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 6.25% in total), we will swap 5 job ids (and stop).
    5. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 3.125% in total), we will swap 6 job ids (and stop).

Making the neighborhood bigger

- Idea: Let's most often swap 2 jobs, but sometimes 3, less often 4, from time to time 5, rarely 6, hardly ever 7, ...
- `nswap` operator idea:
  1. flip a coin: if it is heads (50% probability), we will swap 2 job ids (and stop).
  2. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 25% in total), we will swap 3 job ids (and stop).
  3. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 12.5% in total), we will swap 4 job ids (and stop).
  4. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 6.25% in total), we will swap 5 job ids (and stop).
  5. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 3.125% in total), we will swap 6 job ids (and stop).
  6. and so on.

Making the neighborhood bigger

- Idea: Let's most often swap 2 jobs, but sometimes 3, less often 4, from time to time 5, rarely 6, hardly ever 7, ...
- `nswap` operator idea:
  1. flip a coin: if it is heads (50% probability), we will swap 2 job ids (and stop).
  2. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 25% in total), we will swap 3 job ids (and stop).
  3. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 12.5% in total), we will swap 4 job ids (and stop).
  4. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 6.25% in total), we will swap 5 job ids (and stop).
  5. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 3.125% in total), we will swap 6 job ids (and stop).
  6. and so on.
- We most often make small moves, but sometimes bigger ones.

Making the neighborhood bigger

- Idea: Let's most often swap 2 jobs, but sometimes 3, less often 4, from time to time 5, rarely 6, hardly ever 7, ...
- `nswap` operator idea:
    1. flip a coin: if it is heads (50% probability), we will swap 2 job ids (and stop).
    2. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 25% in total), we will swap 3 job ids (and stop).
    3. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 12.5% in total), we will swap 4 job ids (and stop).
    4. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 6.25% in total), we will swap 5 job ids (and stop).
    5. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 3.125% in total), we will swap 6 job ids (and stop).
    6. and so on.
- We most often make small moves, but sometimes bigger ones.
- Theoretically, we could always escape from any local optimum.

Making the neighborhood bigger

- Idea: Let's most often swap 2 jobs, but sometimes 3, less often 4, from time to time 5, rarely 6, hardly ever 7, ...
- `nswap` operator idea:
    1. flip a coin: if it is heads (50% probability), we will swap 2 job ids (and stop).
    2. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 25% in total), we will swap 3 job ids (and stop).
    3. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 12.5% in total), we will swap 4 job ids (and stop).
    4. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 6.25% in total), we will swap 5 job ids (and stop).
    5. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 3.125% in total), we will swap 6 job ids (and stop).
    6. and so on.
- We most often make small moves, but sometimes bigger ones.
- Theoretically, we could always escape from any local optimum, but the probability may sometimes be very very small.

Implementation of the `nswap` Operator

```
package aitoa.examples.jssp;

public class JSSPUnaryOperatorNSwap {
// unnecessary stuff omitted here...
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//

//
//
}
```

Implementation of the nswap Operator

```
package aitoa.examples.jssp;

public class JSSPUnaryOperatorNSwap implements IUnarySearchOperator<int[]> {
// unnecessary stuff omitted here...
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//

//
//
}
```

Implementation of the `nswap` Operator

```java
package aitoa.examples.jssp;

public class JSSPUnaryOperatorNSwap implements IUnarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] x, int[] dest, Random random) {
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//

//
  }
}
```

Implementation of the nswap Operator

```java
package aitoa.examples.jssp;

public class JSSPUnaryOperatorNSwap implements IUnarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] x, int[] dest, Random random) {
    System.arraycopy(x, 0, dest, 0, x.length); // copy x to dest
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//

//
  }
}
```

Implementation of the `nswap` Operator

```
package aitoa.examples.jssp;

public class JSSPUnaryOperatorNSwap implements IUnarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] x, int[] dest, Random random) {
    System.arraycopy(x, 0, dest, 0, x.length); // copy x to dest
    int i     = random.nextInt(dest.length);   // index of first job to swap
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//

//
  }
}
```

Implementation of the nswap Operator

```
package aitoa.examples.jssp;

public class JSSPUnaryOperatorNSwap implements IUnarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] x, int[] dest, Random random) {
    System.arraycopy(x, 0, dest, 0, x.length); // copy x to dest
    int i     = random.nextInt(dest.length);   // index of first job to swap
    int first = dest[i];
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//

//
  }
}
```

Implementation of the nswap Operator

```
package aitoa.examples.jssp;

public class JSSPUnaryOperatorNSwap implements IUnarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] x, int[] dest, Random random) {
    System.arraycopy(x, 0, dest, 0, x.length); // copy x to dest
    int i     = random.nextInt(dest.length);   // index of first job to swap
    int first = dest[i];
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//

    dest[i] = first; // write back first id to last copied index
  }
}
```

Implementation of the `nswap` Operator

```
package aitoa.examples.jssp;

public class JSSPUnaryOperatorNSwap implements IUnarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] x, int[] dest, Random random) {
    System.arraycopy(x, 0, dest, 0, x.length); // copy x to dest
    int i     = random.nextInt(dest.length);   // index of first job to swap
    int first = dest[i];
//
//
//
//
//
    int j    = random.nextInt(dest.length);
//
//
//
//
//
//
//
//
//
//

    dest[i] = first; // write back first id to last copied index
  }
}
```

Implementation of the `nswap` Operator

```java
package aitoa.examples.jssp;

public class JSSPUnaryOperatorNSwap implements IUnarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] x, int[] dest, Random random) {
    System.arraycopy(x, 0, dest, 0, x.length); // copy x to dest
    int i     = random.nextInt(dest.length);   // index of first job to swap
    int first = dest[i];
//
//
//
//
//
    int j    = random.nextInt(dest.length);
    int jobJ = dest[j];
//
//
//
//
//
//
//
//
//

    dest[i] = first; // write back first id to last copied index
  }
}
```

Implementation of the nswap Operator

```
package aitoa.examples.jssp;

public class JSSPUnaryOperatorNSwap implements IUnarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] x, int[] dest, Random random) {
    System.arraycopy(x, 0, dest, 0, x.length); // copy x to dest
    int i     = random.nextInt(dest.length);   // index of first job to swap
    int first = dest[i];
//
//
//
//
//
    int j    = random.nextInt(dest.length);
    int jobJ = dest[j];
    if (first != jobJ) {
//
//
//
//
//
    }
//
//

    dest[i] = first; // write back first id to last copied index
  }
}
```

Implementation of the nswap Operator

```
package aitoa.examples.jssp;

public class JSSPUnaryOperatorNSwap implements IUnarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] x, int[] dest, Random random) {
    System.arraycopy(x, 0, dest, 0, x.length); // copy x to dest
    int i     = random.nextInt(dest.length);   // index of first job to swap
    int first = dest[i];
//
//
//
//
//
    int j    = random.nextInt(dest.length);
    int jobJ = dest[j];
    if (first != jobJ) {
//
      dest[i] = jobJ; // overwrite job at index i with jobJ
//
//
//
    }
//
//

    dest[i] = first; // write back first id to last copied index
  }
}
```

Implementation of the nswap Operator

```java
package aitoa.examples.jssp;

public class JSSPUnaryOperatorNSwap implements IUnarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] x, int[] dest, Random random) {
    System.arraycopy(x, 0, dest, 0, x.length); // copy x to dest
    int i     = random.nextInt(dest.length);   // index of first job to swap
    int first = dest[i];
//
//
//
//
//
    int j    = random.nextInt(dest.length);
    int jobJ = dest[j];
    if (first != jobJ) {
//
      dest[i] = jobJ; // overwrite job at index i with jobJ
      i       = j; // remember index j: we will overwrite it next
//
//
    }
//
//

    dest[i] = first; // write back first id to last copied index
  }
}
```

Implementation of the `nswap` Operator

```java
package aitoa.examples.jssp;

public class JSSPUnaryOperatorNSwap implements IUnarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] x, int[] dest, Random random) {
    System.arraycopy(x, 0, dest, 0, x.length); // copy x to dest
    int i     = random.nextInt(dest.length);   // index of first job to swap
    int first = dest[i];
//
//
//
//
    inner: for (;;) { // find a location with a different job
      int j    = random.nextInt(dest.length);
      int jobJ = dest[j];
      if (first != jobJ) {
//
        dest[i] = jobJ; // overwrite job at index i with jobJ
        i       = j; // remember index j: we will overwrite it next
//
        break inner;
      }
    }
//

    dest[i] = first; // write back first id to last copied index
  }
}
```

