合肥学院
HEFEI UNIVERSITY

Optimization Algorithms

6. Evolutionary Algorithms

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

Introduction

- Hill Climbers are local search.

Introduction

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

Introduction

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

Introduction

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

Introduction

- Hill Climbers are local search.
- They begin at some point x in the search space and then investigate its neighborhood $N(x)$.
- The neighborhood is defined by the (unary) search operator, in our case `1swap` or `nswap`.
- If they reach a local optimum $x^{\times}$, they get trapped.
- We then can restart them, but this means
    1. to start again back at “zero” and losing all accumulated information

Introduction

- Hill Climbers are local search.
- They begin at some point x in the search space and then investigate its neighborhood $N(x)$.
- The neighborhood is defined by the (unary) search operator, in our case `1swap` or `nswap`.
- If they reach a local optimum $x^{\times}$, they get trapped.
- We then can restart them, but this means
  1. to start again back at "zero" and losing all accumulated information and
  2. they may still land again in a local optimum.

Introduction

- Hill Climbers are local search.
- They begin at some point x in the search space and then investigate its neighborhood $N(x)$.
- The neighborhood is defined by the (unary) search operator, in our case `1swap` or `nswap`.
- If they reach a local optimum $x^{\times}$, they get trapped.
- We then can restart them, but this means
    1. to start again back at "zero" and losing all accumulated information and
    2. they may still land again in a local optimum.
- We can use unary operators which sample non-uniformly from larger neighborhoods, like `nswap`, but the search move needed to escape from a good but non-optimal point might be too unlikely.

Introduction

- Hill Climbers are local search.
- They begin at some point x in the search space and then investigate its neighborhood $N(x)$.
- The neighborhood is defined by the (unary) search operator, in our case `1swap` or `nswap`.
- If they reach a local optimum $x^{\times}$, they get trapped.
- We then can restart them, but this means
    1. to start again back at "zero" and losing all accumulated information and
    2. they may still land again in a local optimum.
- We can use unary operators which sample non-uniformly from larger neighborhoods, like `nswap`, but the search move needed to escape from a good but non-optimal point might be too unlikely.
- Idea: We could investigate multiple points in the search space at once and use the additional information in a clever way?

Population-Based Metaheuristics

- Population-based metaheuristics[2–6] try to maintain a set of points in the search space which are iteratively refined.

Population-Based Metaheuristics

- Population-based metaheuristics[2–6] try to maintain a set of points in the search space which are iteratively refined.
- This has a couple of advantages

Population-Based Metaheuristics

- Population-based metaheuristics[2–6] try to maintain a set of points in the search space which are iteratively refined.
- This has a couple of advantages:
    - We are less likely to get trapped in a single local optimum (because we work on multiple points).

Population-Based Metaheuristics

- Population-based metaheuristics[2–6] try to maintain a set of points in the search space which are iteratively refined.
- This has a couple of advantages:
  - We are less likely to get trapped in a single local optimum (because we work on multiple points).
  - We might more likely find a better (local) optimum.

Population-Based Metaheuristics

- Population-based metaheuristics[2–6] try to maintain a set of points in the search space which are iteratively refined.
- This has a couple of advantages:
  - We are less likely to get trapped in a single local optimum (because we work on multiple points).
  - We might more likely find a better (local) optimum.
  - If we have different good points from the search space in our population, we can try to use this additional information...

Algorithm Concept: Population

$(\mu + \lambda)$ EA

- Evolutionary Algorithms (EAs) are the most successful family of population-based metaheuristics.[2 4 5]

$(\mu + \lambda)$ **EA**

- Evolutionary Algorithms (EAs) are the most successful family of population-based metaheuristics.[2 4 5]
- Here we focus on $(\mu + \lambda)$ EAs

$(\mu + \lambda)$ EA

- Evolutionary Algorithms (EAs) are the most successful family of population-based metaheuristics.[2 4 5]
- Here we focus on $(\mu + \lambda)$ EAs, which work as follows:
    1. Generate a population of $\mu + \lambda$ random points in the search space (and map them to solutions and evaluate them).

$(\mu + \lambda)$ EA

- Evolutionary Algorithms (EAs) are the most successful family of population-based metaheuristics.[2 4 5]
- Here we focus on $(\mu + \lambda)$ EAs, which work as follows:
    1. Generate a population of $\mu + \lambda$ random points in the search space (and map them to solutions and evaluate them).
    2. From the population, select the μ best points as "parents" for the next "generation" of points, discard the remaining λ points.

$(\mu + \lambda)$ EA

- Evolutionary Algorithms (EAs) are the most successful family of population-based metaheuristics.[2 4 5]
- Here we focus on $(\mu + \lambda)$ EAs, which work as follows:
  1. Generate a population of $\mu + \lambda$ random points in the search space (and map them to solutions and evaluate them).
  2. From the population, select the μ best points as "parents" for the next "generation" of points, discard the remaining λ points.
  3. Generate λ new "offspring" points by applying a unary search operator (which creates a randomly modified copy from a selected point).

$(\mu + \lambda)$ EA

- Evolutionary Algorithms (EAs) are the most successful family of population-based metaheuristics.[2 4 5]
- Here we focus on $(\mu + \lambda)$ EAs, which work as follows:
  1. Generate a population of $\mu + \lambda$ random points in the search space (and map them to solutions and evaluate them).
  2. From the population, select the μ best points as "parents" for the next "generation" of points, discard the remaining λ points.
  3. Generate λ new "offspring" points by applying a unary search operator (which creates a randomly modified copy from a selected point).
  4. Evaluate the λ offsprings, add them to the population, and go back to step 2..

Ingredient: Solution Record

```java
package aitoa.structure;

public class Record<X> {

  /** The comparator to be used for sorting according
      quality */
  public static final Comparator<Record<?>> BY_QUALITY =
      (a, b) -> Double.compare(a.quality, b.quality);

  /** the point in the search space */
  public final X x;
  /** the quality */
  public double quality;

// unnecessary stuff omitted here...
}
```

Evolutionary Algorithm Implementation

```
package aitoa.algorithms;

public class EA<X, Y> {
// abridged code: unnecessary stuff omitted here and in function solve...
//
//
//
//

//
//
//
//
//
//

//
//
//
//
//
//
//
//
//
//
//
//
//
//
} // end class
```

Evolutionary Algorithm Implementation

```
package aitoa.algorithms;

public class EA<X, Y> extends Metaheuristic2<X, Y> {
// abridged code: unnecessary stuff omitted here and in function solve...
  public void solve(IBlackBoxProcess<X, Y> process) {
//
//
//

//
//
//
//
//
//

//
//
//
//
//
//
//
//
//
//
//
//
//
  } // end solve
} // end class
```

Evolutionary Algorithm Implementation

```java
package aitoa.algorithms;

public class EA<X, Y> extends Metaheuristic2<X, Y> {
// abridged code: unnecessary stuff omitted here and in function solve...
  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
//
//

//
//
//
//
//
//

//
//
//
//
//
//
//
//
//
//
//
//
//
  } // end solve
} // end class
```

Evolutionary Algorithm Implementation

```java
package aitoa.algorithms;

public class EA<X, Y> extends Metaheuristic2<X, Y> {
// abridged code: unnecessary stuff omitted here and in function solve...
  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
//

//
//
//
//
//
//

//
//
//
//
//
//
//
//
//
//
//
//
//
  } // end solve
} // end class
```

Evolutionary Algorithm Implementation

```
package aitoa.algorithms;

public class EA<X, Y> extends Metaheuristic2<X, Y> {
// abridged code: unnecessary stuff omitted here and in function solve...
  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

//
//
//
//
//
//

//
//
//
//
//
//
//
//
//
//
//
//
//
  } // end solve
} // end class
```

Evolutionary Algorithm Implementation

```
package aitoa.algorithms;

public class EA<X, Y> extends Metaheuristic2<X, Y> {
// abridged code: unnecessary stuff omitted here and in function solve...
  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
//
//
//
//
    } // end of filling the first population

//
//
//
//
//
//
//
//
//
//
//
//
//
  } // end solve
} // end class
```

Evolutionary Algorithm Implementation

```java
package aitoa.algorithms;

public class EA<X, Y> extends Metaheuristic2<X, Y> {
// abridged code: unnecessary stuff omitted here and in function solve...
  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();  // allocate point
//
//
//
    } // end of filling the first population

//
//
//
//
//
//
//
//
//
//
//
//
//
  } // end solve
} // end class
```

Evolutionary Algorithm Implementation

```java
package aitoa.algorithms;

public class EA<X, Y> extends Metaheuristic2<X, Y> {
// abridged code: unnecessary stuff omitted here and in function solve...
  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
//
//
    } // end of filling the first population

//
//
//
//
//
//
//
//
//
//
//
//
//
  } // end solve
} // end class
```

Evolutionary Algorithm Implementation

```java
package aitoa.algorithms;

public class EA<X, Y> extends Metaheuristic2<X, Y> {
// abridged code: unnecessary stuff omitted here and in function solve...
  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
//
    } // end of filling the first population

//
//
//
//
//
//
//
//
//
//
//
//
//
  } // end solve
} // end class
```

Evolutionary Algorithm Implementation

```
package aitoa.algorithms;

public class EA<X, Y> extends Metaheuristic2<X, Y> {
// abridged code: unnecessary stuff omitted here and in function solve...
  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
//
//
//
//
//
//
//
//
//
//
//
//
  } // end solve
} // end class
```

Evolutionary Algorithm Implementation

```java
package aitoa.algorithms;

public class EA<X, Y> extends Metaheuristic2<X, Y> {
// abridged code: unnecessary stuff omitted here and in function solve...
  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
    Arrays.sort(P, Record.BY_QUALITY); // sort the population: mu best at front
//
//
//
//
//
//
//
//
//
//
//
  } // end solve
} // end class
```

Evolutionary Algorithm Implementation

```
package aitoa.algorithms;

public class EA<X, Y> extends Metaheuristic2<X, Y> {
// abridged code: unnecessary stuff omitted here and in function solve...
  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
    Arrays.sort(P, Record.BY_QUALITY); // sort the population: mu best at front
    RandomUtils.shuffle(random, P, 0, this.mu); // shuffle parents for fairness
//
//
//
//
//
//
//
//
//
//
  } // end solve
} // end class
```

Evolutionary Algorithm Implementation

```
package aitoa.algorithms;

public class EA<X, Y> extends Metaheuristic2<X, Y> {
// abridged code: unnecessary stuff omitted here and in function solve...
  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
    Arrays.sort(P, Record.BY_QUALITY); // sort the population: mu best at front
    RandomUtils.shuffle(random, P, 0, this.mu); // shuffle parents for fairness
    int p1 = -1; // index to iterate over first parent
//
//
//
//
//
//
//
//
//
  } // end solve
} // end class
```

Evolutionary Algorithm Implementation

```
package aitoa.algorithms;

public class EA<X, Y> extends Metaheuristic2<X, Y> {
// abridged code: unnecessary stuff omitted here and in function solve...
  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
    Arrays.sort(P, Record.BY_QUALITY); // sort the population: mu best at front
    RandomUtils.shuffle(random, P, 0, this.mu); // shuffle parents for fairness
    int p1 = -1; // index to iterate over first parent
    for (int index = P.length; (--index) >= this.mu;) { // overwrite lambda worst
//
//
//
//
//
//
    } // the end of the offspring generation
//
  } // end solve
} // end class
```

Evolutionary Algorithm Implementation

```java
package aitoa.algorithms;

public class EA<X, Y> extends Metaheuristic2<X, Y> {
// abridged code: unnecessary stuff omitted here and in function solve...
  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
    Arrays.sort(P, Record.BY_QUALITY); // sort the population: mu best at front
    RandomUtils.shuffle(random, P, 0, this.mu); // shuffle parents for fairness
    int p1 = -1; // index to iterate over first parent
    for (int index = P.length; (--index) >= this.mu;) { // overwrite lambda worst
      if (process.shouldTerminate()) return;
//
//
//
//
//
    } // the end of the offspring generation
//
  } // end solve
} // end class
```

Evolutionary Algorithm Implementation

```java
package aitoa.algorithms;

public class EA<X, Y> extends Metaheuristic2<X, Y> {
// abridged code: unnecessary stuff omitted here and in function solve...
  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
    Arrays.sort(P, Record.BY_QUALITY); // sort the population: mu best at front
    RandomUtils.shuffle(random, P, 0, this.mu); // shuffle parents for fairness
    int p1 = -1; // index to iterate over first parent
    for (int index = P.length; (--index) >= this.mu;) { // overwrite lambda worst
      if (process.shouldTerminate()) return;
      Record<X> dest = P[index];
//
//
//
//
    } // the end of the offspring generation
//
  } // end solve
} // end class
```

Evolutionary Algorithm Implementation

```java
package aitoa.algorithms;

public class EA<X, Y> extends Metaheuristic2<X, Y> {
// abridged code: unnecessary stuff omitted here and in function solve...
  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
    Arrays.sort(P, Record.BY_QUALITY); // sort the population: mu best at front
    RandomUtils.shuffle(random, P, 0, this.mu); // shuffle parents for fairness
    int p1 = -1; // index to iterate over first parent
    for (int index = P.length; (--index) >= this.mu;) { // overwrite lambda worst
      if (process.shouldTerminate()) return;
      Record<X> dest = P[index];
      p1 = (p1 + 1) % this.mu; // step the parent 1 index
//
//
//
    } // the end of the offspring generation
//
  } // end solve
} // end class
```

Evolutionary Algorithm Implementation

```
package aitoa.algorithms;

public class EA<X, Y> extends Metaheuristic2<X, Y> {
// abridged code: unnecessary stuff omitted here and in function solve...
  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
    Arrays.sort(P, Record.BY_QUALITY); // sort the population: mu best at front
    RandomUtils.shuffle(random, P, 0, this.mu); // shuffle parents for fairness
    int p1 = -1; // index to iterate over first parent
    for (int index = P.length; (--index) >= this.mu;) { // overwrite lambda worst
      if (process.shouldTerminate()) return;
      Record<X> dest = P[index];
      p1 = (p1 + 1) % this.mu; // step the parent 1 index
      Record<X> sel  = P[p1];
//
//
    } // the end of the offspring generation
//
  } // end solve
} // end class
```

Evolutionary Algorithm Implementation

```
package aitoa.algorithms;

public class EA<X, Y> extends Metaheuristic2<X, Y> {
// abridged code: unnecessary stuff omitted here and in function solve...
  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
    Arrays.sort(P, Record.BY_QUALITY); // sort the population: mu best at front
    RandomUtils.shuffle(random, P, 0, this.mu); // shuffle parents for fairness
    int p1 = -1; // index to iterate over first parent
    for (int index = P.length; (--index) >= this.mu;) { // overwrite lambda worst
      if (process.shouldTerminate()) return;
      Record<X> dest = P[index];
      p1 = (p1 + 1) % this.mu; // step the parent 1 index
      Record<X> sel  = P[p1];
      this.unary.apply(sel.x, dest.x, random); // generate offspring
//
    } // the end of the offspring generation
//
  } // end solve
} // end class
```

Evolutionary Algorithm Implementation

```java
package aitoa.algorithms;

public class EA<X, Y> extends Metaheuristic2<X, Y> {
// abridged code: unnecessary stuff omitted here and in function solve...
  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
    Arrays.sort(P, Record.BY_QUALITY); // sort the population: mu best at front
    RandomUtils.shuffle(random, P, 0, this.mu); // shuffle parents for fairness
    int p1 = -1; // index to iterate over first parent
    for (int index = P.length; (--index) >= this.mu;) { // overwrite lambda worst
      if (process.shouldTerminate()) return;
      Record<X> dest = P[index];
      p1 = (p1 + 1) % this.mu; // step the parent 1 index
      Record<X> sel  = P[p1];
      this.unary.apply(sel.x, dest.x, random); // generate offspring
      dest.quality = process.evaluate(dest.x); // evaluate offspring
    } // the end of the offspring generation
//
  } // end solve
} // end class
```

Evolutionary Algorithm Implementation

```java
package aitoa.algorithms;

public class EA<X, Y> extends Metaheuristic2<X, Y> {
// abridged code: unnecessary stuff omitted here and in function solve...
  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

    for (;;) { // main loop: one iteration = one generation
      Arrays.sort(P, Record.BY_QUALITY); // sort the population: mu best at front
      RandomUtils.shuffle(random, P, 0, this.mu); // shuffle parents for fairness
      int p1 = -1; // index to iterate over first parent
      for (int index = P.length; (--index) >= this.mu;) { // overwrite lambda worst
        if (process.shouldTerminate()) return;
        Record<X> dest = P[index];
        p1 = (p1 + 1) % this.mu; // step the parent 1 index
        Record<X> sel  = P[p1];
        this.unary.apply(sel.x, dest.x, random); // generate offspring
        dest.quality = process.evaluate(dest.x); // evaluate offspring
      } // the end of the offspring generation
    } // the end of the main loop
  } // end solve
} // end class
```

Experiment and Analysis

Configuring the Algorithm

- Our EA has two parameters, μ and λ.