Implementation of the nswap Operator

```java
package aitoa.examples.jssp;

public class JSSPUnaryOperatorNSwap implements IUnarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] x, int[] dest, Random random) {
    System.arraycopy(x, 0, dest, 0, x.length); // copy x to dest
    int i     = random.nextInt(dest.length);   // index of first job to swap
    int first = dest[i];
//
//
    for(;;) {
//
      inner: for (;;) { // find a location with a different job
        int j    = random.nextInt(dest.length);
        int jobJ = dest[j];
        if (first != jobJ) {
//
          dest[i] = jobJ; // overwrite job at index i with jobJ
          i       = j; // remember index j: we will overwrite it next
//
          break inner;
        }
      }
    }

    dest[i] = first; // write back first id to last copied index
  }
}
```

Implementation of the `nswap` Operator

```java
package aitoa.examples.jssp;

public class JSSPUnaryOperatorNSwap implements IUnarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] x, int[] dest, Random random) {
    System.arraycopy(x, 0, dest, 0, x.length); // copy x to dest
    int i     = random.nextInt(dest.length);   // index of first job to swap
    int first = dest[i];
//
    boolean hasNext;
    do { // we repeat a geometrically distributed number of times
      hasNext = random.nextBoolean();
      inner: for (;;) { // find a location with a different job
        int j    = random.nextInt(dest.length);
        int jobJ = dest[j];
        if (first != jobJ) {
//
          dest[i] = jobJ; // overwrite job at index i with jobJ
          i       = j; // remember index j: we will overwrite it next
//
          break inner;
        }
      }
    } while (hasNext); // Bernoulli process

    dest[i] = first; // write back first id to last copied index
  }
}
```

Implementation of the `nswap` Operator

```
package aitoa.examples.jssp;

public class JSSPUnaryOperatorNSwap implements IUnarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] x, int[] dest, Random random) {
    System.arraycopy(x, 0, dest, 0, x.length); // copy x to dest
    int i     = random.nextInt(dest.length);   // index of first job to swap
    int first = dest[i];
    int last  = first;   // last stores the job id to "swap in"
    boolean hasNext;
    do { // we repeat a geometrically distributed number of times
      hasNext = random.nextBoolean();
      inner: for (;;) { // find a location with a different job
        int j    = random.nextInt(dest.length);
        int jobJ = dest[j];
        if (first != jobJ) {
//
          dest[i] = jobJ; // overwrite job at index i with jobJ
          i       = j; // remember index j: we will overwrite it next
//
          break inner;
        }
      }
    } while (hasNext); // Bernoulli process

    dest[i] = first; // write back first id to last copied index
  }
}
```

Implementation of the `nswap` Operator

```
package aitoa.examples.jssp;

public class JSSPUnaryOperatorNSwap implements IUnarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] x, int[] dest, Random random) {
    System.arraycopy(x, 0, dest, 0, x.length); // copy x to dest
    int i     = random.nextInt(dest.length);   // index of first job to swap
    int first = dest[i];
    int last  = first;   // last stores the job id to "swap in"
    boolean hasNext;
    do { // we repeat a geometrically distributed number of times
      hasNext = random.nextBoolean();
      inner: for (;;) { // find a location with a different job
        int j    = random.nextInt(dest.length);
        int jobJ = dest[j];
        if (first != jobJ) {
//
          dest[i] = jobJ; // overwrite job at index i with jobJ
          i       = j; // remember index j: we will overwrite it next
          last    = jobJ; // but not with the same value jobJ...
          break inner;
        }
      }
    } while (hasNext); // Bernoulli process

    dest[i] = first; // write back first id to last copied index
  }
}
```

Implementation of the `nswap` Operator

```java
package aitoa.examples.jssp;

public class JSSPUnaryOperatorNSwap implements IUnarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] x, int[] dest, Random random) {
    System.arraycopy(x, 0, dest, 0, x.length); // copy x to dest
    int i     = random.nextInt(dest.length);   // index of first job to swap
    int first = dest[i];
    int last  = first;   // last stores the job id to "swap in"
    boolean hasNext;
    do { // we repeat a geometrically distributed number of times
      hasNext = random.nextBoolean();
      inner: for (;;) { // find a location with a different job
        int j    = random.nextInt(dest.length);
        int jobJ = dest[j];
        if ((last != jobJ) && // don't swap job with itself
            (           (first != jobJ))) { // also not at end
          dest[i] = jobJ; // overwrite job at index i with jobJ
          i       = j; // remember index j: we will overwrite it next
          last    = jobJ; // but not with the same value jobJ...
          break inner;
        }
      }
    } while (hasNext); // Bernoulli process

    dest[i] = first; // write back first id to last copied index
  }
}
```

Implementation of the nswap Operator

```java
package aitoa.examples.jssp;

public class JSSPUnaryOperatorNSwap implements IUnarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] x, int[] dest, Random random) {
    System.arraycopy(x, 0, dest, 0, x.length); // copy x to dest
    int i     = random.nextInt(dest.length);   // index of first job to swap
    int first = dest[i];
    int last  = first;   // last stores the job id to "swap in"
    boolean hasNext;
    do { // we repeat a geometrically distributed number of times
      hasNext = random.nextBoolean();
      inner: for (;;) { // find a location with a different job
        int j    = random.nextInt(dest.length);
        int jobJ = dest[j];
        if ((last != jobJ) && // don't swap job with itself
            (hasNext || (first != jobJ))) { // also not at end
          dest[i] = jobJ; // overwrite job at index i with jobJ
          i       = j; // remember index j: we will overwrite it next
          last    = jobJ; // but not with the same value jobJ...
          break inner;
        }
      }
    } while (hasNext); // Bernoulli process

    dest[i] = first; // write back first id to last copied index
  }
}
```

Experiment and Analysis

So what do we get?

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

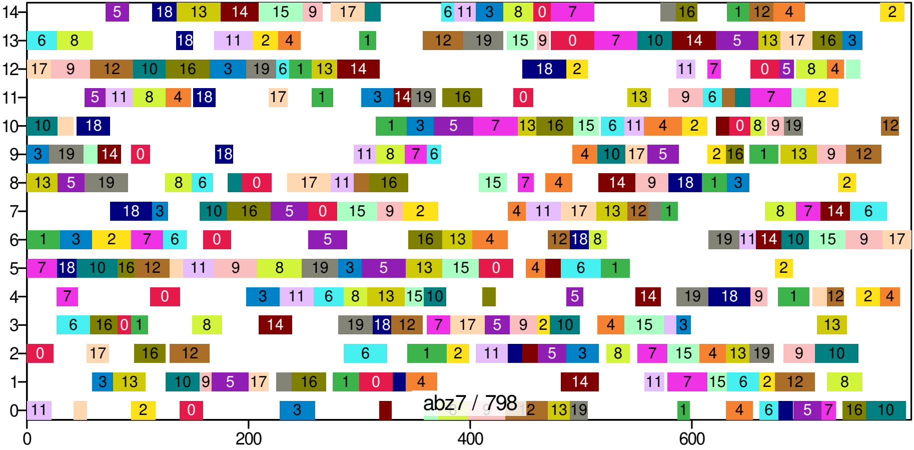
So what do we get?

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

		makespan				last improvement	
$\mathcal{I}$	algo	best	mean	med	sd	med(t)	med(FEs)
abz7	hc_1swap	717	800	798	28	0s	16'978
	hcr_16384_1swap	**714**	**732**	**733**	**6	91s	18'423'530
	hc_nswap	724	758	758	17	35s	7'781'762
la24	hc_1swap	999	1095	1086	56	0s	6'612
	hcr_16384_1swap	953	**976**	**976**	**7**	80s	34'437'999
	hc_nswap	**945**	1018	1016	29	25s	9'072'935
swv15	hc_1swap	3837	4108	4108	137	1s	104'598
	hcr_16384_1swap	3752	**3859**	**3861**	**42**	92s	11'756'497
	hc_nswap	**3602**	3880	3872	112	70s	8'351'112
yn4	hc_1swap	1109	1222	1220	48	0s	31'789
	hcr_16384_1swap	**1081**	**1115**	**1115**	**11**	91s	14'804'358
	hc_nswap	1095	1162	1160	34	71s	11'016'757

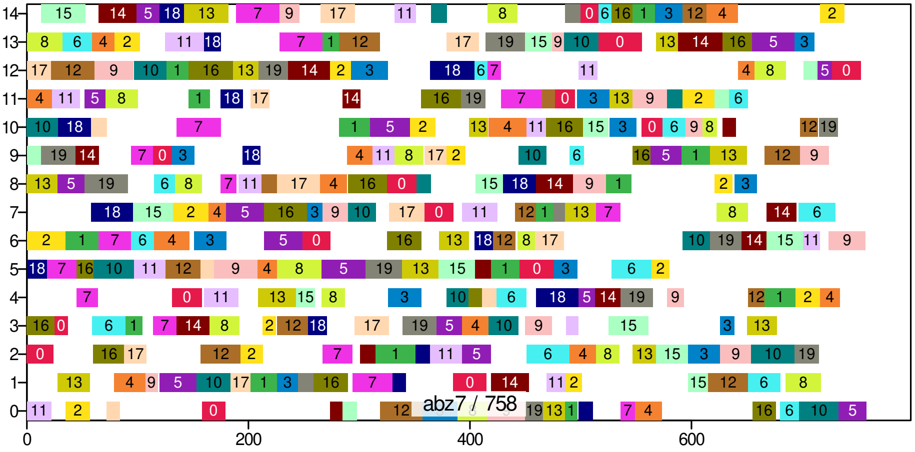
So what do we get?

`hc_1swap`: median result of 3 min of hill climber using `1swap`



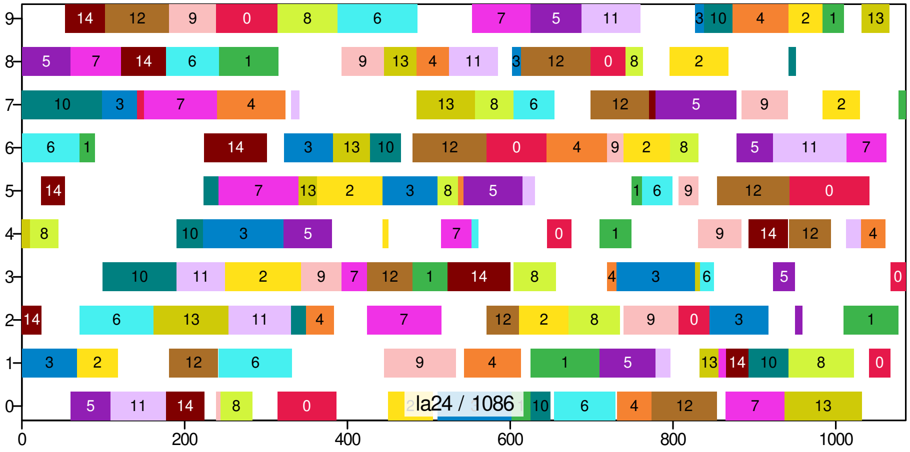
So what do we get?

`hc_nswap`: median result of 3 min of hill climber using `nswap`

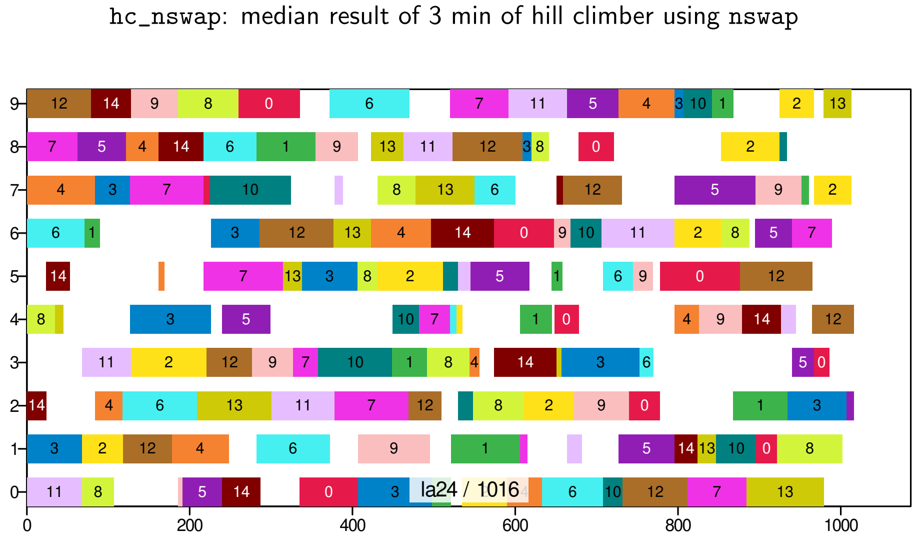


So what do we get?

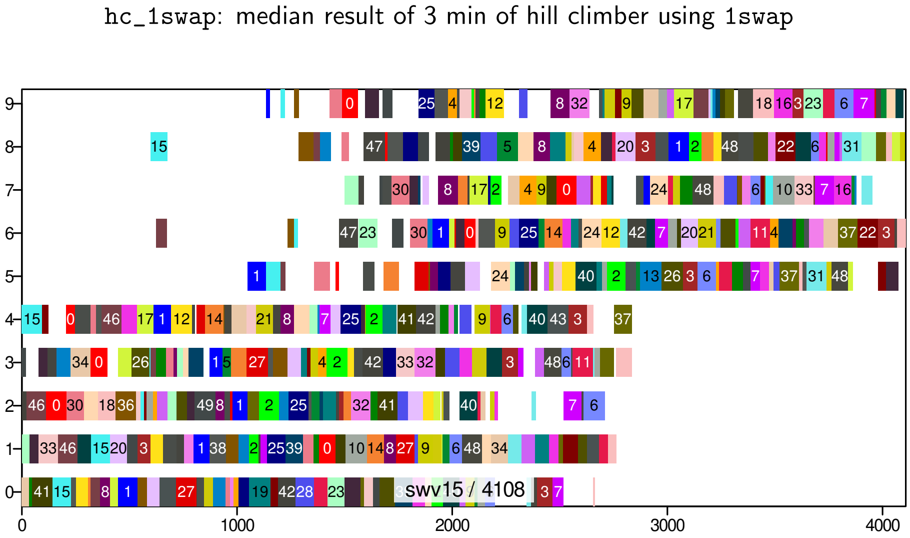
`hc_1swap`: median result of 3 min of hill climber using `1swap`



So what do we get?

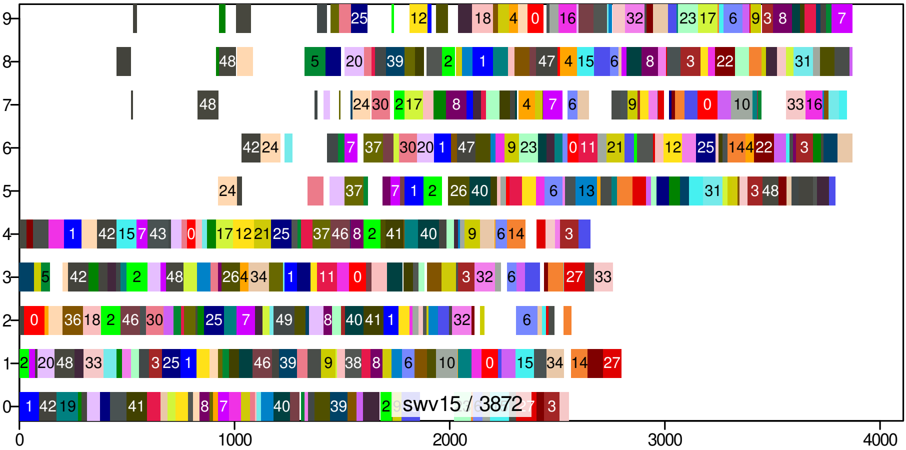


So what do we get?

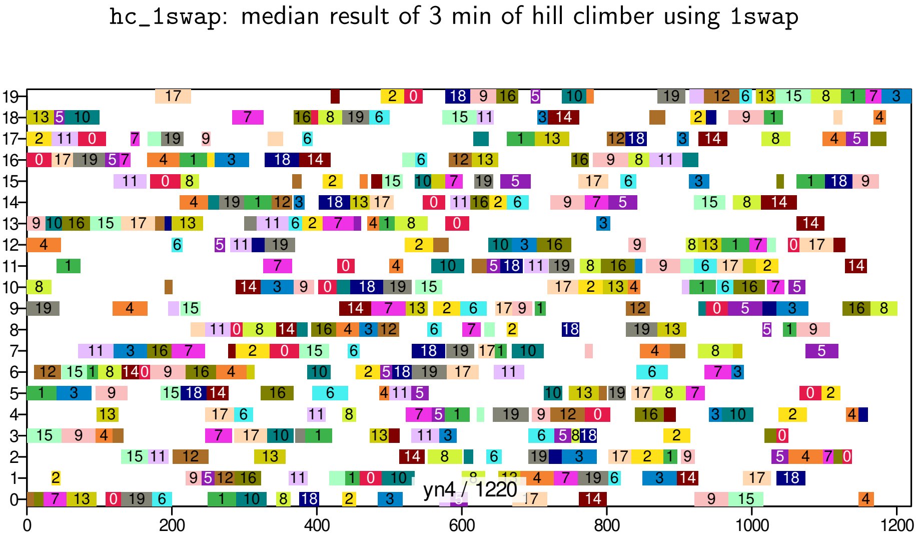


So what do we get?