Configuring the Algorithm

- Our EA has two parameters, μ and λ.
- Actually, it has three parameters.

Configuring the Algorithm

- Our EA has two parameters, μ and λ.
- Actually, it has three parameters: We can choose `1swap` or `nswap` as unary search operation.

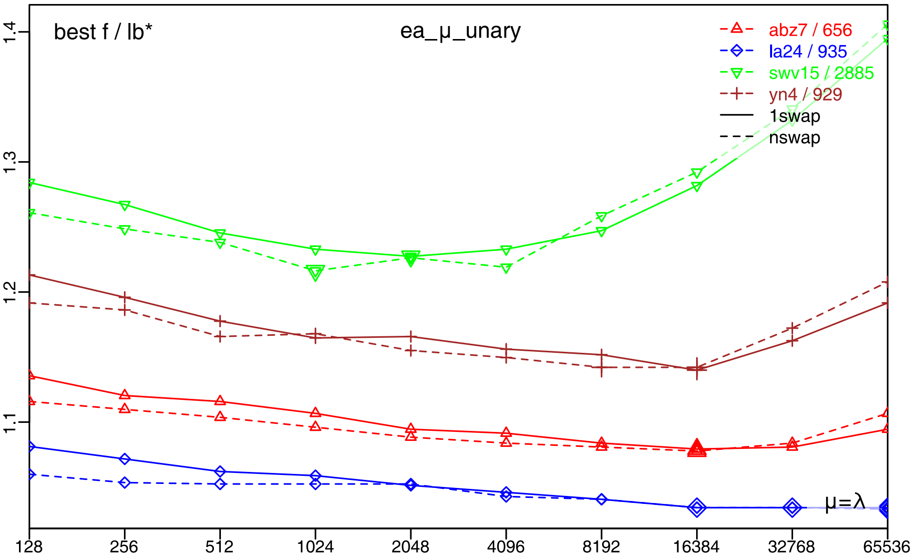
Configuring the Algorithm

- Our EA has two parameters, μ and λ.
- Actually, it has three parameters: We can choose `1swap` or `nswap` as unary search operation.
- For now, let's set $\mu = \lambda$, meaning the number of parents equals the number of offspring in each generation.

Configuring the Algorithm

- Our EA has two parameters, μ and λ.
- Actually, it has three parameters: We can choose `1swap` or `nswap` as unary search operation.
- For now, let's set $\mu = \lambda$, meaning the number of parents equals the number of offspring in each generation.
- This leaves us two parameters to investigate, so let's take a look.

Configuring the Algorithm



Configuring the Algorithm

- Our EA has two parameters, μ and λ.
- Actually, it has three parameters: We can choose `1swap` or `nswap` as unary search operation.
- For now, let's set $\mu = \lambda$, meaning the number of parents equals the number of offspring in each generation.
- This leaves us two parameters to investigate, so let's take a look.
- Except for `swv15`, a setting of $\mu = \lambda = 16'384$ seems reasonable.

Configuring the Algorithm

- Our EA has two parameters, μ and λ.
- Actually, it has three parameters: We can choose `1swap` or `nswap` as unary search operation.
- For now, let's set $\mu = \lambda$, meaning the number of parents equals the number of offspring in each generation.
- This leaves us two parameters to investigate, so let's take a look.
- Except for `swv15`, a setting of $\mu = \lambda = 16'384$ seems reasonable.
- Interestingly, there are only little differences between `1swap` and `nswap`.

Configuring the Algorithm

- Our EA has two parameters, μ and λ.
- Actually, it has three parameters: We can choose `1swap` or `nswap` as unary search operation.
- For now, let's set $\mu = \lambda$, meaning the number of parents equals the number of offspring in each generation.
- This leaves us two parameters to investigate, so let's take a look.
- Except for `swv15`, a setting of $\mu = \lambda = 16'384$ seems reasonable.
- Interestingly, there are only little differences between `1swap` and `nswap`, but we pick `nswap` because it tends to be the better choice more often.

Configuring the Algorithm

- Our EA has two parameters, μ and λ.
- Actually, it has three parameters: We can choose `1swap` or `nswap` as unary search operation.
- For now, let's set $\mu = \lambda$, meaning the number of parents equals the number of offspring in each generation.
- This leaves us two parameters to investigate, so let's take a look.
- Except for `swv15`, a setting of $\mu = \lambda = 16'384$ seems reasonable.
- Interestingly, there are only little differences between `1swap` and `nswap`, but we pick `nswap` because it tends to be the better choice more often.
- Generally, the EA seems to be quite robust and performs well for many parameter settings (except on `swv15`).

So what do we get?

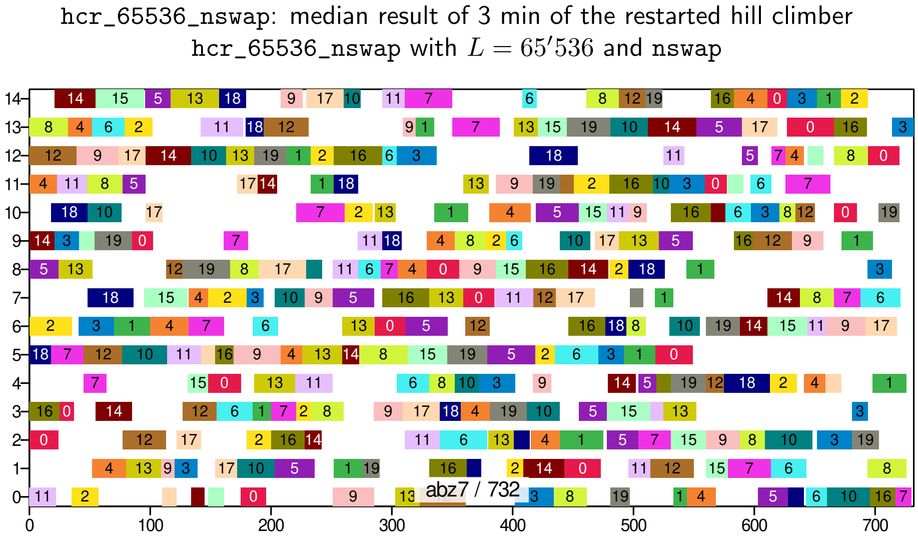
- I execute the program 101 times for each of the instances `abz7`, `la24`, `swv15`, and `yn4`

So what do we get?

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

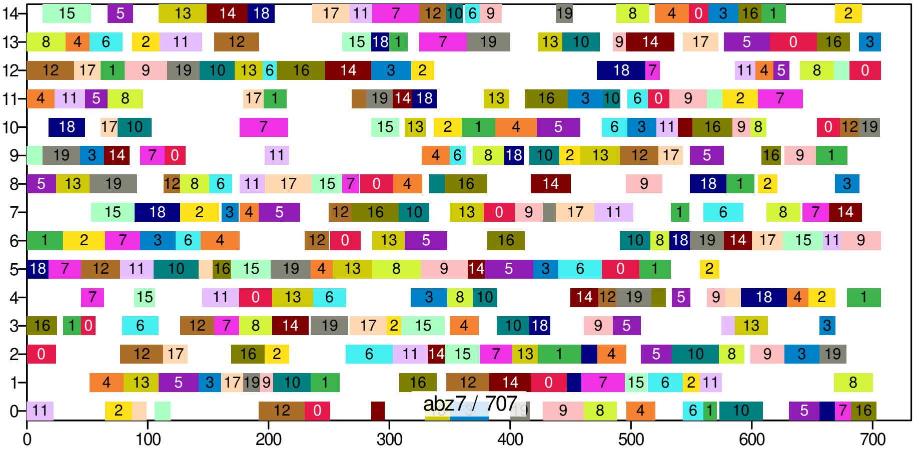
		makespan				last improvement	
$\mathcal{I}$	algo	best	mean	med	sd	med(t)	med(FEs)
abz7	hcr_65536_nswap	712	731	732	**6**	96s	21'189'358
	ea_16384_nswap	**691**	**707**	**707**	8	151s	25'293'859
la24	hcr_65536_nswap	**942**	973	974	**8**	71s	31'466'420
	ea_16384_nswap	945	**968**	**967**	12	39s	10'161'119
swv15	hcr_65536_nswap	3740	3818	3826	**35**	89s	10'783'296
	ea_16384_nswap	**3577**	**3723**	**3728**	50	178s	18'897'833
yn4	hcr_65536_nswap	1068	1109	1110	**12**	78s	18'756'636
	ea_16384_nswap	**1022**	**1063**	**1061**	16	168s	26'699'633

So what do we get?



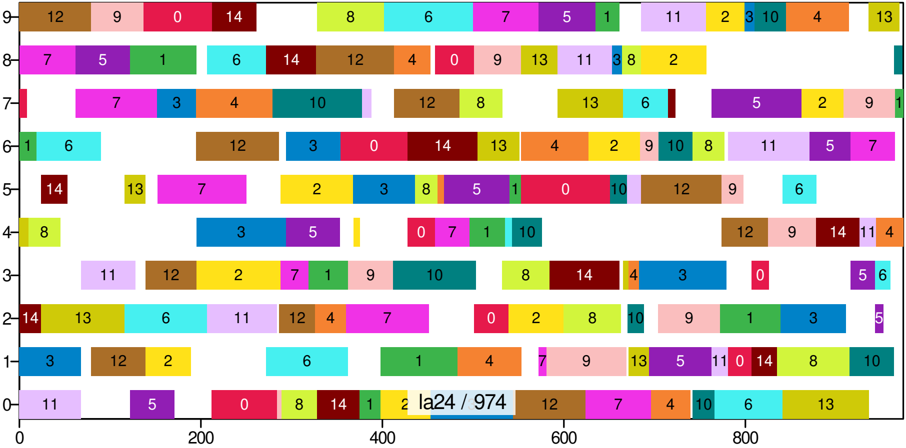
So what do we get?

ea_16384_nswap: median result of 3 min of the EA with $\mu = \lambda = 16'384$ with nswap unary operator



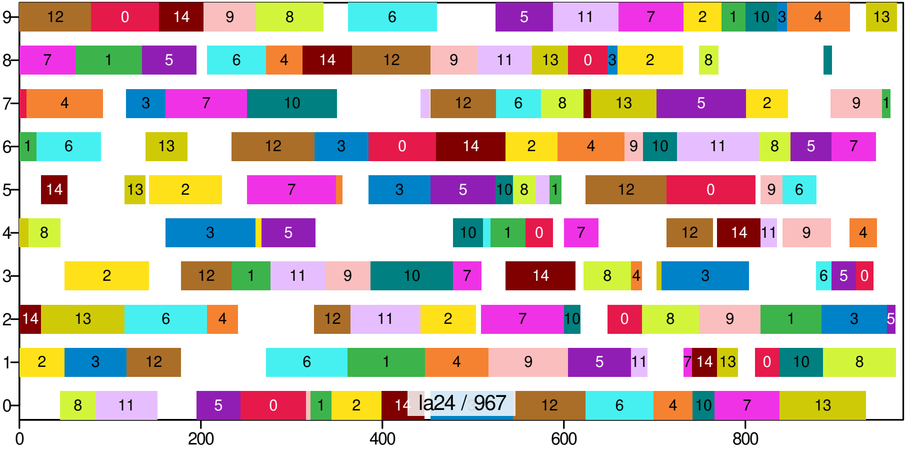
So what do we get?

`hcr_65536_nswap`: median result of 3 min of the restarted hill climber `hcr_65536_nswap` with $L = 65'536$ and `nswap`



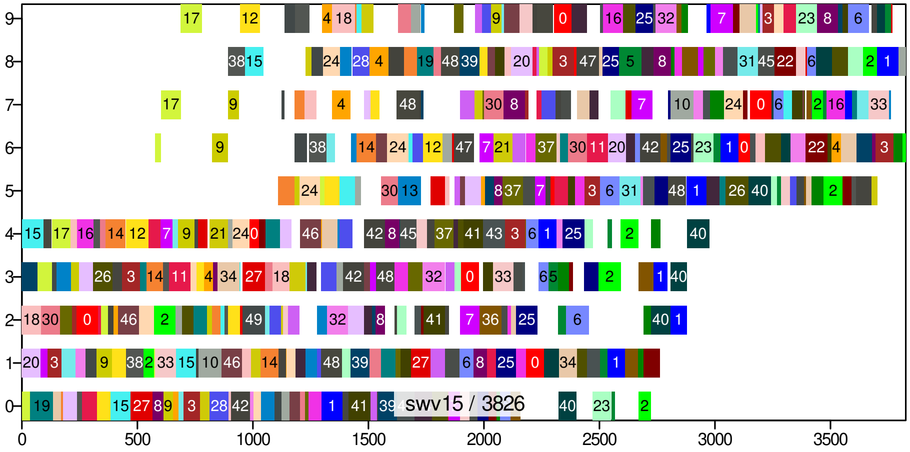
So what do we get?

`ea_16384_nswap`: median result of 3 min of the EA with $\mu = \lambda = 16'384$ with `nswap` unary operator



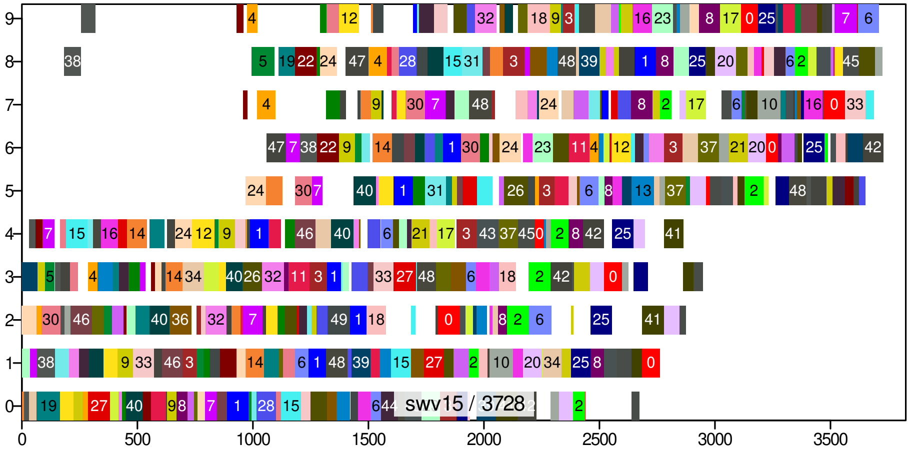
So what do we get?