`hc_nswap`: median result of 3 min of hill climber using `nswap`

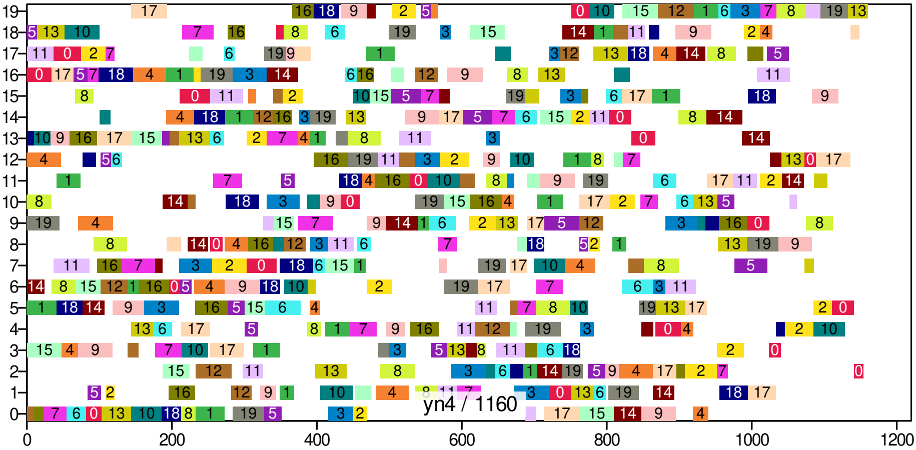


So what do we get?



So what do we get?

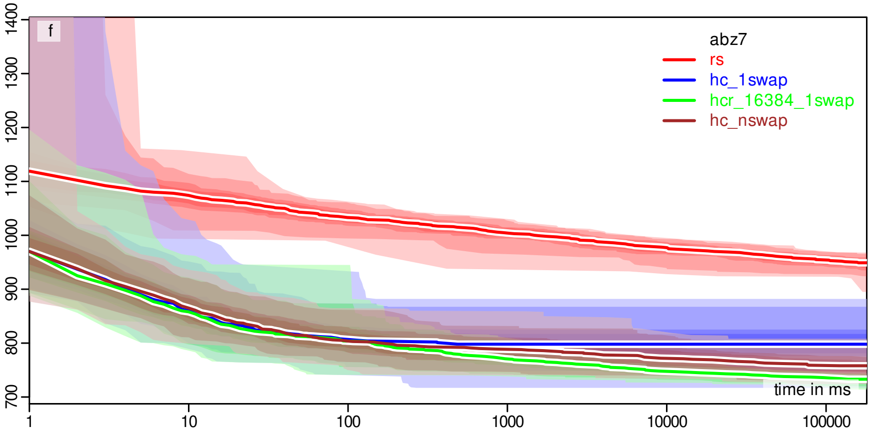
hc_nswap: median result of 3 min of hill climber using nswap



Progress over Time

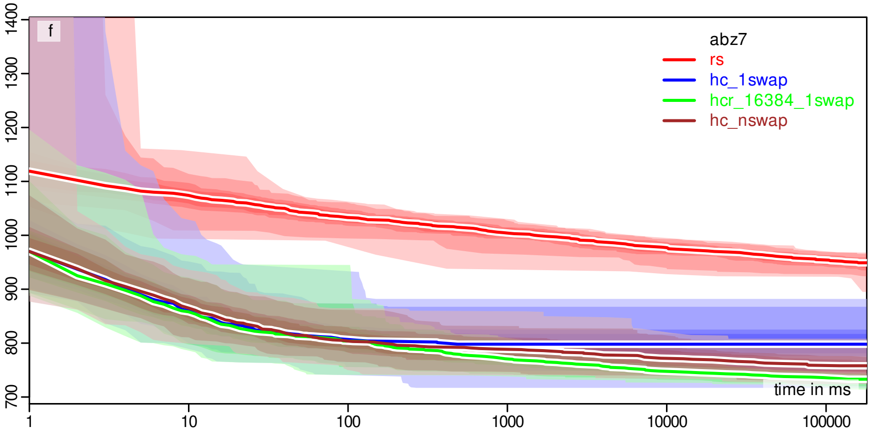
What progress does the algorithm make over time?

Progress over Time



What progress does the algorithm make over time?

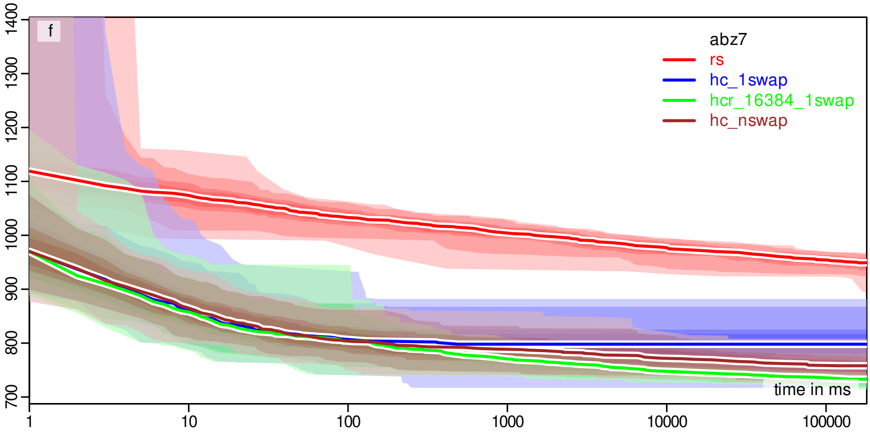
Progress over Time



What progress does the algorithm make over time?

- `hc_nswap` first behaves like `hc_1swap`, because most of the `nswap` moves are the same as `1swap` moves.

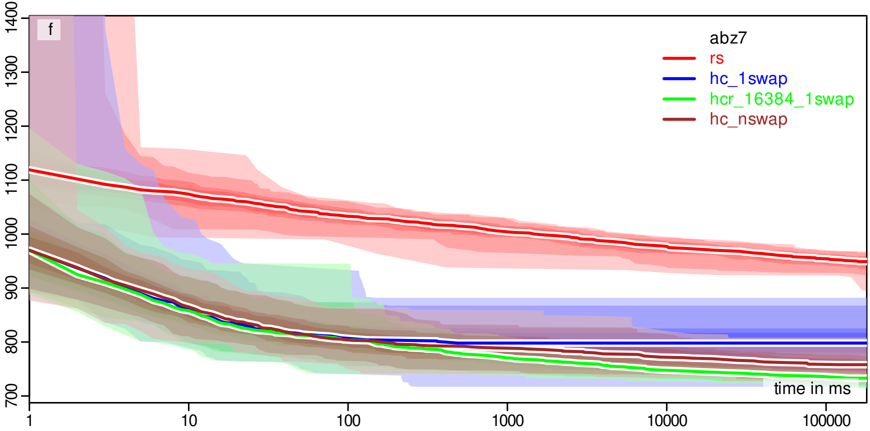
Progress over Time



What progress does the algorithm make over time?

- The rare larger moves allow it to escape from local optima that would trap `hc_1swap`.

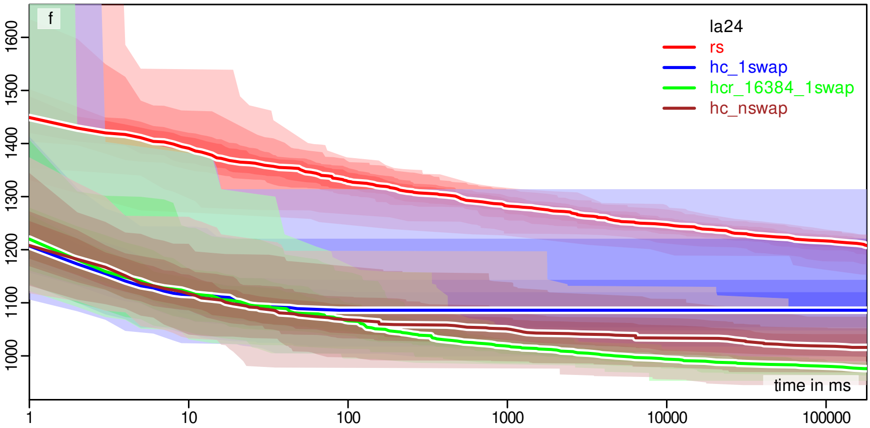
Progress over Time



What progress does the algorithm make over time?

- The rare larger moves allow it to escape from local optima that would trap `hc_1swap`.
- The hill climber with restarts seems to improve longer.

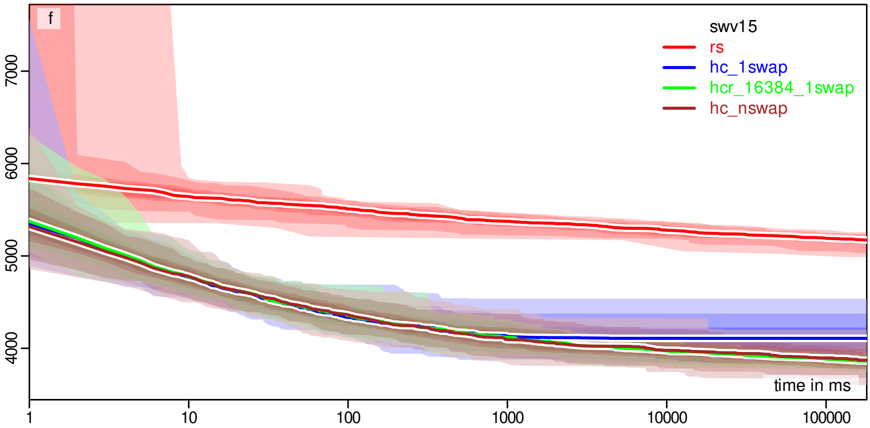
Progress over Time



What progress does the algorithm make over time?

- The rare larger moves allow it to escape from local optima that would trap `hc_1swap`.
- The hill climber with restarts seems to improve longer.

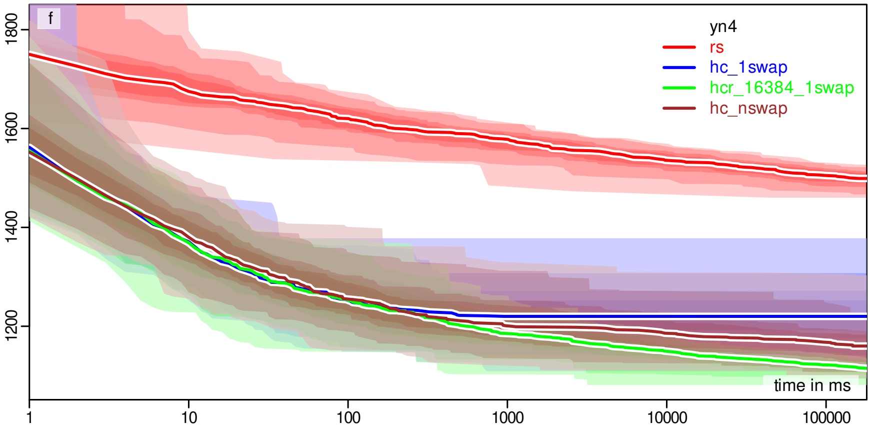
Progress over Time



What progress does the algorithm make over time?

- The rare larger moves allow it to escape from local optima that would trap `hc_1swap`.
- The hill climber with restarts seems to improve longer.

Progress over Time



What progress does the algorithm make over time?

- The rare larger moves allow it to escape from local optima that would trap `hc_1swap`.
- The hill climber with restarts seems to improve longer.

Improved Algorithm Concept 3

Combining the Two Improvements

- Now we know two ways to improve the performance of our hill climber.

Combining the Two Improvements

- Now we know two ways to improve the performance of our hill climber:
  1. we can restart it

Combining the Two Improvements

- Now we know two ways to improve the performance of our hill climber:
  1. we can restart it and
  2. we can use a unary operator with larger neighborhood that still mostly makes small steps.

Combining the Two Improvements

- Now we know two ways to improve the performance of our hill climber:
  1. we can restart it and
  2. we can use a unary operator with larger neighborhood that still mostly makes small steps.
- It is only natural to try to combine these two improvements.

Configuring the Algorithm

		makespan				last improvement	
$\mathcal{I}$	algo	best	mean	med	sd	med(t)	med(FEs)
abz7	hc_1swap	717	800	798	28	0s	16'978
	hc_nswap	724	758	758	17	35s	7'781'762
la24	hc_1swap	999	1095	1086	56	0s	6'612
	hc_nswap	945	1018	1016	29	25s	9'072'935
swv15	hc_1swap	3837	4108	4108	137	1s	104'598
	hc_nswap	3602	3880	3872	112	70s	8'351'112
yn4	hc_1swap	1109	1222	1220	48	0s	31'789
	hc_nswap	1095	1162	1160	34	71s	11'016'757

Configuring the Algorithm

- The `hc_nswap` improves longer than `hc_1swap`

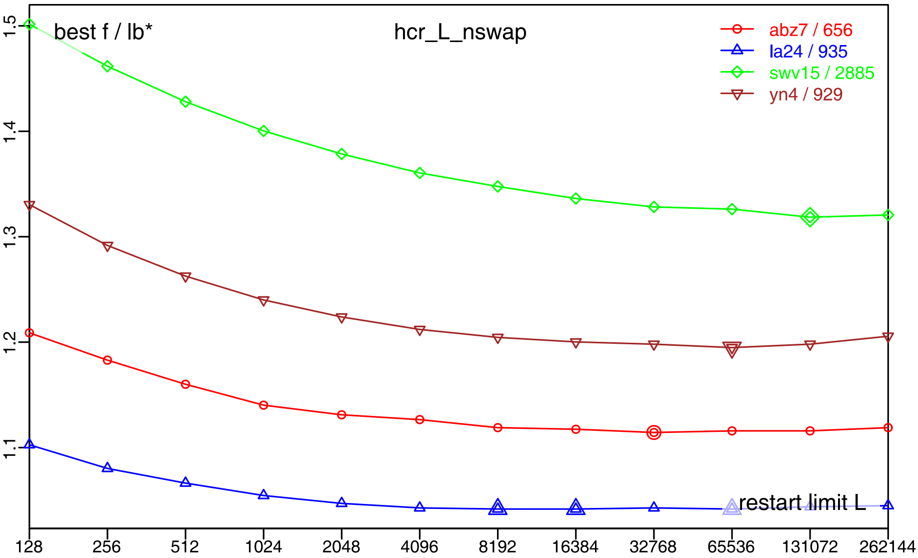
		makespan				last improvement	
$\mathcal{I}$	algo	best	mean	med	sd	med(t)	med(FEs)
abz7	hc_1swap	717	800	798	28	0s	16'978
	hc_nswap	724	758	758	17	35s	7'781'762
la24	hc_1swap	999	1095	1086	56	0s	6'612
	hc_nswap	945	1018	1016	29	25s	9'072'935
swv15	hc_1swap	3837	4108	4108	137	1s	104'598
	hc_nswap	3602	3880	3872	112	70s	8'351'112
yn4	hc_1swap	1109	1222	1220	48	0s	31'789
	hc_nswap	1095	1162	1160	34	71s	11'016'757

Configuring the Algorithm

- The `hc_nswap` improves longer than `hc_1swap`
- We can expect that the number L of unsuccessful steps before a restart should be higher now.

		makespan				**last improvement**	
$\mathcal{I}$	algo	best	mean	med	sd	med(t)	med(FEs)
abz7	hc_1swap	717	800	798	28	0s	16'978
	hc_nswap	724	758	758	17	35s	7'781'762
la24	hc_1swap	999	1095	1086	56	0s	6'612
	hc_nswap	945	1018	1016	29	25s	9'072'935
swv15	hc_1swap	3837	4108	4108	137	1s	104'598
	hc_nswap	3602	3880	3872	112	70s	8'351'112
yn4	hc_1swap	1109	1222	1220	48	0s	31'789
	hc_nswap	1095	1162	1160	34	71s	11'016'757

Configuring the Algorithm



Configuring the Algorithm

- The hc_nswap improves longer than hc_1swap
- We can expect that the number L of unsuccessful steps before a restart should be higher now.
- Let's choose $L = 65'536$, i.e., hcr_65536_nswap.

		makespan				**last improvement**	
$\mathcal{I}$	algo	best	mean	med	sd	med(t)	med(FEs)
abz7	hc_1swap	717	800	798	28	0s	16'978
	hc_nswap	724	758	758	17	35s	7'781'762
la24	hc_1swap	999	1095	1086	56	0s	6'612
	hc_nswap	945	1018	1016	29	25s	9'072'935
swv15	hc_1swap	3837	4108	4108	137	1s	104'598
	hc_nswap	3602	3880	3872	112	70s	8'351'112
yn4	hc_1swap	1109	1222	1220	48	0s	31'789
	hc_nswap	1095	1162	1160	34	71s	11'016'757

Experiment and Analysis

So what do we get?