`hcr_65536_nswap`: median result of 3 min of the restarted hill climber `hcr_65536_nswap` with $L = 65'536$ and `nswap`



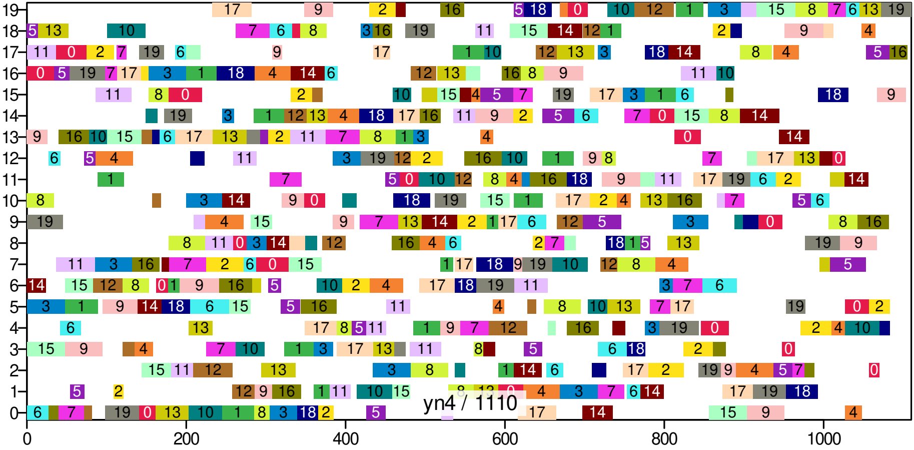
So what do we get?

ea_16384_nswap: median result of 3 min of the EA with $\mu = \lambda = 16'384$ with nswap unary operator

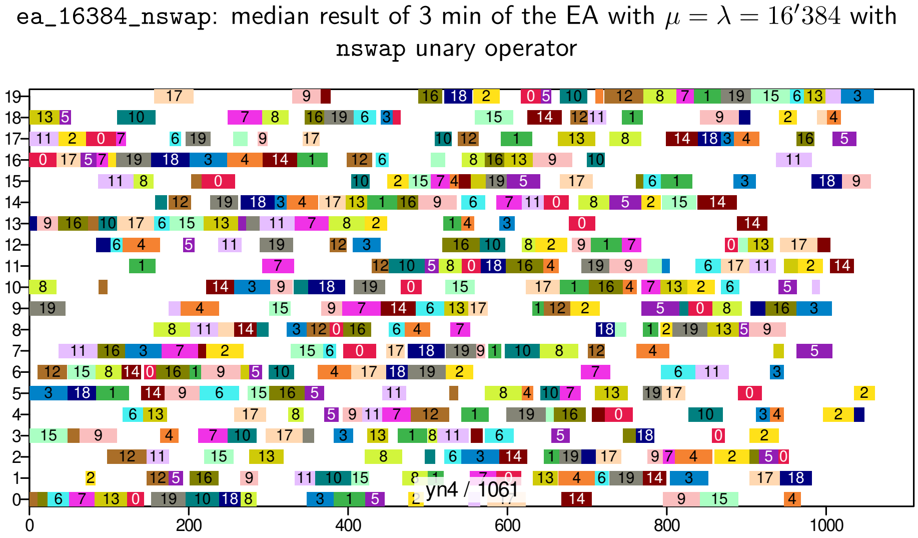


So what do we get?

`hcr_65536_nswap`: median result of 3 min of the restarted hill climber `hcr_65536_nswap` with $L = 65'536$ and `nswap`



So what do we get?

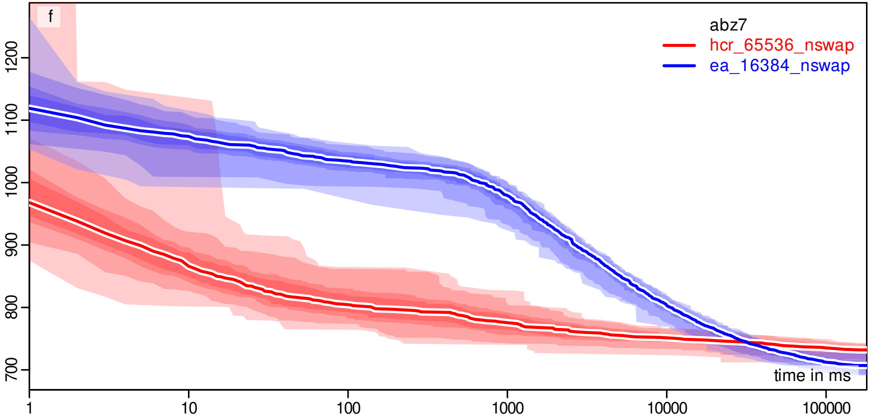


Progress over Time

What progress does the algorithm make over time?

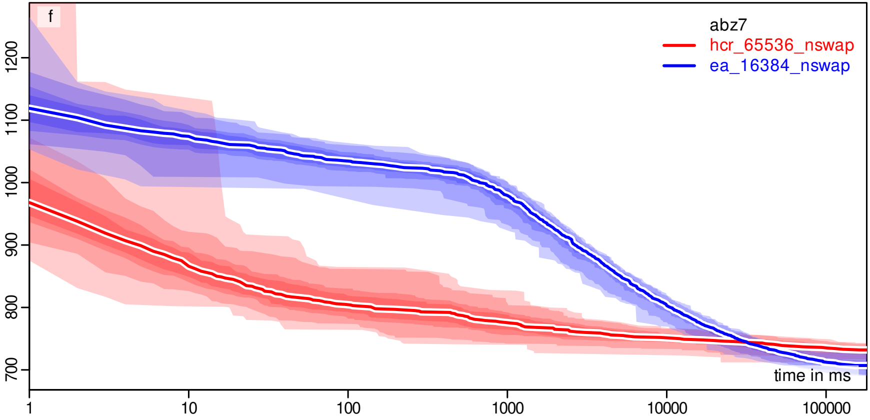
Progress over Time

What progress does the algorithm make over time?



Progress over Time

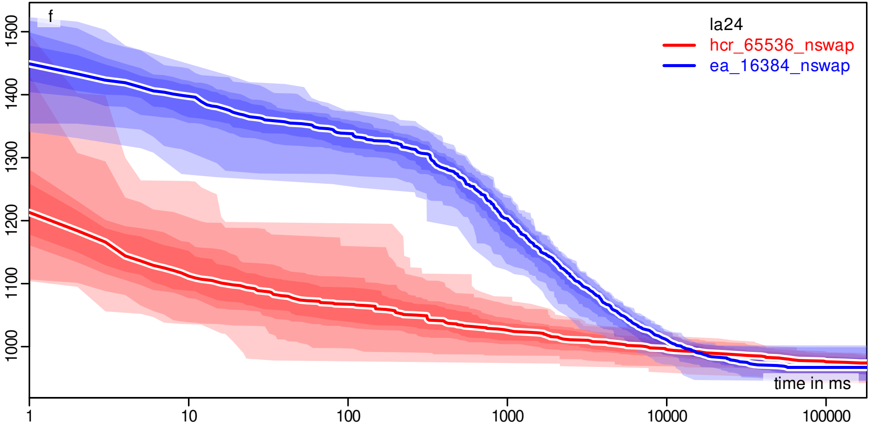
What progress does the algorithm make over time?



On the log-scale, it seems as if the EA first is much slower and very late in the search makes much progress.

Progress over Time

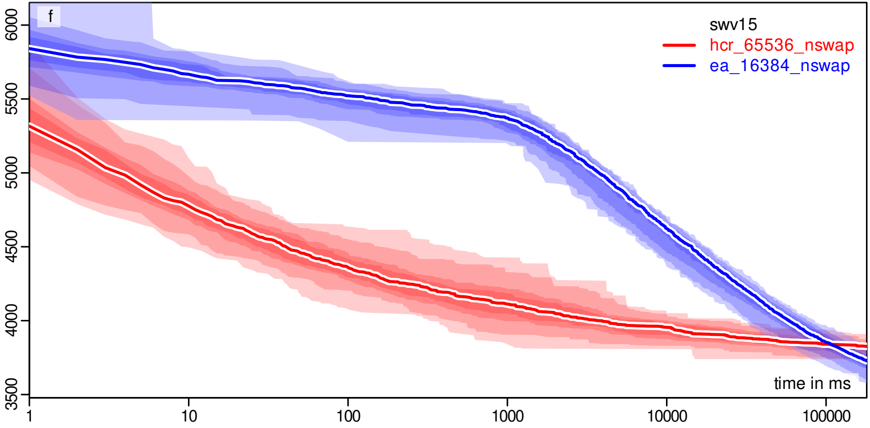
What progress does the algorithm make over time?



On the log-scale, it seems as if the EA first is much slower and very late in the search makes much progress.

Progress over Time

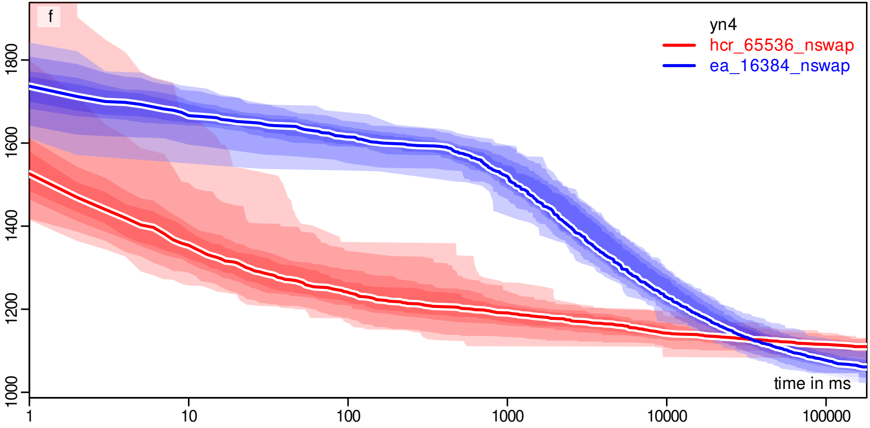
What progress does the algorithm make over time?



On the log-scale, it seems as if the EA first is much slower and very late in the search makes much progress.

Progress over Time

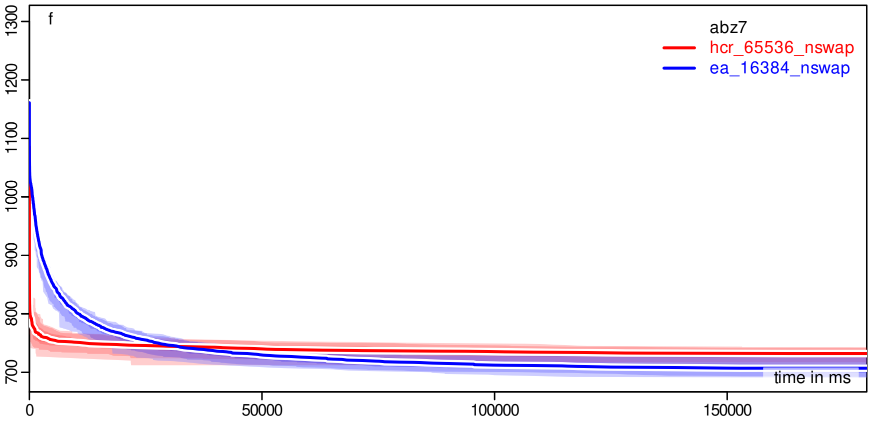
What progress does the algorithm make over time?



On the log-scale, it seems as if the EA first is much slower and very late in the search makes much progress.

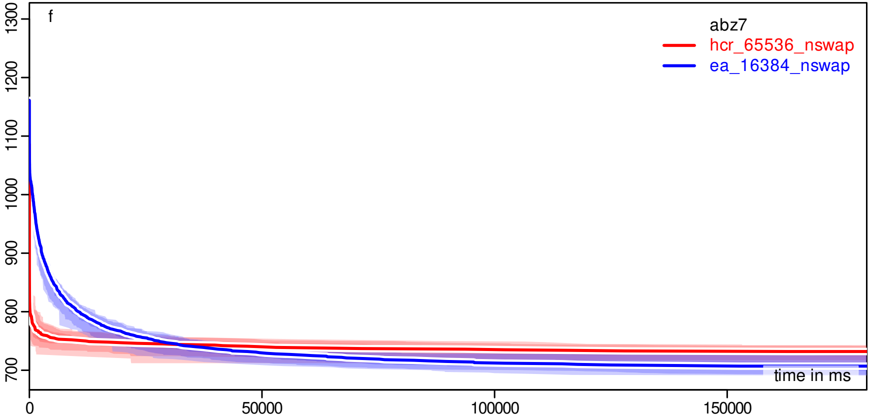
Progress over Time

What progress does the algorithm make over time?



Progress over Time

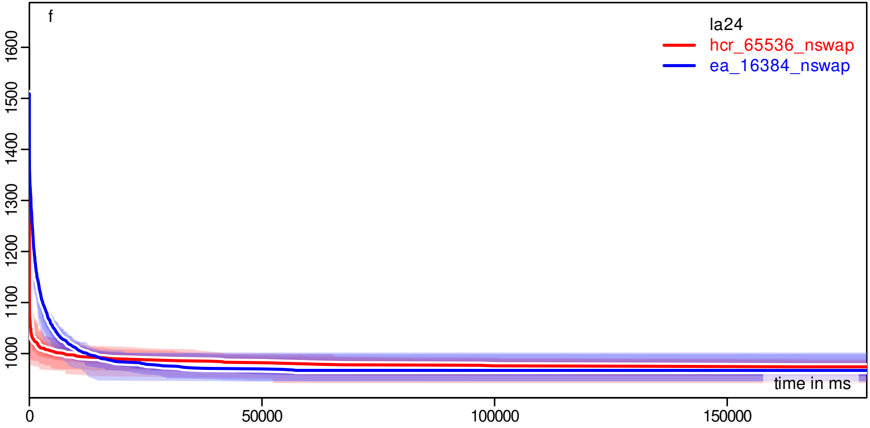
What progress does the algorithm make over time?



However, on the linear time scale we can see that it keeps improving slowly but surely during all the time.

Progress over Time

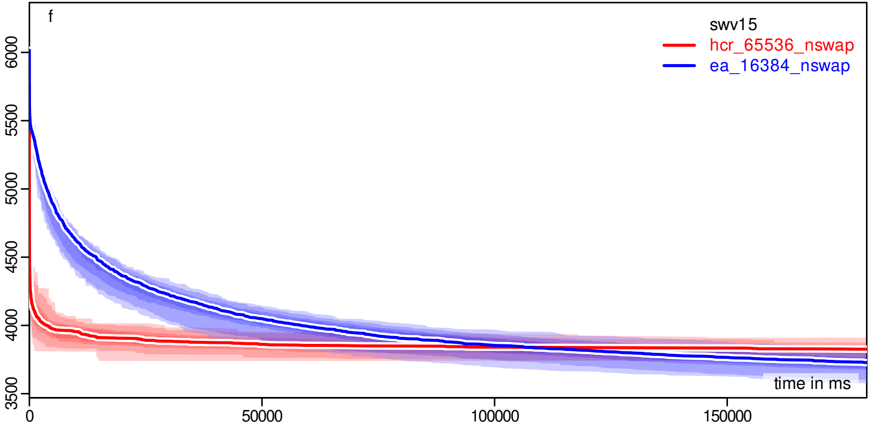
What progress does the algorithm make over time?



However, on the linear time scale we can see that it keeps improving slowly but surely during all the time.

Progress over Time

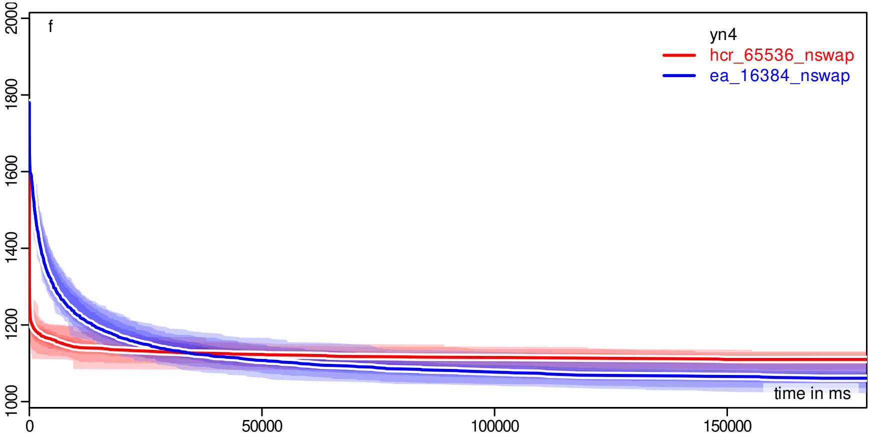
What progress does the algorithm make over time?



However, on the linear time scale we can see that it keeps improving slowly but surely during all the time.

Progress over Time

What progress does the algorithm make over time?



However, on the linear time scale we can see that it keeps improving slowly but surely during all the time.

Relationship EA / Hill Climber / Random Sampling

- We can imagine this first version of the $(\mu + \lambda)$ EA as a generalized version of a hill climber.