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

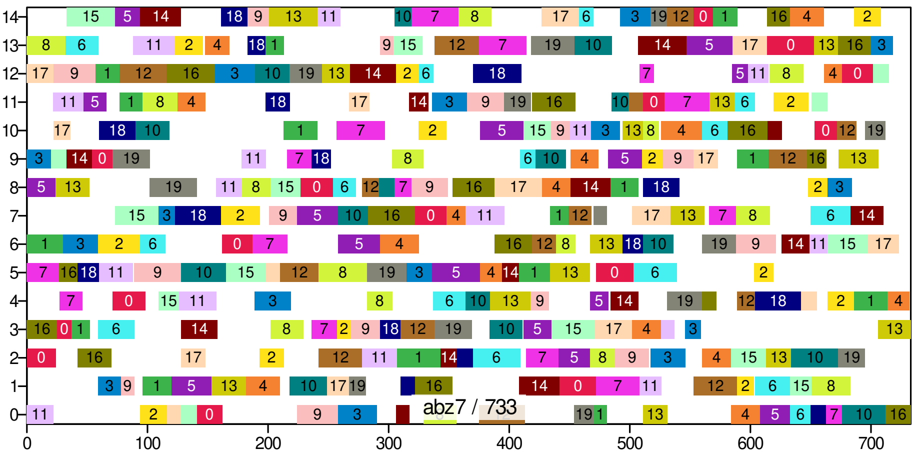
So what do we get?

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

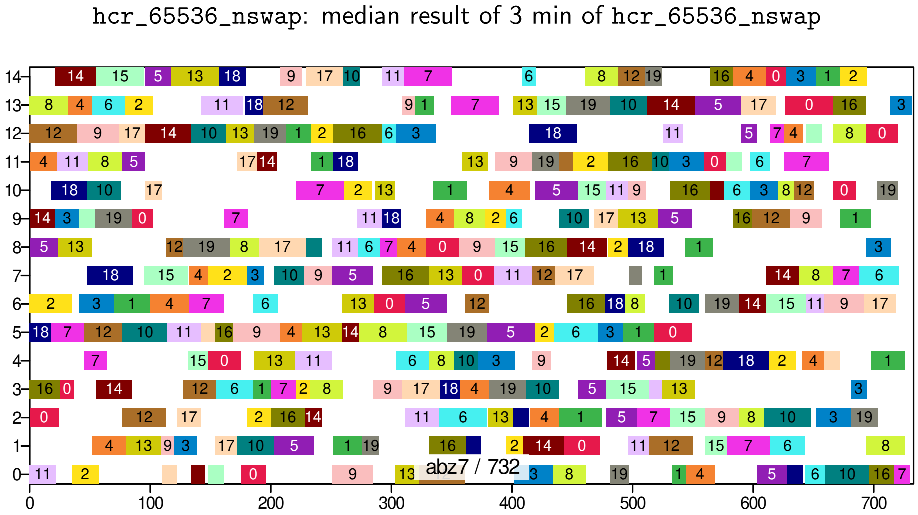
		makespan				last improvement	
$\mathcal{I}$	algo	best	mean	med	sd	med(t)	med(FEs)
abz7	hcr_16384_1swap	714	732	733	**6**	91s	18'423'530
	hc_nswap	724	758	758	17	35s	7'781'762
	hcr_65536_nswap	**712**	**731**	**732**	6	96s	21'189'358
la24	hcr_16384_1swap	953	976	976	**7**	80s	34'437'999
	hc_nswap	945	1018	1016	29	25s	9'072'935
	hcr_65536_nswap	**942**	**973**	**974**	8	71s	31'466'420
swv15	hcr_16384_1swap	3752	3859	3861	42	92s	11'756'497
	hc_nswap	**3602**	3880	3872	112	70s	8'351'112
	hcr_65536_nswap	3740	**3818**	**3826**	**35**	89s	10'783'296
yn4	hcr_16384_1swap	1081	1115	1115	**11**	91s	14'804'358
	hc_nswap	1095	1162	1160	34	71s	11'016'757
	hcr_65536_nswap	**1068**	**1109**	**1110**	12	78s	18'756'636

So what do we get?

`hcr_16384_1swap`: median result of 3 min of `hcr_16384_1swap`

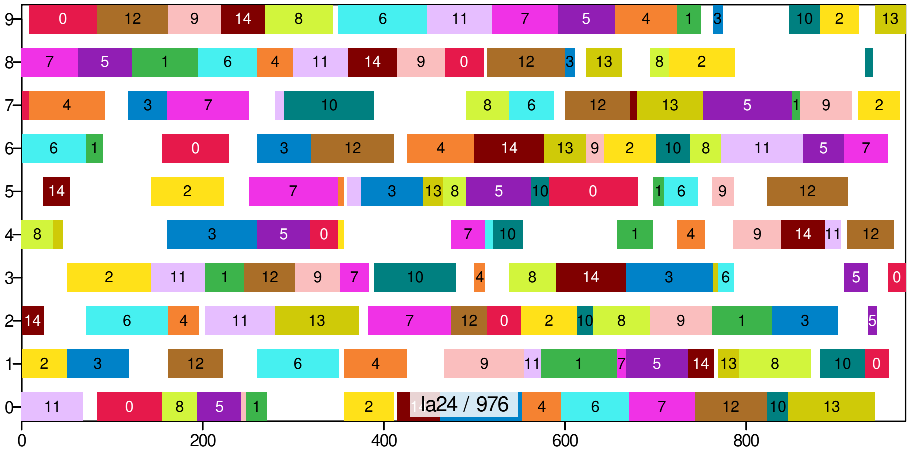


So what do we get?

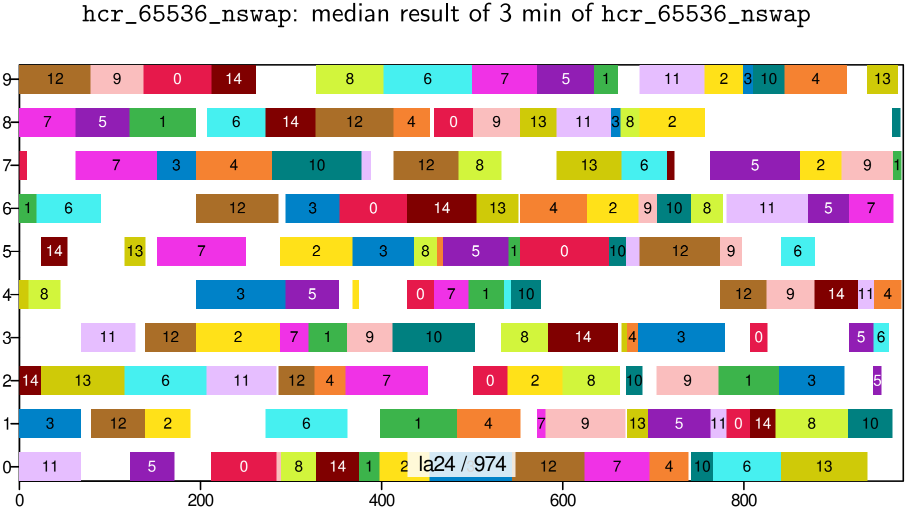


So what do we get?

hcr_16384_1swap: median result of 3 min of hcr_16384_1swap

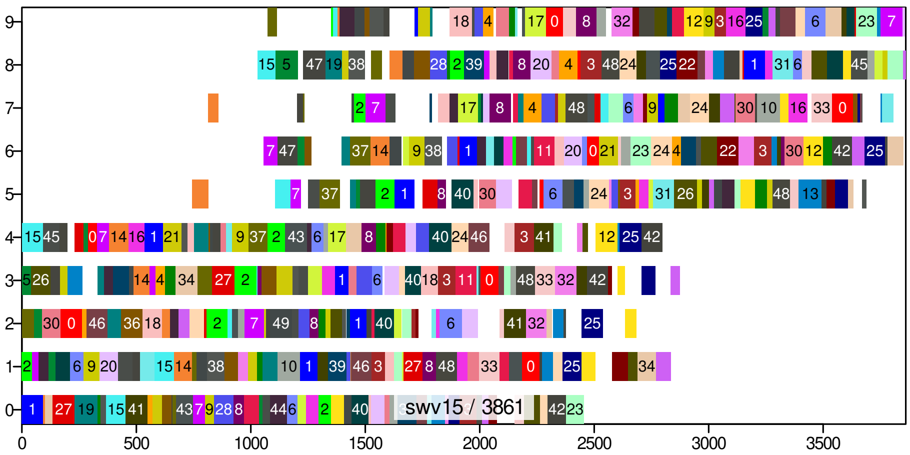


So what do we get?