Relationship EA / Hill Climber / Random Sampling

- We can imagine this first version of the $(\mu + \lambda)$ EA as a generalized version of a hill climber.
- Or the other way around: A hill climber is a $(1 + 1)$ EA

Relationship EA / Hill Climber / Random Sampling

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

Relationship EA / Hill Climber / Random Sampling

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

Relationship EA / Hill Climber / Random Sampling

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

Relationship EA / Hill Climber / Random Sampling

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

Relationship EA / Hill Climber / Random Sampling

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

Relationship EA / Hill Climber / Random Sampling

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

Relationship EA / Hill Climber / Random Sampling

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

Exploration versus Exploitation

- **We can regard this basic EA as a way to choose an algorithm behavior in between random sampling and hill climbing!**

Exploration versus Exploitation

- **We can regard this basic EA as a way to choose an algorithm behavior in between random sampling and hill climbing!**
- The parameter μ basically allows us to “tune” between these two behaviors[7]

Exploration versus Exploitation

- **We can regard this basic EA as a way to choose an algorithm behavior in between random sampling and hill climbing!**
- The parameter μ basically allows us to "tune" between these two behaviors[7]
- If we pick it small, our algorithm becomes more "greedy".

Exploration versus Exploitation

- **We can regard this basic EA as a way to choose an algorithm behavior in between random sampling and hill climbing!**
- The parameter μ basically allows us to "tune" between these two behaviors[7]
- If we pick it small, our algorithm becomes more "greedy".
- It will investigate (exploit) the neighborhood current best solutions more eagerly.

Exploration versus Exploitation

- **We can regard this basic EA as a way to choose an algorithm behavior in between random sampling and hill climbing!**
- The parameter μ basically allows us to "tune" between these two behaviors[7]
- If we pick it small, our algorithm becomes more "greedy".
- It will investigate (exploit) the neighborhood current best solutions more eagerly, which means that it will trace down local optima faster.

Exploration versus Exploitation

- **We can regard this basic EA as a way to choose an algorithm behavior in between random sampling and hill climbing!**
- The parameter μ basically allows us to "tune" between these two behaviors[7]
- If we pick it small, our algorithm becomes more "greedy".
- It will investigate (exploit) the neighborhood current best solutions more eagerly, which means that it will trace down local optima faster but be trapped more easily in local optima as well.

Exploration versus Exploitation

- **We can regard this basic EA as a way to choose an algorithm behavior in between random sampling and hill climbing!**
- The parameter μ basically allows us to "tune" between these two behaviors[7]
- If we pick it small, our algorithm becomes more "greedy".
- It will investigate (exploit) the neighborhood current best solutions more eagerly, which means that it will trace down local optima faster but be trapped more easily in local optima as well.
- The bigger μ, the more points in the search space are maintained and the more likely are we do to good "global" search, we do more exploration.

Exploration versus Exploitation

- **We can regard this basic EA as a way to choose an algorithm behavior in between random sampling and hill climbing!**
- The parameter μ basically allows us to "tune" between these two behaviors[7]
- If we pick it small, our algorithm becomes more "greedy".
- It will investigate (exploit) the neighborhood current best solutions more eagerly, which means that it will trace down local optima faster but be trapped more easily in local optima as well.
- The bigger μ, the more points in the search space are maintained and the more likely are we do to good "global" search, we do more exploration. We pay for that by a slower exploitation (investigation) of the current best solution (because we always work on all μ points, not just one).

Exploration versus Exploitation

- **We can regard this basic EA as a way to choose an algorithm behavior in between random sampling and hill climbing!**
- The parameter μ basically allows us to “tune” between these two behaviors[7]
- If we pick it small, our algorithm becomes more “greedy”.
- It will investigate (exploit) the neighborhood current best solutions more eagerly, which means that it will trace down local optima faster but be trapped more easily in local optima as well.
- The bigger μ, the more points in the search space are maintained and the more likely are we do to good “global” search, we do more exploration. We pay for that by a slower exploitation (investigation) of the current best solution (because we always work on all μ points, not just one).
- This is dilemma of Exploration versus Exploitation.[2 8–10]

Algorithm Concept: Binary Operator

Binary Search Operator

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

Binary Search Operator

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

Binary Search Operator

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

Binary Search Operator

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

Binary Search Operator

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

Binary Search Operator

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

Binary Search Operator

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

Binary Search Operator

- We now have more than one candidate solution in our "population."
- But we only use one existing point from $\mathbb{X}$ as "blueprint" when we create a new one.
- Why can't we use *two* instead?
  - If two candidate solutions have been selected, they are probably good.
  - If two different candidate solutions are good, they may have different positive characteristics.
  - Let's try to create a new "offspring" solution which inherits characteristics from both "parents".
  - It could maybe inherit the positive traits and combine them...
- This is the idea of the crossover or recombination operator in Evolutionary Algorithms.[2 3 6]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Implementation

```java
package aitoa.algorithms;

public class EA<X, Y> extends Metaheuristic2<X, Y> {
// abridged code: unnecessary stuff omitted here and in function solve...
  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

    for (;;) { // main loop: one iteration = one generation
      Arrays.sort(P, Record.BY_QUALITY); // sort the population: mu best at front
      RandomUtils.shuffle(random, P, 0, this.mu); // shuffle parents for fairness
      int p1 = -1; // index to iterate over first parent
      for (int index = P.length; (--index) >= this.mu;) { // overwrite lambda worst
        if (process.shouldTerminate()) return;
        Record<X> dest = P[index];
        p1 = (p1 + 1) % this.mu; // step the parent 1 index
        Record<X> sel  = P[p1];
        this.unary.apply(sel.x, dest.x, random); // generate offspring
        dest.quality = process.evaluate(dest.x); // evaluate offspring
      } // the end of the offspring generation
    } // the end of the main loop
  } // end solve
} // end class
```

Implementation

```java
package aitoa.algorithms;

public class EA<X, Y> extends Metaheuristic2<X, Y> {
// abridged code: unnecessary stuff omitted here and in function solve...
  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

    for (;;) { // main loop: one iteration = one generation
      Arrays.sort(P, Record.BY_QUALITY); // sort the population: mu best at front
      RandomUtils.shuffle(random, P, 0, this.mu); // shuffle parents for fairness
      int p1 = -1; // index to iterate over first parent
      for (int index = P.length; (--index) >= this.mu;) { // overwrite lambda worst
        if (process.shouldTerminate()) return;
        Record<X> dest = P[index];
        p1 = (p1 + 1) % this.mu; // step the parent 1 index
        Record<X> sel  = P[p1];
//
//
//
//
//
//
        this.unary.apply(sel.x, dest.x, random); // generate offspring via unary
        dest.quality = process.evaluate(dest.x); // evaluate offspring
      } // the end of the offspring generation
    } // the end of the main loop
  } // end solve
} // end class
```

Implementation

```java
package aitoa.algorithms;

public class EA<X, Y> extends Metaheuristic2<X, Y> {
// abridged code: unnecessary stuff omitted here and in function solve...
  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

    for (;;) { // main loop: one iteration = one generation
      Arrays.sort(P, Record.BY_QUALITY); // sort the population: mu best at front
      RandomUtils.shuffle(random, P, 0, this.mu); // shuffle parents for fairness
      int p1 = -1; // index to iterate over first parent
      for (int index = P.length; (--index) >= this.mu;) { // overwrite lambda worst
        if (process.shouldTerminate()) return;
        Record<X> dest = P[index];
        p1 = (p1 + 1) % this.mu; // step the parent 1 index
        Record<X> sel  = P[p1];
        if (random.nextDouble() <= this.cr) { // crossover!
//
//
//
//
//
        } else this.unary.apply(sel.x, dest.x, random); // generate offspring via unary
        dest.quality = process.evaluate(dest.x); // evaluate offspring
      } // the end of the offspring generation
    } // the end of the main loop
  } // end solve
} // end class
```

Implementation

```java
package aitoa.algorithms;

public class EA<X, Y> extends Metaheuristic2<X, Y> {
// abridged code: unnecessary stuff omitted here and in function solve...
  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

    for (;;) { // main loop: one iteration = one generation
      Arrays.sort(P, Record.BY_QUALITY); // sort the population: mu best at front
      RandomUtils.shuffle(random, P, 0, this.mu); // shuffle parents for fairness
      int p1 = -1; // index to iterate over first parent
      for (int index = P.length; (--index) >= this.mu;) { // overwrite lambda worst
        if (process.shouldTerminate()) return;
        Record<X> dest = P[index];
        p1 = (p1 + 1) % this.mu; // step the parent 1 index
        Record<X> sel  = P[p1];
        if (random.nextDouble() <= this.cr) { // crossover!
          int p2;
          do { // find a second, different record
            p2 = random.nextInt(this.mu);
          } while (p2 == p1); // repeat until p1 != p2
//
        } else this.unary.apply(sel.x, dest.x, random); // generate offspring via unary
        dest.quality = process.evaluate(dest.x); // evaluate offspring
      } // the end of the offspring generation
    } // the end of the main loop
  } // end solve
} // end class
```

Implementation

```java
package aitoa.algorithms;

public class EA<X, Y> extends Metaheuristic2<X, Y> {
// abridged code: unnecessary stuff omitted here and in function solve...
  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

    for (;;) { // main loop: one iteration = one generation
      Arrays.sort(P, Record.BY_QUALITY); // sort the population: mu best at front
      RandomUtils.shuffle(random, P, 0, this.mu); // shuffle parents for fairness
      int p1 = -1; // index to iterate over first parent
      for (int index = P.length; (--index) >= this.mu;) { // overwrite lambda worst
        if (process.shouldTerminate()) return;
        Record<X> dest = P[index];
        p1 = (p1 + 1) % this.mu; // step the parent 1 index
        Record<X> sel  = P[p1];
        if (random.nextDouble() <= this.cr) { // crossover!
          int p2;
          do { // find a second, different record
            p2 = random.nextInt(this.mu);
          } while (p2 == p1); // repeat until p1 != p2
          this.binary.apply(sel.x, P[p2].x, dest.x, random); // perform recombination
        } else this.unary.apply(sel.x, dest.x, random); // generate offspring via unary
        dest.quality = process.evaluate(dest.x); // evaluate offspring
      } // the end of the offspring generation
    } // the end of the main loop
  } // end solve
} // end class
```

Recombination for our Representation: One Possible Idea

1. Data structure x' be the destination to hold the new point in the search space that we want to sample.

Recombination for our Representation: One Possible Idea

1. Data structure x' be the destination to hold the new point in the search space that we want to sample.
2. Set the index i where the next operation should be stored in x' to $i = 0$.

Recombination for our Representation: One Possible Idea

1. Data structure x' be the destination to hold the new point in the search space that we want to sample.
2. Set the index i where the next operation should be stored in x' to $i = 0$.
3. Repeat

Recombination for our Representation: One Possible Idea

1. Data structure x' be the destination to hold the new point in the search space that we want to sample.
2. Set the index i where the next operation should be stored in x' to $i = 0$.
3. Repeat
   3.1 Randomly choose one of the input points $x1$ or $x2$ with equal probability as source x.

Recombination for our Representation: One Possible Idea

1. Data structure x' be the destination to hold the new point in the search space that we want to sample.
2. Set the index i where the next operation should be stored in x' to $i = 0$.
3. Repeat
   3.1 Randomly choose one of the input points $x1$ or $x2$ with equal probability as source x.
   3.2 Select the first (at the lowest index) operation in x that is not marked yet and store it in variable J.

Recombination for our Representation: One Possible Idea

1. Data structure x' be the destination to hold the new point in the search space that we want to sample.
2. Set the index i where the next operation should be stored in x' to $i = 0$.
3. Repeat
   3.1 Randomly choose one of the input points $x1$ or $x2$ with equal probability as source x.
   3.2 Select the first (at the lowest index) operation in x that is not marked yet and store it in variable J.
   3.3 Set $x'_i = J$.

Recombination for our Representation: One Possible Idea

1. Data structure x' be the destination to hold the new point in the search space that we want to sample.
2. Set the index i where the next operation should be stored in x' to $i = 0$.
3. Repeat
   3.1 Randomly choose one of the input points $x1$ or $x2$ with equal probability as source x.
   3.2 Select the first (at the lowest index) operation in x that is not marked yet and store it in variable J.
   3.3 Set $x'_i = J$.
   3.4 Increase i by one ($i = i + 1$).

Recombination for our Representation: One Possible Idea

1. Data structure x' be the destination to hold the new point in the search space that we want to sample.
2. Set the index i where the next operation should be stored in x' to $i = 0$.
3. Repeat
   3.1 Randomly choose one of the input points $x1$ or $x2$ with equal probability as source x.
   3.2 Select the first (at the lowest index) operation in x that is not marked yet and store it in variable J.
   3.3 Set $x'_i = J$.
   3.4 Increase i by one ($i = i + 1$).
   3.5 If $i = n * m$, then all operations have been assigned. We exit and returning x'.

Recombination for our Representation: One Possible Idea

1. Data structure x' be the destination to hold the new point in the search space that we want to sample.
2. Set the index i where the next operation should be stored in x' to $i = 0$.
3. Repeat
   3.1 Randomly choose one of the input points $x1$ or $x2$ with equal probability as source x.
   3.2 Select the first (at the lowest index) operation in x that is not marked yet and store it in variable J.
   3.3 Set $x'_i = J$.
   3.4 Increase i by one ($i = i + 1$).
   3.5 If $i = n * m$, then all operations have been assigned. We exit and returning x'.
   3.6 Mark the first unmarked occurrence of J as "already assigned" in $x1$.

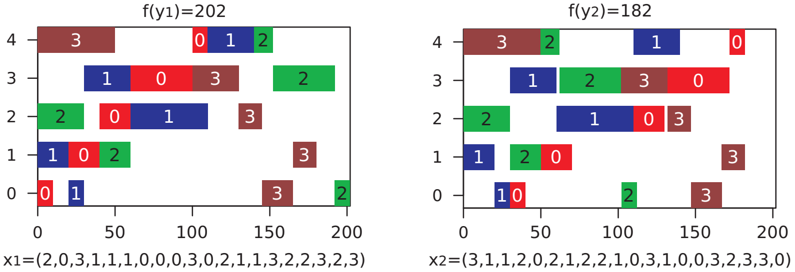
Recombination for our Representation: One Possible Idea

1. Data structure x' be the destination to hold the new point in the search space that we want to sample.
2. Set the index i where the next operation should be stored in x' to $i = 0$.
3. Repeat
   - 3.1 Randomly choose one of the input points $x1$ or $x2$ with equal probability as source x.
   - 3.2 Select the first (at the lowest index) operation in x that is not marked yet and store it in variable J.
   - 3.3 Set $x'_i = J$.
   - 3.4 Increase i by one ($i = i + 1$).
   - 3.5 If $i = n * m$, then all operations have been assigned. We exit and returning x'.
   - 3.6 Mark the first unmarked occurrence of J as "already assigned" in $x1$.
   - 3.7 Mark the first unmarked occurrence of J as "already assigned" in $x2$.