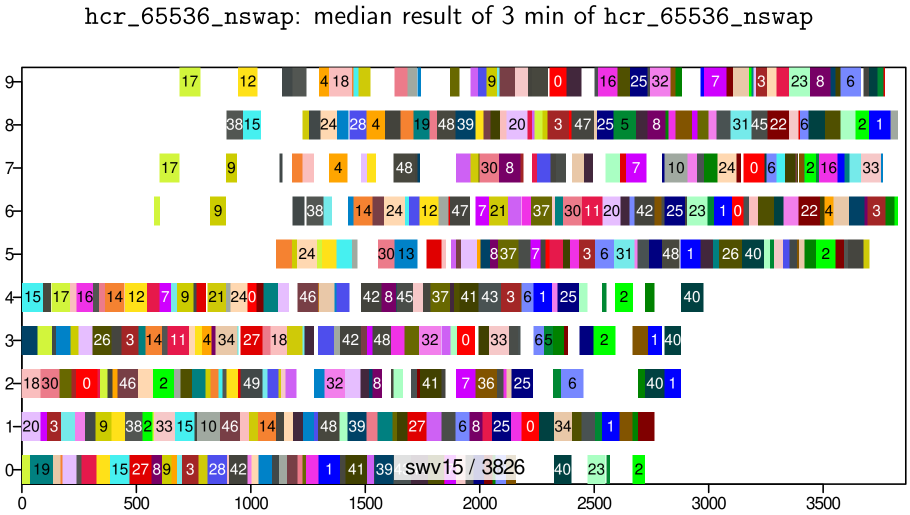


So what do we get?

hcr_16384_1swap: median result of 3 min of hcr_16384_1swap

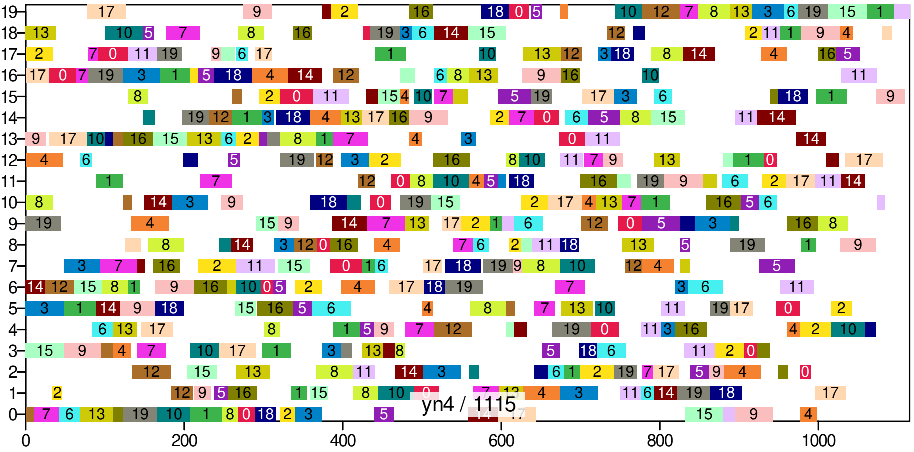


So what do we get?

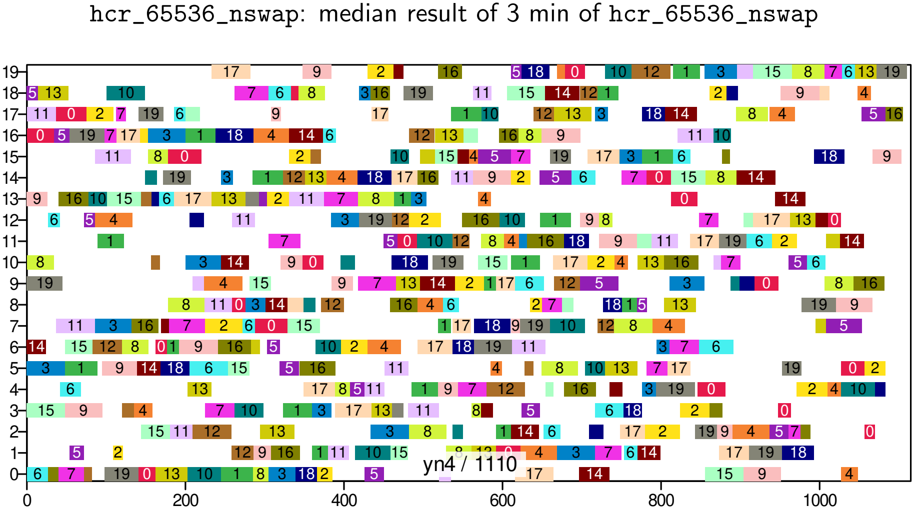


So what do we get?

`hcr_16384_1swap`: median result of 3 min of `hcr_16384_1swap`



So what do we get?

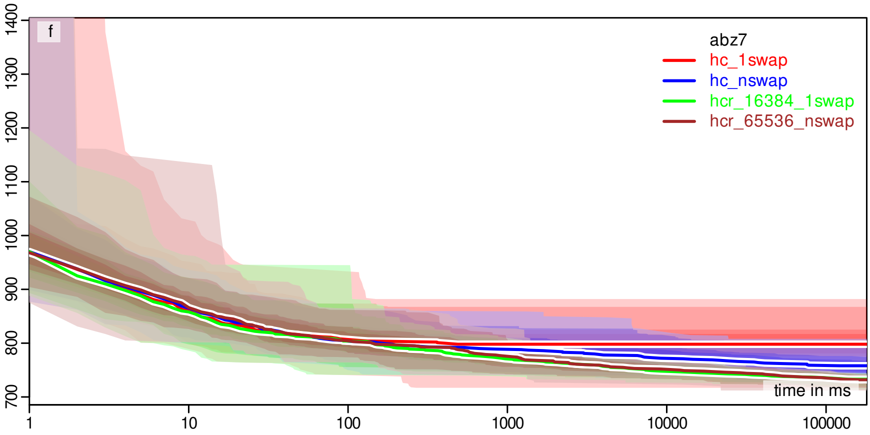


Progress over Time

What progress does the algorithm make over time?

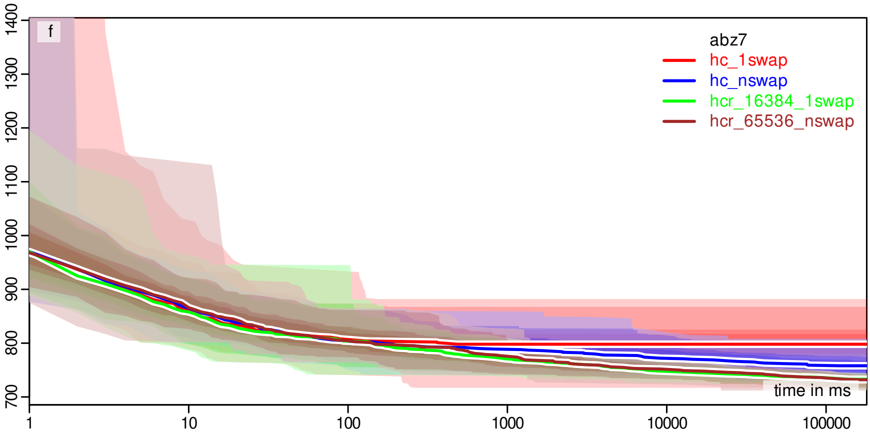
Progress over Time

What progress does the algorithm make over time?



Progress over Time

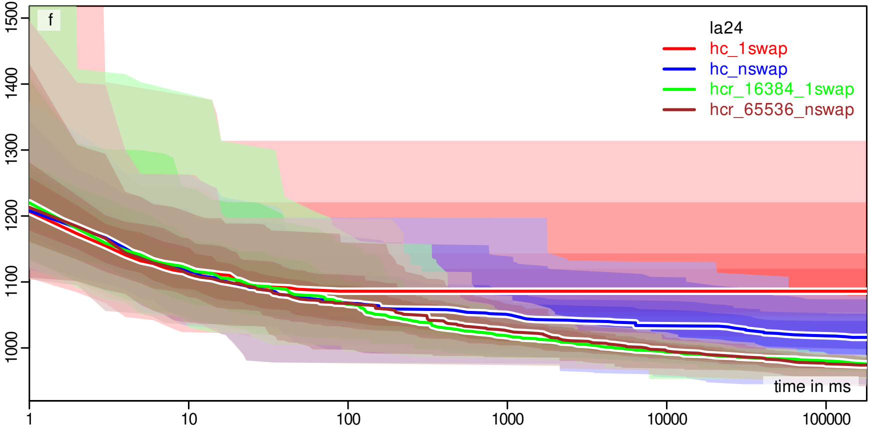
What progress does the algorithm make over time?



`hcr_nswap` tends to be a tiny little bit better than `hcr_1swap` ... but not much

Progress over Time

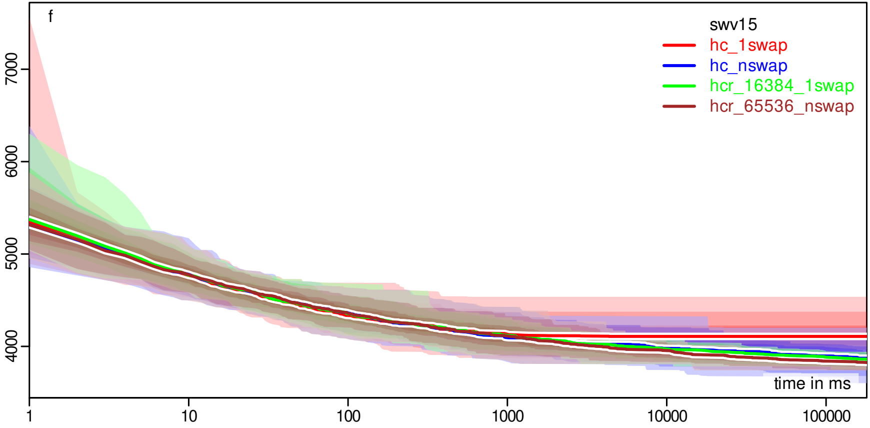
What progress does the algorithm make over time?



`hcr_nswap` tends to be a tiny little bit better than `hcr_1swap` ... but not much

Progress over Time

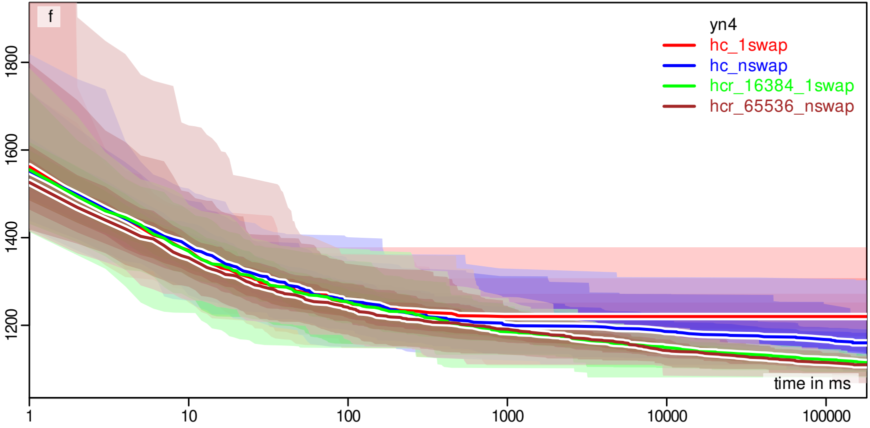
What progress does the algorithm make over time?



`hcr_nswap` tends to be a tiny little bit better than `hcr_1swap` ... but not much

Progress over Time

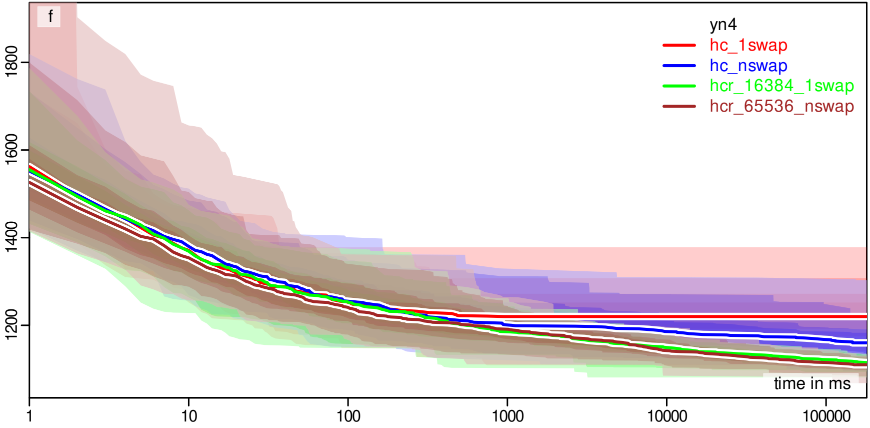
What progress does the algorithm make over time?



`hcr_nswap` tends to be a tiny little bit better than `hcr_1swap` ... but not much

Progress over Time

What progress does the algorithm make over time?



`hcr_nswap` tends to be a tiny little bit better than `hcr_1swap` ... but not much

Summary

Summary

- We now have learned a second, more efficient metaheuristic optimization algorithm: stochastic hill climber.

Summary

- We now have learned a second, more efficient metaheuristic optimization algorithm: stochastic hill climber.
- By making use of the best point in the search space we have seen so far and iteratively trying to improve it, we can dramatically improve the results compared to random sampling.

Summary

- We now have learned a second, more efficient metaheuristic optimization algorithm: stochastic hill climber.
- By making use of the best point in the search space we have seen so far and iteratively trying to improve it, we can dramatically improve the results compared to random sampling.
- Like random sampling, we can apply it to all sorts of problems, as long as we provide the basic structural ingredients.

Summary

- We now have learned a second, more efficient metaheuristic optimization algorithm: stochastic hill climber.
- By making use of the best point in the search space we have seen so far and iteratively trying to improve it, we can dramatically improve the results compared to random sampling.
- Like random sampling, we can apply it to all sorts of problems, as long as we provide the basic structural ingredients.
- Hill climbing is a local search and vulnerable to get trapped in local optima.

Summary

- We now have learned a second, more efficient metaheuristic optimization algorithm: stochastic hill climber.
- By making use of the best point in the search space we have seen so far and iteratively trying to improve it, we can dramatically improve the results compared to random sampling.
- Like random sampling, we can apply it to all sorts of problems, as long as we provide the basic structural ingredients.
- Hill climbing is a local search and vulnerable to get trapped in local optima.
- We can try to work around that by implementing good search operators and by restarting the algorithm.

谢谢

Thank you

References I

1. Thomas Weise. *An Introduction to Optimization Algorithms*. Institute of Applied Optimization (IAO) [应用优化研究所] of the School of Artificial Intelligence and Big Data [人工智能与大数据学院] of Hefei University [合肥学院], Hefei [合肥市], Anhui [安徽省], China [中国], 2018–2020. URL http://thomasweise.github.io/aitoa/.
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. Holger H. Hoos and Thomas Stützle. *Stochastic Local Search: Foundations and Applications*. The Morgan Kaufmann Series in Artificial Intelligence. Elsevier, 2005. ISBN 1493303732.
4. Stuart Jonathan Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (AIMA)*. Prentice Hall International Inc., Upper Saddle River, NJ, USA, 2 edition, 2002. ISBN 0-13-080302-2.
5. James C. Spall. *Introduction to Stochastic Search and Optimization*, volume 6 of *Estimation, Simulation, and Control – Wiley-Interscience Series in Discrete Mathematics and Optimization*. Wiley Interscience, Chichester, West Sussex, UK, April 2003. ISBN 0-471-33052-3. URL http://www.jhuapl.edu/ISSO/.
6. Ingo Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. PhD thesis, Technische Universität Berlin, Berlin, Germany, 1971–1973.
7. Ingo Rechenberg. *Evolutionsstrategie '94*, volume 1 of *Werkstatt Bionik und Evolutionstechnik*. Frommann-Holzboog Verlag, Bad Cannstadt, Stuttgart, Baden-Württemberg, Germany, 1994. ISBN 3-7728-1642-8.
8. 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.
9. 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.