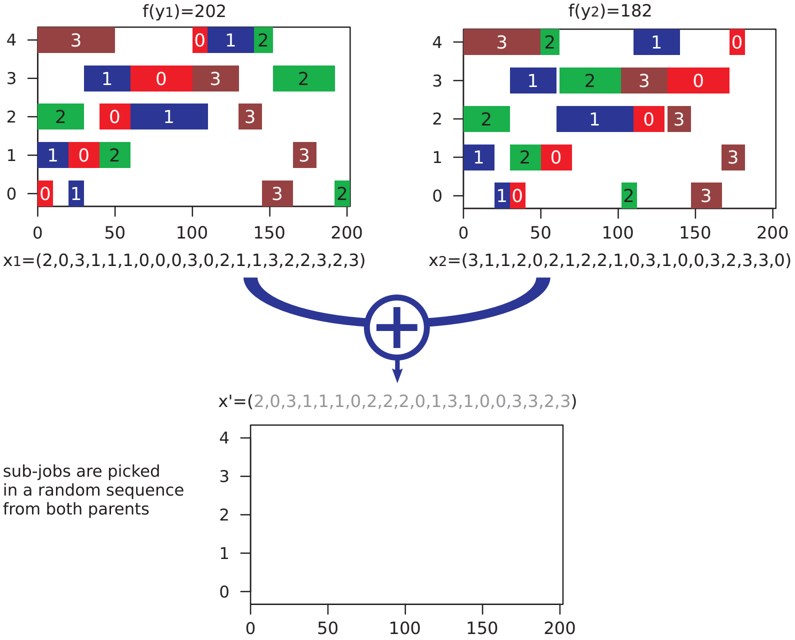
Example for Sequence Recombination

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

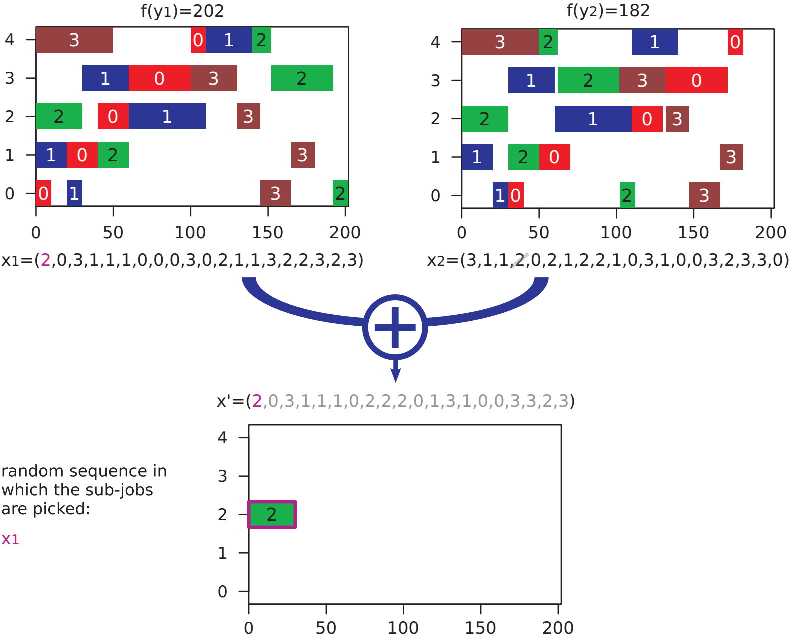
Example for Sequence Recombination



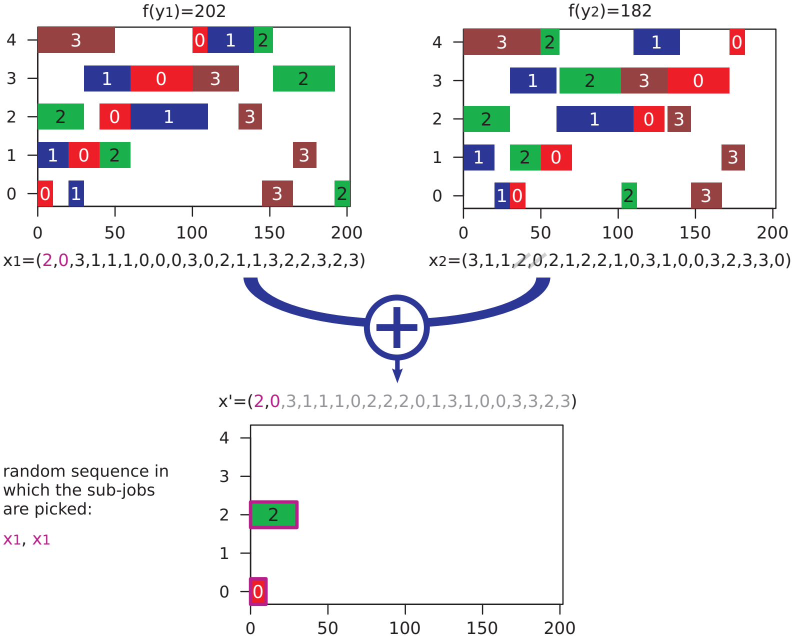
Example for Sequence Recombination



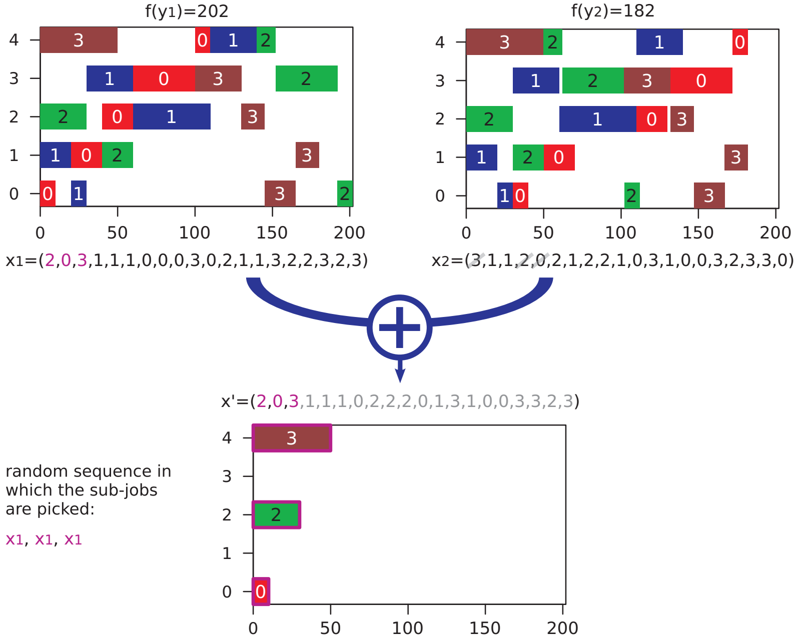
Example for Sequence Recombination



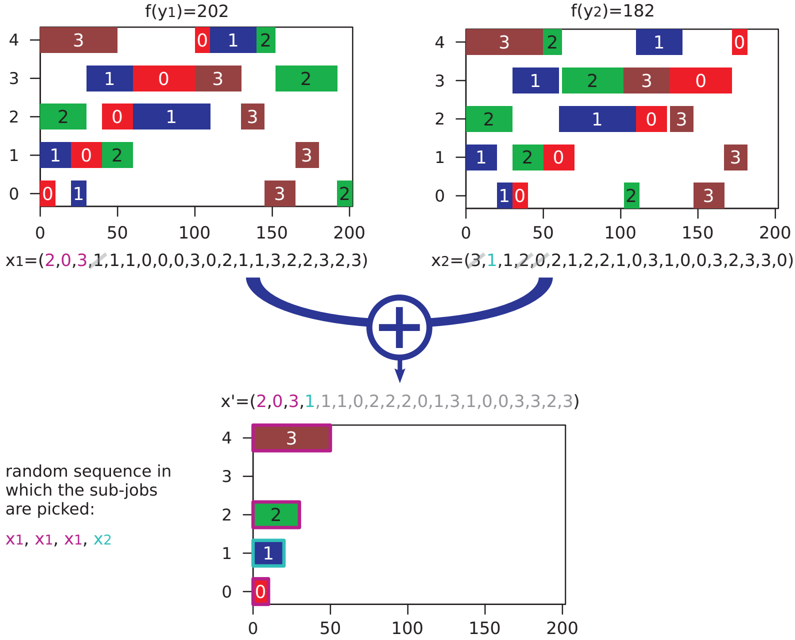
Example for Sequence Recombination



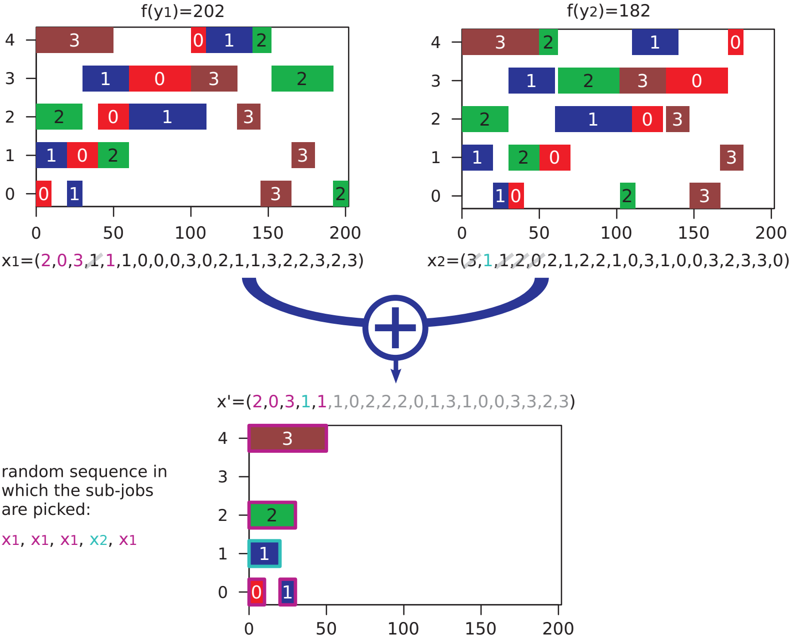
Example for Sequence Recombination



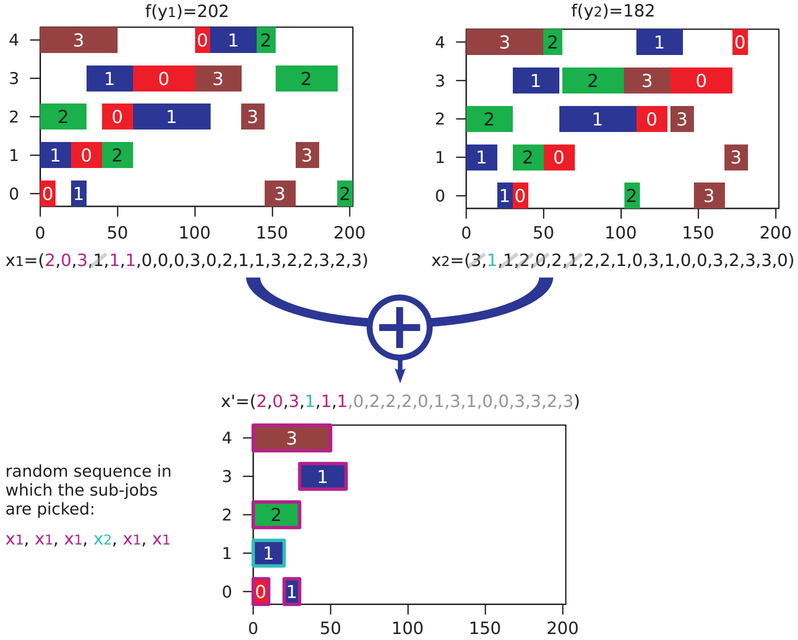
Example for Sequence Recombination



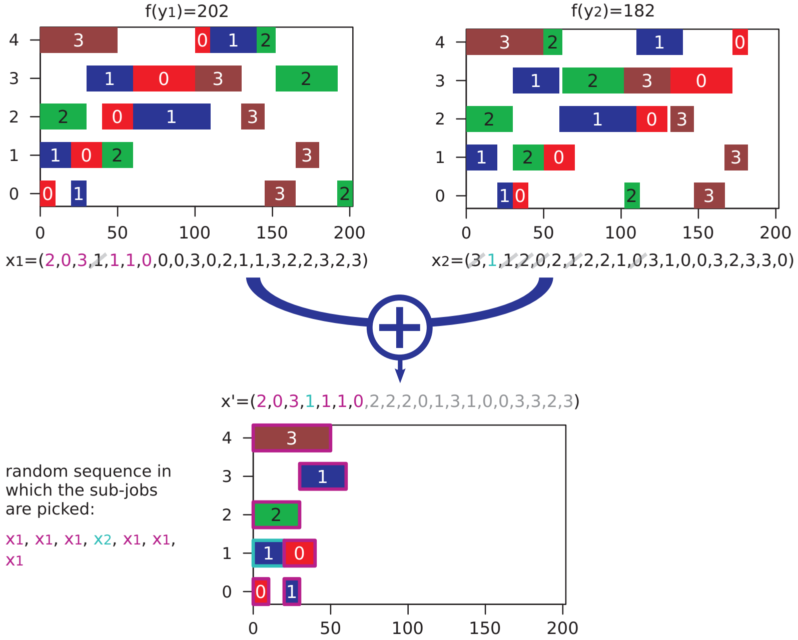
Example for Sequence Recombination



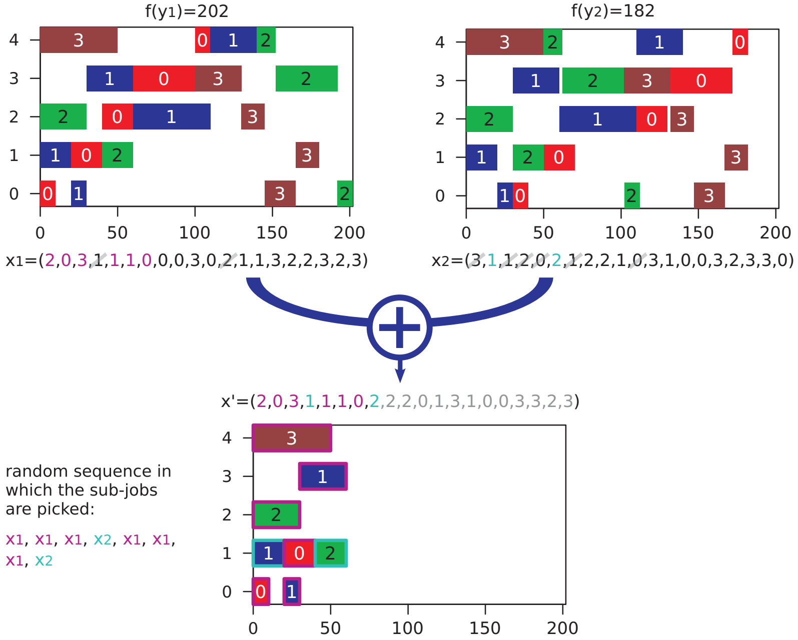
Example for Sequence Recombination



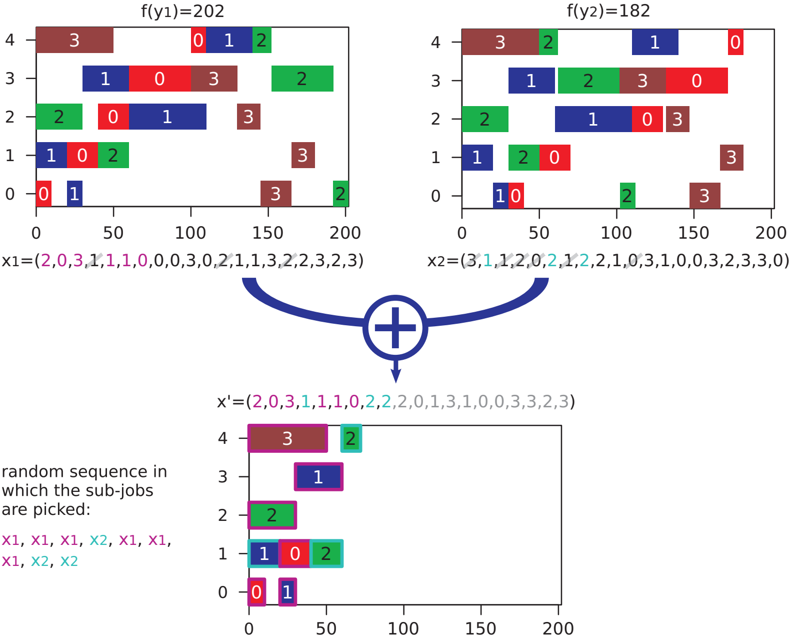
Example for Sequence Recombination



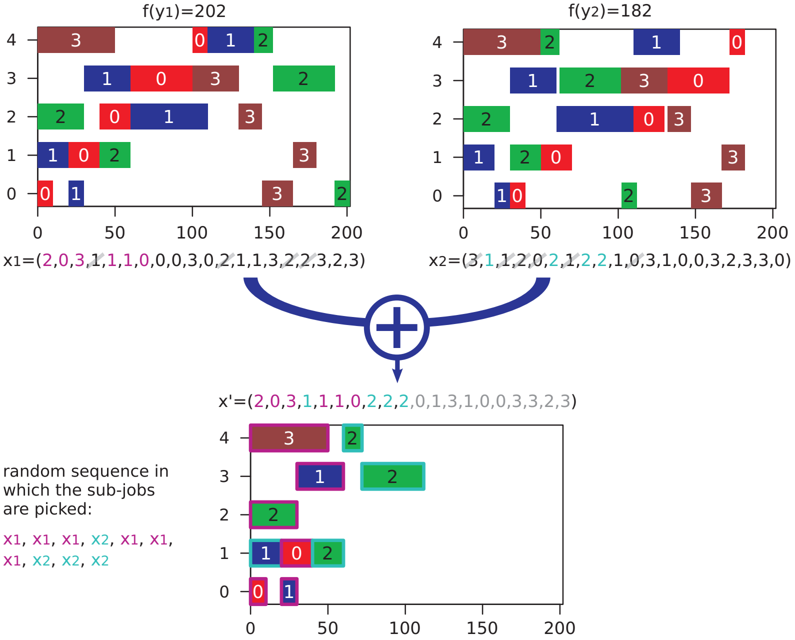
Example for Sequence Recombination



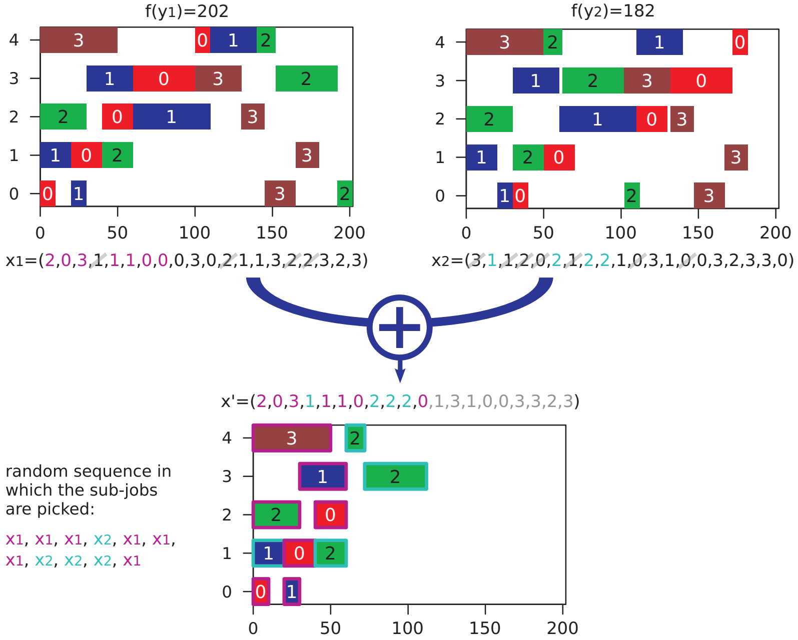
Example for Sequence Recombination



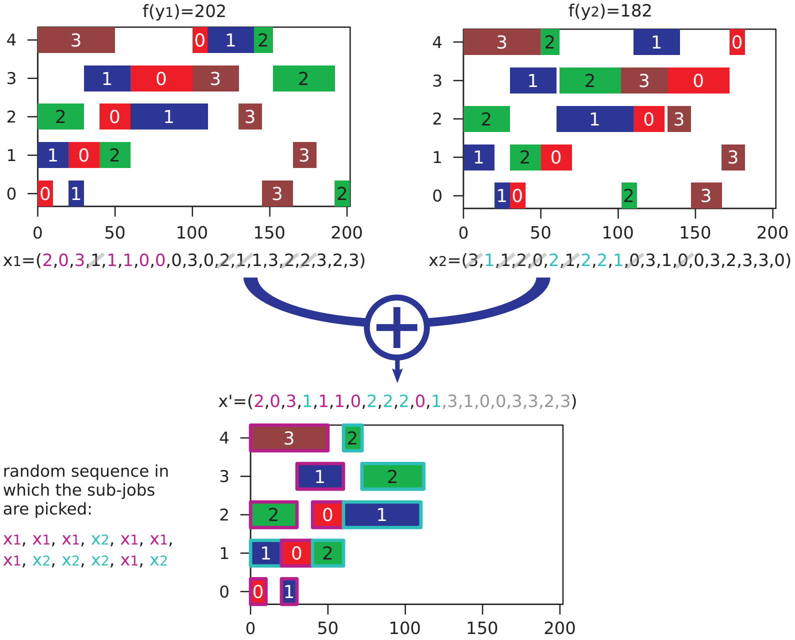
Example for Sequence Recombination



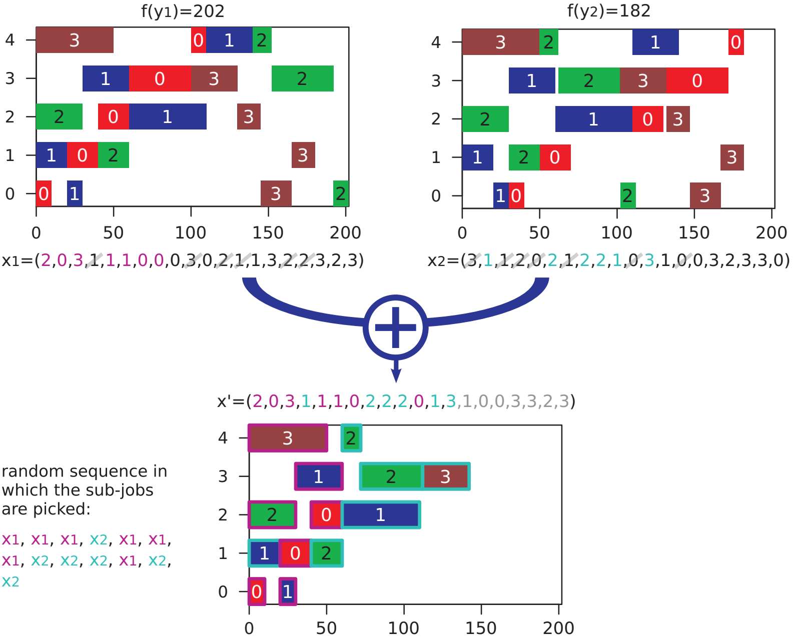
Example for Sequence Recombination



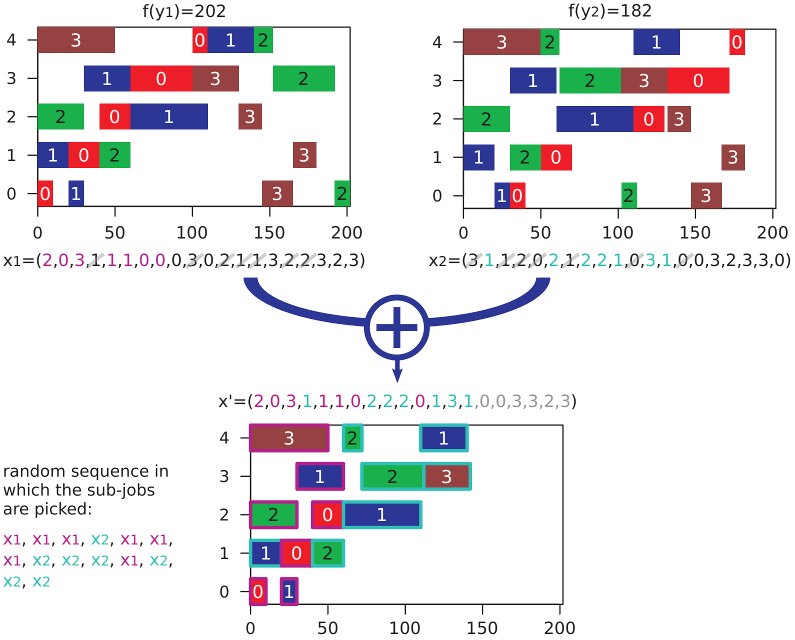
Example for Sequence Recombination



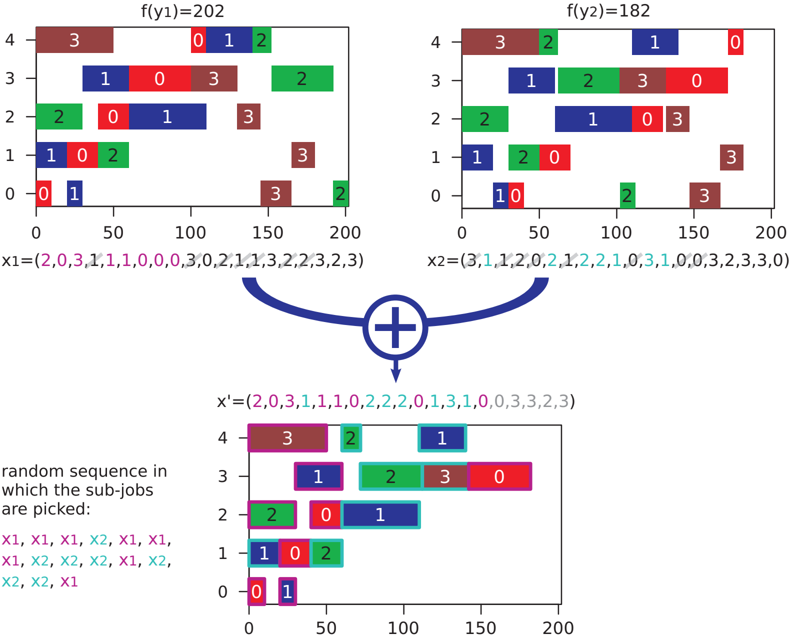
Example for Sequence Recombination



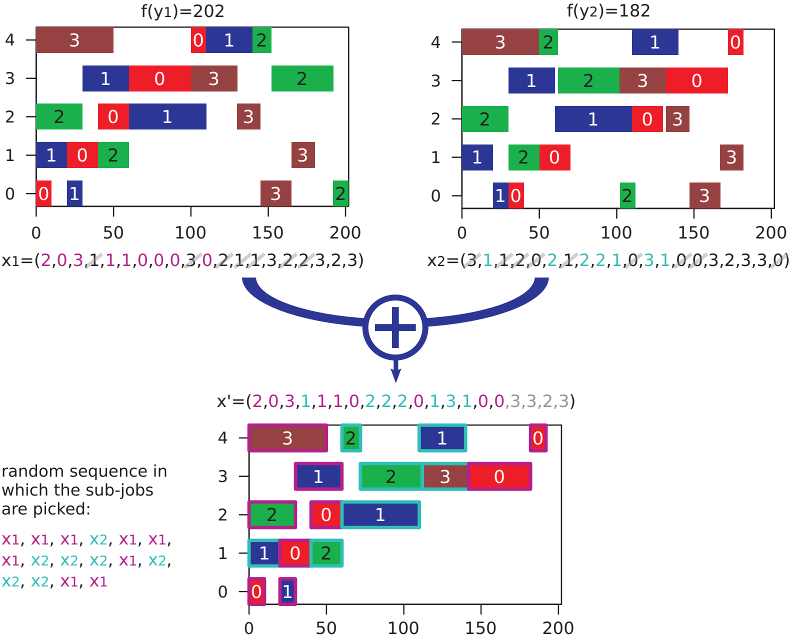
Example for Sequence Recombination



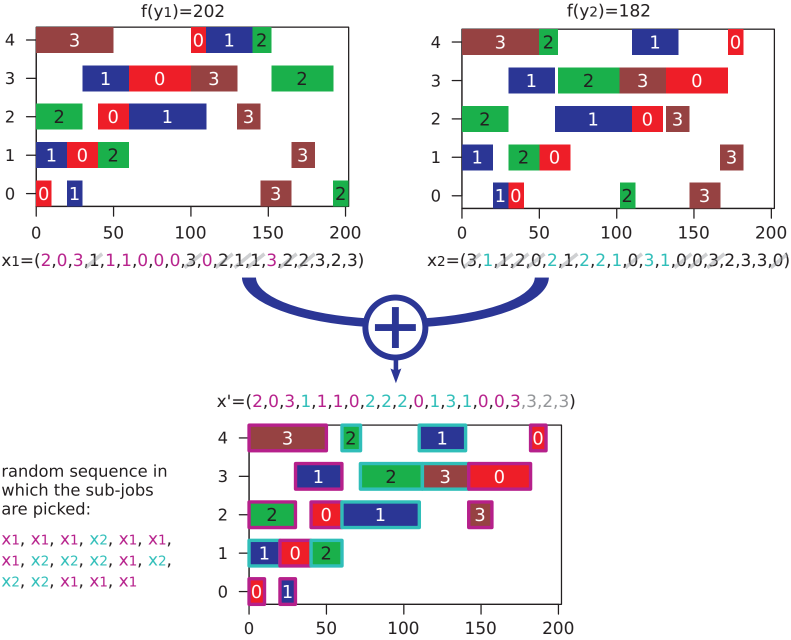
Example for Sequence Recombination



Example for Sequence Recombination



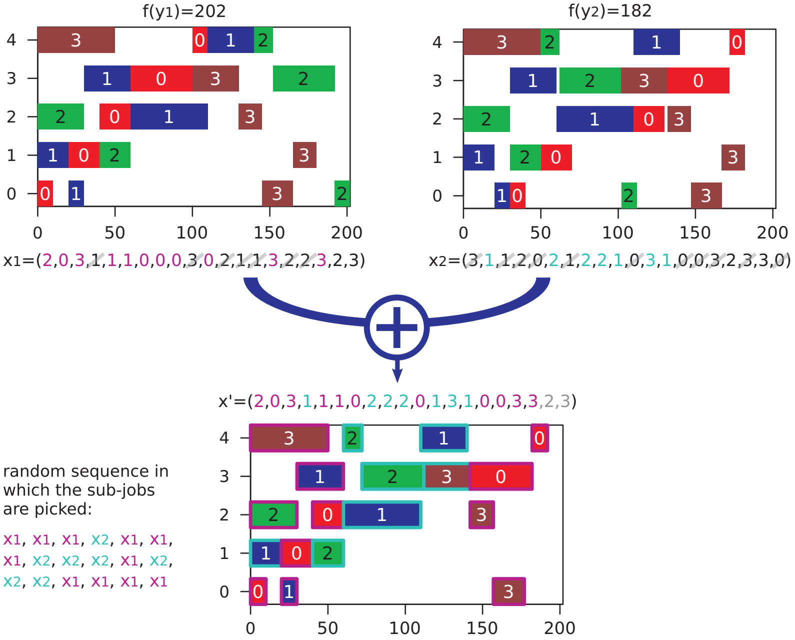
Example for Sequence Recombination



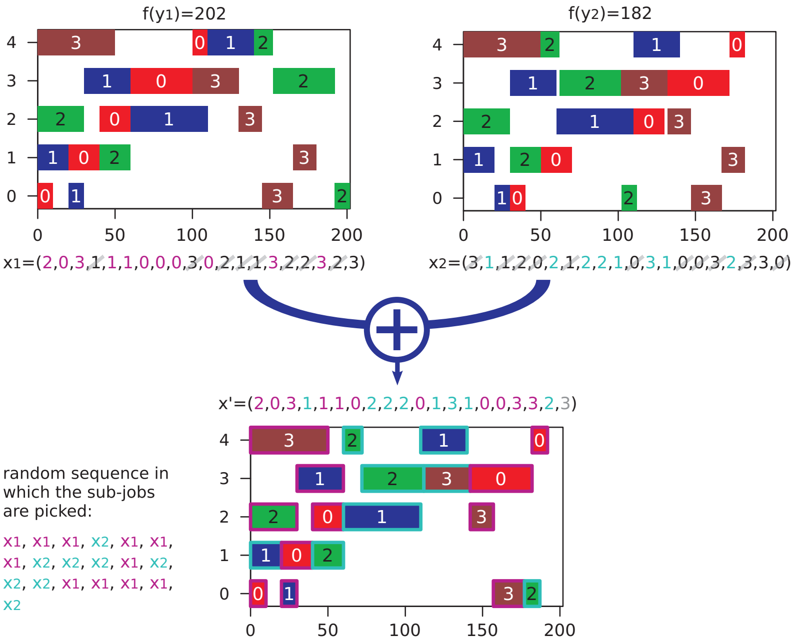
random sequence in which the sub-jobs are picked:

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

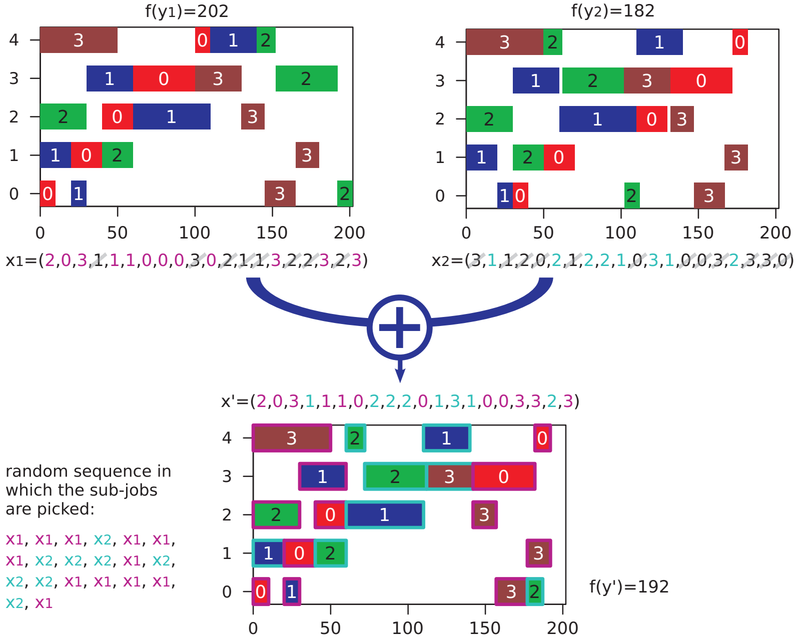
Example for Sequence Recombination



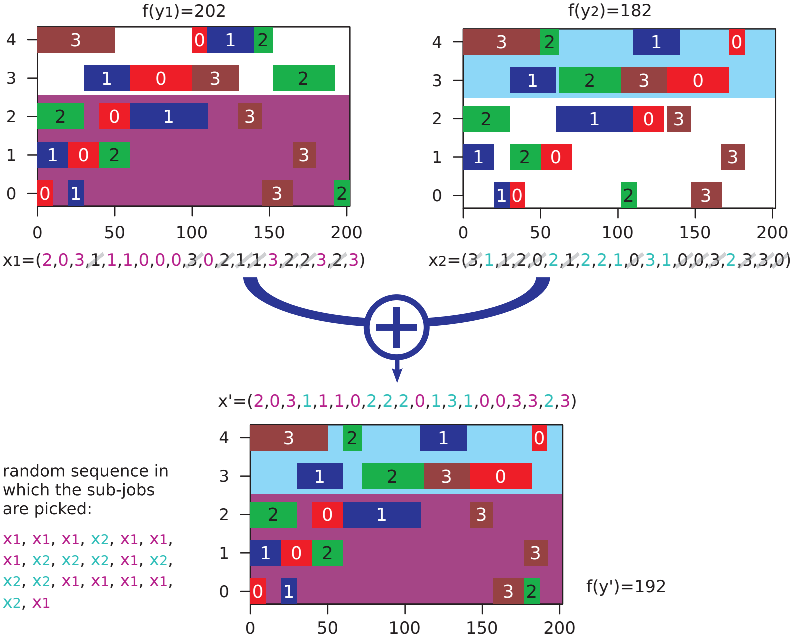
Example for Sequence Recombination



Example for Sequence Recombination



Example for Sequence Recombination



Implementing Sequence Recombination

```
package aitoa.examples.jssp;

public class JSSPBinaryOperatorSequence {
//
//
//

//
//
//
//

//
//
//
//
//

//
//
//
//
//
//
//

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

Implementing Sequence Recombination

```
package aitoa.examples.jssp;

public class JSSPBinaryOperatorSequence implements IBinarySearchOperator<int[]> {
//
//
//

//
//
//
//

//
//
//
//
//

//
//
//
//
//
//
//

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

Implementing Sequence Recombination

```
package aitoa.examples.jssp;

public class JSSPBinaryOperatorSequence implements IBinarySearchOperator<int[]> {
  public void apply(int[] x0, int[] x1, int[] dest, Random random) {
//
//

//
//
//
//

//
//
//
//
//

//
//
//
//
//
//
//

//
//
//
//
//
//
//
//
  } // end of function
}
```

Implementing Sequence Recombination

```
package aitoa.examples.jssp;

public class JSSPBinaryOperatorSequence implements IBinarySearchOperator<int[]> {
  public void apply(int[] x0, int[] x1, int[] dest, Random random) {
    boolean[] doneX0 = new boolean[x0.length];
//

//
//
//
//

//
//
//
//
//

//
//
//
//
//
//
//

//
//
//
//
//
//
//
//
  } // end of function
}
```

Implementing Sequence Recombination

```
package aitoa.examples.jssp;

public class JSSPBinaryOperatorSequence implements IBinarySearchOperator<int[]> {
  public void apply(int[] x0, int[] x1, int[] dest, Random random) {
    boolean[] doneX0 = new boolean[x0.length];
    boolean[] doneX1 = new boolean[x0.length];

//
//
//
//

//
//
//
//
//

//
//
//
//
//
//
//

//
//
//
//
//
//
//
//
  } // end of function
}
```

Implementing Sequence Recombination

```java
package aitoa.examples.jssp;

public class JSSPBinaryOperatorSequence implements IBinarySearchOperator<int[]> {
  public void apply(int[] x0, int[] x1, int[] dest, Random random) {
    boolean[] doneX0 = new boolean[x0.length]; // can be stored as reuseable
    boolean[] doneX1 = new boolean[x0.length]; // member variable instead

//
//
//
//

//
//
//
//
//

//
//
//
//
//
//
//

//
//
//
//
//
//
//
//
  } // end of function
}
```

Implementing Sequence Recombination

```
package aitoa.examples.jssp;

public class JSSPBinaryOperatorSequence implements IBinarySearchOperator<int[]> {
  public void apply(int[] x0, int[] x1, int[] dest, Random random) {
    boolean[] doneX0 = new boolean[x0.length]; // can be stored as reuseable
    boolean[] doneX1 = new boolean[x0.length]; // member variable instead

    int length = doneX0.length; // length = m*n
//
//
//

//
//
//
//
//

//
//
//
//
//
//
//

//
//
//
//
//
//
//
//
  } // end of function
}
```

Implementing Sequence Recombination

```
package aitoa.examples.jssp;

public class JSSPBinaryOperatorSequence implements IBinarySearchOperator<int[]> {
  public void apply(int[] x0, int[] x1, int[] dest, Random random) {
    boolean[] doneX0 = new boolean[x0.length]; // can be stored as reuseable
    boolean[] doneX1 = new boolean[x0.length]; // member variable instead

    int length = doneX0.length; // length = m*n
    int desti  = 0;             // all array indexes = 0
//
//

//
//
//
//
//

//
//
//
//
//
//
//

//
//
//
//
//
//
//
//
  } // end of function
}
```

Implementing Sequence Recombination

```java
package aitoa.examples.jssp;

public class JSSPBinaryOperatorSequence implements IBinarySearchOperator<int[]> {
  public void apply(int[] x0, int[] x1, int[] dest, Random random) {
    boolean[] doneX0 = new boolean[x0.length]; // can be stored as reuseable
    boolean[] doneX1 = new boolean[x0.length]; // member variable instead

    int length = doneX0.length; // length = m*n
    int desti  = 0;             // all array indexes = 0
    int x0i    = 0;             // index of first unfinished operation in x0
//

//
//
//
//
//

//
//
//
//
//
//
//

//
//
//
//
//
//
//
//
  } // end of function
}
```

Implementing Sequence Recombination

```
package aitoa.examples.jssp;

public class JSSPBinaryOperatorSequence implements IBinarySearchOperator<int[]> {
  public void apply(int[] x0, int[] x1, int[] dest, Random random) {
    boolean[] doneX0 = new boolean[x0.length]; // can be stored as reuseable
    boolean[] doneX1 = new boolean[x0.length]; // member variable instead

    int length = doneX0.length; // length = m*n
    int desti  = 0;             // all array indexes = 0
    int x0i    = 0;             // index of first unfinished operation in x0
    int x1i    = 0;             // index of first unfinished operation in x1

//
//
//
//
//

//
//
//
//
//
//
//

//
//
//
//
//
//
//
//
  } // end of function
}
```

Implementing Sequence Recombination

```java
package aitoa.examples.jssp;

public class JSSPBinaryOperatorSequence implements IBinarySearchOperator<int[]> {
  public void apply(int[] x0, int[] x1, int[] dest, Random random) {
    boolean[] doneX0 = new boolean[x0.length]; // can be stored as reuseable
    boolean[] doneX1 = new boolean[x0.length]; // member variable instead

    int length = doneX0.length; // length = m*n
    int desti  = 0;             // all array indexes = 0
    int x0i    = 0;             // index of first unfinished operation in x0
    int x1i    = 0;             // index of first unfinished operation in x1

//
// randomly chose a source point and pick next operation from it
    int add = random.nextBoolean() ? x0[x0i] : x1[x1i];
//
//

//
//
//
//
//
//
//

//
//
//
//
//
//
//
//
  } // end of function
}
```

Implementing Sequence Recombination

```java
package aitoa.examples.jssp;

public class JSSPBinaryOperatorSequence implements IBinarySearchOperator<int[]> {
  public void apply(int[] x0, int[] x1, int[] dest, Random random) {
    boolean[] doneX0 = new boolean[x0.length]; // can be stored as reuseable
    boolean[] doneX1 = new boolean[x0.length]; // member variable instead

    int length = doneX0.length; // length = m*n
    int desti  = 0;             // all array indexes = 0
    int x0i    = 0;             // index of first unfinished operation in x0
    int x1i    = 0;             // index of first unfinished operation in x1

//
// randomly chose a source point and pick next operation from it
    int add = random.nextBoolean() ? x0[x0i] : x1[x1i];
    dest[desti++] = add; // we picked a operation and added it
//

//
//
//
//
//
//
//

//
//
//
//
//
//
//
//
  } // end of function
}
```

Implementing Sequence Recombination

```
package aitoa.examples.jssp;

public class JSSPBinaryOperatorSequence implements IBinarySearchOperator<int[]> {
  public void apply(int[] x0, int[] x1, int[] dest, Random random) {
    boolean[] doneX0 = new boolean[x0.length]; // can be stored as reuseable
    boolean[] doneX1 = new boolean[x0.length]; // member variable instead

    int length = doneX0.length; // length = m*n
    int desti  = 0;             // all array indexes = 0
    int x0i    = 0;             // index of first unfinished operation in x0
    int x1i    = 0;             // index of first unfinished operation in x1

//
// randomly chose a source point and pick next operation from it
    int add = random.nextBoolean() ? x0[x0i] : x1[x1i];
    dest[desti++] = add; // we picked a operation and added it
//

    for (int i = x0i;; i++) { // mark the operation as done in x0
//
//
//
//
    }
//

//
//
//
//
//
//
//
//
  } // end of function
}
```

Implementing Sequence Recombination

```
package aitoa.examples.jssp;

public class JSSPBinaryOperatorSequence implements IBinarySearchOperator<int[]> {
  public void apply(int[] x0, int[] x1, int[] dest, Random random) {
    boolean[] doneX0 = new boolean[x0.length]; // can be stored as reuseable
    boolean[] doneX1 = new boolean[x0.length]; // member variable instead

    int length = doneX0.length; // length = m*n
    int desti  = 0;             // all array indexes = 0
    int x0i    = 0;             // index of first unfinished operation in x0
    int x1i    = 0;             // index of first unfinished operation in x1

//
// randomly chose a source point and pick next operation from it
    int add = random.nextBoolean() ? x0[x0i] : x1[x1i];
    dest[desti++] = add; // we picked a operation and added it
//

    for (int i = x0i;; i++) { // mark the operation as done in x0
      if ((x0[i] == add) && (!doneX0[i])) { // find added job
//
//
      }
    }
//

//
//
//
//
//
//
//
//
  } // end of function
}
```

Implementing Sequence Recombination

```
package aitoa.examples.jssp;

public class JSSPBinaryOperatorSequence implements IBinarySearchOperator<int[]> {
  public void apply(int[] x0, int[] x1, int[] dest, Random random) {
    boolean[] doneX0 = new boolean[x0.length]; // can be stored as reuseable
    boolean[] doneX1 = new boolean[x0.length]; // member variable instead

    int length = doneX0.length; // length = m*n
    int desti  = 0;             // all array indexes = 0
    int x0i    = 0;             // index of first unfinished operation in x0
    int x1i    = 0;             // index of first unfinished operation in x1

//
// randomly chose a source point and pick next operation from it
    int add = random.nextBoolean() ? x0[x0i] : x1[x1i];
    dest[desti++] = add; // we picked a operation and added it
//

    for (int i = x0i;; i++) { // mark the operation as done in x0
      if ((x0[i] == add) && (!doneX0[i])) { // find added job
        doneX0[i] = true; // found it and marked it
//
      }
    }
//

//
//
//
//
//
//
//
//
  } // end of function
}
```

Implementing Sequence Recombination

```java
package aitoa.examples.jssp;

public class JSSPBinaryOperatorSequence implements IBinarySearchOperator<int[]> {
  public void apply(int[] x0, int[] x1, int[] dest, Random random) {
    boolean[] doneX0 = new boolean[x0.length]; // can be stored as reuseable
    boolean[] doneX1 = new boolean[x0.length]; // member variable instead

    int length = doneX0.length; // length = m*n
    int desti  = 0;             // all array indexes = 0
    int x0i    = 0;             // index of first unfinished operation in x0
    int x1i    = 0;             // index of first unfinished operation in x1

//
// randomly chose a source point and pick next operation from it
    int add = random.nextBoolean() ? x0[x0i] : x1[x1i];
    dest[desti++] = add; // we picked a operation and added it
//

    for (int i = x0i;; i++) { // mark the operation as done in x0
      if ((x0[i] == add) && (!doneX0[i])) { // find added job
        doneX0[i] = true; // found it and marked it
        break; // quit operation finding loop
      }
    }
//

//
//
//
//
//
//
//
//
  } // end of function
}
```

Implementing Sequence Recombination

```
package aitoa.examples.jssp;

public class JSSPBinaryOperatorSequence implements IBinarySearchOperator<int[]> {
  public void apply(int[] x0, int[] x1, int[] dest, Random random) {
    boolean[] doneX0 = new boolean[x0.length]; // can be stored as reuseable
    boolean[] doneX1 = new boolean[x0.length]; // member variable instead

    int length = doneX0.length; // length = m*n
    int desti  = 0;             // all array indexes = 0
    int x0i    = 0;             // index of first unfinished operation in x0
    int x1i    = 0;             // index of first unfinished operation in x1

//
// randomly chose a source point and pick next operation from it
    int add = random.nextBoolean() ? x0[x0i] : x1[x1i];
    dest[desti++] = add; // we picked a operation and added it
//

    for (int i = x0i;; i++) { // mark the operation as done in x0
      if ((x0[i] == add) && (!doneX0[i])) { // find added job
        doneX0[i] = true; // found it and marked it
        break; // quit operation finding loop
      }
    }
    while (doneX0[x0i]) x0i++; // move x0i to first unfinished operation in x0

//
//
//
//
//
//
//
//
  } // end of function
}
```

Implementing Sequence Recombination

```java
package aitoa.examples.jssp;

public class JSSPBinaryOperatorSequence implements IBinarySearchOperator<int[]> {
  public void apply(int[] x0, int[] x1, int[] dest, Random random) {
    boolean[] doneX0 = new boolean[x0.length]; // can be stored as reuseable
    boolean[] doneX1 = new boolean[x0.length]; // member variable instead

    int length = doneX0.length; // length = m*n
    int desti  = 0;             // all array indexes = 0
    int x0i    = 0;             // index of first unfinished operation in x0
    int x1i    = 0;             // index of first unfinished operation in x1

//
// randomly chose a source point and pick next operation from it
    int add = random.nextBoolean() ? x0[x0i] : x1[x1i];
    dest[desti++] = add; // we picked a operation and added it
//

    for (int i = x0i;; i++) { // mark the operation as done in x0
      if ((x0[i] == add) && (!doneX0[i])) { // find added job
        doneX0[i] = true; // found it and marked it
        break; // quit operation finding loop
      }
    }
    while (doneX0[x0i]) x0i++; // move x0i to first unfinished operation in x0

    for (int i = x1i;; i++) { // mark the operation as done in x1
//
//
//
//
    }
//
//
  } // end of function
}
```

Implementing Sequence Recombination

```java
package aitoa.examples.jssp;

public class JSSPBinaryOperatorSequence implements IBinarySearchOperator<int[]> {
  public void apply(int[] x0, int[] x1, int[] dest, Random random) {
    boolean[] doneX0 = new boolean[x0.length]; // can be stored as reuseable
    boolean[] doneX1 = new boolean[x0.length]; // member variable instead

    int length = doneX0.length; // length = m*n
    int desti  = 0;             // all array indexes = 0
    int x0i    = 0;             // index of first unfinished operation in x0
    int x1i    = 0;             // index of first unfinished operation in x1

//
// randomly chose a source point and pick next operation from it
    int add = random.nextBoolean() ? x0[x0i] : x1[x1i];
    dest[desti++] = add; // we picked a operation and added it
//

    for (int i = x0i;; i++) { // mark the operation as done in x0
      if ((x0[i] == add) && (!doneX0[i])) { // find added job
        doneX0[i] = true; // found it and marked it
        break; // quit operation finding loop
      }
    }
    while (doneX0[x0i]) x0i++; // move x0i to first unfinished operation in x0

    for (int i = x1i;; i++) { // mark the operation as done in x1
      if ((x1[i] == add) && (!doneX1[i])) { // find added job
//
//
      }
    }
//
//
  } // end of function
}
```

Implementing Sequence Recombination

```java
package aitoa.examples.jssp;

public class JSSPBinaryOperatorSequence implements IBinarySearchOperator<int[]> {
  public void apply(int[] x0, int[] x1, int[] dest, Random random) {
    boolean[] doneX0 = new boolean[x0.length]; // can be stored as reuseable
    boolean[] doneX1 = new boolean[x0.length]; // member variable instead

    int length = doneX0.length; // length = m*n
    int desti  = 0;             // all array indexes = 0
    int x0i    = 0;             // index of first unfinished operation in x0
    int x1i    = 0;             // index of first unfinished operation in x1

//
// randomly chose a source point and pick next operation from it
    int add = random.nextBoolean() ? x0[x0i] : x1[x1i];
    dest[desti++] = add; // we picked a operation and added it
//

    for (int i = x0i;; i++) { // mark the operation as done in x0
      if ((x0[i] == add) && (!doneX0[i])) { // find added job
        doneX0[i] = true; // found it and marked it
        break; // quit operation finding loop
      }
    }
    while (doneX0[x0i]) x0i++; // move x0i to first unfinished operation in x0

    for (int i = x1i;; i++) { // mark the operation as done in x1
      if ((x1[i] == add) && (!doneX1[i])) { // find added job
        doneX1[i] = true; // found it and marked it
//
      }
    }
//
//
  } // end of function
}
```

Implementing Sequence Recombination

```java
package aitoa.examples.jssp;

public class JSSPBinaryOperatorSequence implements IBinarySearchOperator<int[]> {
  public void apply(int[] x0, int[] x1, int[] dest, Random random) {
    boolean[] doneX0 = new boolean[x0.length]; // can be stored as reuseable
    boolean[] doneX1 = new boolean[x0.length]; // member variable instead

    int length = doneX0.length; // length = m*n
    int desti  = 0;             // all array indexes = 0
    int x0i    = 0;             // index of first unfinished operation in x0
    int x1i    = 0;             // index of first unfinished operation in x1

//
// randomly chose a source point and pick next operation from it
    int add = random.nextBoolean() ? x0[x0i] : x1[x1i];
    dest[desti++] = add; // we picked a operation and added it
//

    for (int i = x0i;; i++) { // mark the operation as done in x0
      if ((x0[i] == add) && (!doneX0[i])) { // find added job
        doneX0[i] = true; // found it and marked it
        break; // quit operation finding loop
      }
    }
    while (doneX0[x0i]) x0i++; // move x0i to first unfinished operation in x0

    for (int i = x1i;; i++) { // mark the operation as done in x1
      if ((x1[i] == add) && (!doneX1[i])) { // find added job
        doneX1[i] = true; // found it and marked it
        break; // quit operation finding loop
      }
    }
//
//
  } // end of function
}
```

Implementing Sequence Recombination

```java
package aitoa.examples.jssp;

public class JSSPBinaryOperatorSequence implements IBinarySearchOperator<int[]> {
  public void apply(int[] x0, int[] x1, int[] dest, Random random) {
    boolean[] doneX0 = new boolean[x0.length]; // can be stored as reuseable
    boolean[] doneX1 = new boolean[x0.length]; // member variable instead

    int length = doneX0.length; // length = m*n
    int desti  = 0;             // all array indexes = 0
    int x0i    = 0;             // index of first unfinished operation in x0
    int x1i    = 0;             // index of first unfinished operation in x1

//
// randomly chose a source point and pick next operation from it
    int add = random.nextBoolean() ? x0[x0i] : x1[x1i];
    dest[desti++] = add; // we picked a operation and added it
//

    for (int i = x0i;; i++) { // mark the operation as done in x0
      if ((x0[i] == add) && (!doneX0[i])) { // find added job
        doneX0[i] = true; // found it and marked it
        break; // quit operation finding loop
      }
    }
    while (doneX0[x0i]) x0i++; // move x0i to first unfinished operation in x0

    for (int i = x1i;; i++) { // mark the operation as done in x1
      if ((x1[i] == add) && (!doneX1[i])) { // find added job
        doneX1[i] = true; // found it and marked it
        break; // quit operation finding loop
      }
    }
    while (doneX1[x1i]) x1i++; // move x1i to first unfinished operation in x1
//
  } // end of function
}
```

Implementing Sequence Recombination

```java
package aitoa.examples.jssp;

public class JSSPBinaryOperatorSequence implements IBinarySearchOperator<int[]> {
  public void apply(int[] x0, int[] x1, int[] dest, Random random) {
    boolean[] doneX0 = new boolean[x0.length]; // can be stored as reuseable
    boolean[] doneX1 = new boolean[x0.length]; // member variable instead

    int length = doneX0.length; // length = m*n
    int desti  = 0;             // all array indexes = 0
    int x0i    = 0;             // index of first unfinished operation in x0
    int x1i    = 0;             // index of first unfinished operation in x1

    for (;;) { // repeat until dest is filled, i.e., desti=length
// randomly chose a source point and pick next operation from it
      int add = random.nextBoolean() ? x0[x0i] : x1[x1i];
      dest[desti++] = add; // we picked a operation and added it
//

      for (int i = x0i;; i++) { // mark the operation as done in x0
        if ((x0[i] == add) && (!doneX0[i])) { // find added job
          doneX0[i] = true; // found it and marked it
          break; // quit operation finding loop
        }
      }
      while (doneX0[x0i]) x0i++; // move x0i to first unfinished operation in x0

      for (int i = x1i;; i++) { // mark the operation as done in x1
        if ((x1[i] == add) && (!doneX1[i])) { // find added job
          doneX1[i] = true; // found it and marked it
          break; // quit operation finding loop
        }
      }
      while (doneX1[x1i]) x1i++; // move x1i to first unfinished operation in x1
    } // loop back to main loop and to add next operation
  } // end of function
}
```

Implementing Sequence Recombination

```
package aitoa.examples.jssp;

public class JSSPBinaryOperatorSequence implements IBinarySearchOperator<int[]> {
  public void apply(int[] x0, int[] x1, int[] dest, Random random) {
    boolean[] doneX0 = new boolean[x0.length]; // can be stored as reuseable
    boolean[] doneX1 = new boolean[x0.length]; // member variable instead

    int length = doneX0.length; // length = m*n
    int desti  = 0;             // all array indexes = 0
    int x0i    = 0;             // index of first unfinished operation in x0
    int x1i    = 0;             // index of first unfinished operation in x1

    for (;;) { // repeat until dest is filled, i.e., desti=length
// randomly chose a source point and pick next operation from it
      int add = random.nextBoolean() ? x0[x0i] : x1[x1i];
      dest[desti++] = add; // we picked a operation and added it
      if (desti >= length) return;

      for (int i = x0i;; i++) { // mark the operation as done in x0
        if ((x0[i] == add) && (!doneX0[i])) { // find added job
          doneX0[i] = true; // found it and marked it
          break; // quit operation finding loop
        }
      }
      while (doneX0[x0i]) x0i++; // move x0i to first unfinished operation in x0

      for (int i = x1i;; i++) { // mark the operation as done in x1
        if ((x1[i] == add) && (!doneX1[i])) { // find added job
          doneX1[i] = true; // found it and marked it
          break; // quit operation finding loop
        }
      }
      while (doneX1[x1i]) x1i++; // move x1i to first unfinished operation in x1
    } // loop back to main loop and to add next operation
  } // end of function
}
```

Experiment and Analysis

Configuring the Algorithm

- We now have everything together, the EA that can use a binary operator and a simple idea for a binary operator.

Configuring the Algorithm

- We now have everything together, the EA that can use a binary operator and a simple idea for a binary operator.
- But now we have five parameters!

Configuring the Algorithm

- We now have everything together, the EA that can use a binary operator and a simple idea for a binary operator.
- But now we have five parameters μ.

Configuring the Algorithm

- We now have everything together, the EA that can use a binary operator and a simple idea for a binary operator.
- But now we have five parameters μ, λ.

Configuring the Algorithm

- We now have everything together, the EA that can use a binary operator and a simple idea for a binary operator.
- But now we have five parameters μ, λ, the unary operator.

Configuring the Algorithm

- We now have everything together, the EA that can use a binary operator and a simple idea for a binary operator.
- But now we have five parameters μ, λ, the unary operator, the binary operator.

Configuring the Algorithm

- We now have everything together, the EA that can use a binary operator and a simple idea for a binary operator.
- But now we have five parameters μ, λ, the unary operator, the binary operator, and the crossover rate cr.

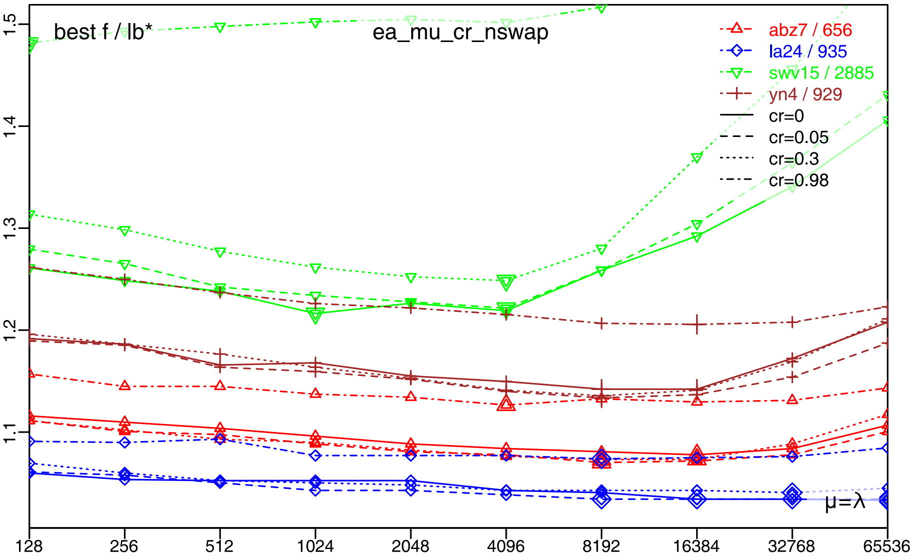
Configuring the Algorithm

- We now have everything together, the EA that can use a binary operator and a simple idea for a binary operator.
- But now we have five parameters μ, λ, the unary operator, the binary operator, and the crossover rate cr.
- Let's stick with $\mu = \lambda$, `nswap`, and our `sequence` recombination operator.

Configuring the Algorithm

- We now have everything together, the EA that can use a binary operator and a simple idea for a binary operator.
- But now we have five parameters μ, λ, the unary operator, the binary operator, and the crossover rate cr.
- Let's stick with $\mu = \lambda$, `nswap`, and our `sequence` recombination operator.
- This leaves us to choose the value of λ and cr.

Configuring the Algorithm



Configuring the Algorithm

- We now have everything together, the EA that can use a binary operator and a simple idea for a binary operator.
- But now we have five parameters μ, λ, the unary operator, the binary operator, and the crossover rate cr.
- Let's stick with $\mu = \lambda$, `nswap`, and our `sequence` recombination operator.
- This leaves us to choose the value of λ and cr.
- The improvements that the binary operator offered us in this scenario are quite small.

Configuring the Algorithm

- We now have everything together, the EA that can use a binary operator and a simple idea for a binary operator.
- But now we have five parameters μ, λ, the unary operator, the binary operator, and the crossover rate cr.
- Let's stick with $\mu = \lambda$, `nswap`, and our `sequence` recombination operator.
- This leaves us to choose the value of λ and cr.
- The improvements that the binary operator offered us in this scenario are quite small.
- Nevertheless, creating 5% of the offspring with it seems a reasonable idea at $\lambda = \mu = 8192$.

So what do we get?

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

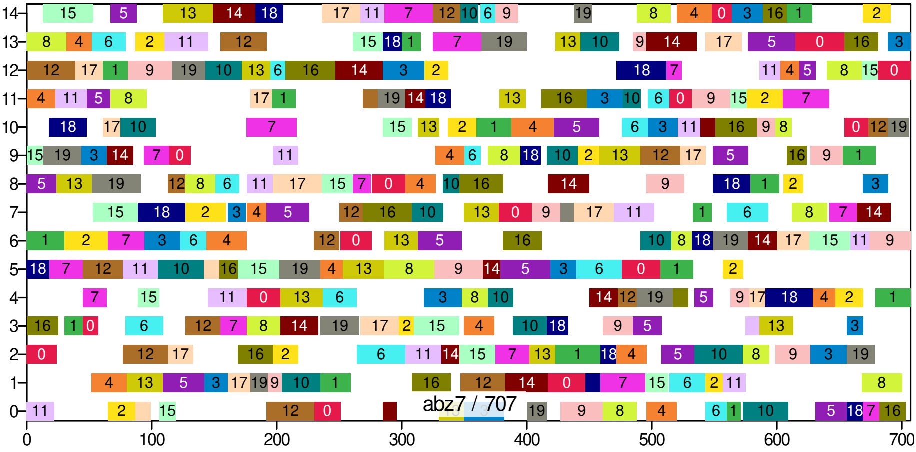
So what do we get?

- 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_65536_nswap	712	731	732	**6**	96s	21'189'358
	ea_16384_nswap	691	707	707	8	151s	25'293'859
	ea_8192_5%_nswap	**684**	**703**	**702**	8	54s	10'688'314
la24	hcr_65536_nswap	**942**	973	974	**8**	71s	31'466'420
	ea_16384_nswap	945	968	**967**	12	39s	10'161'119
	ea_8192_5%_nswap	943	**967**	**967**	11	18s	4'990'002
swv15	hcr_65536_nswap	3740	3818	3826	**35**	89s	10'783'296
	ea_16384_nswap	3577	3723	3728	50	178s	18'897'833
	ea_8192_5%_nswap	**3498**	**3631**	**3632**	65	178s	17'747'983
yn4	hcr_65536_nswap	1068	1109	1110	**12**	78s	18'756'636
	ea_16384_nswap	**1022**	1063	1061	16	168s	26'699'633
	ea_8192_5%_nswap	1026	**1056**	**1053**	17	114s	13'206'552

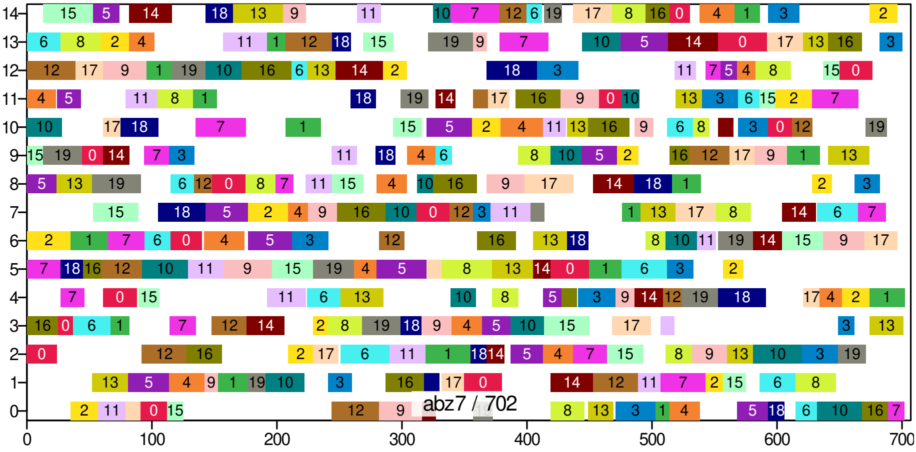
So what do we get?

ea_16384_nswap: median result of 3 min of the EA with $\mu = \lambda = 16'384$ with nswap unary operator



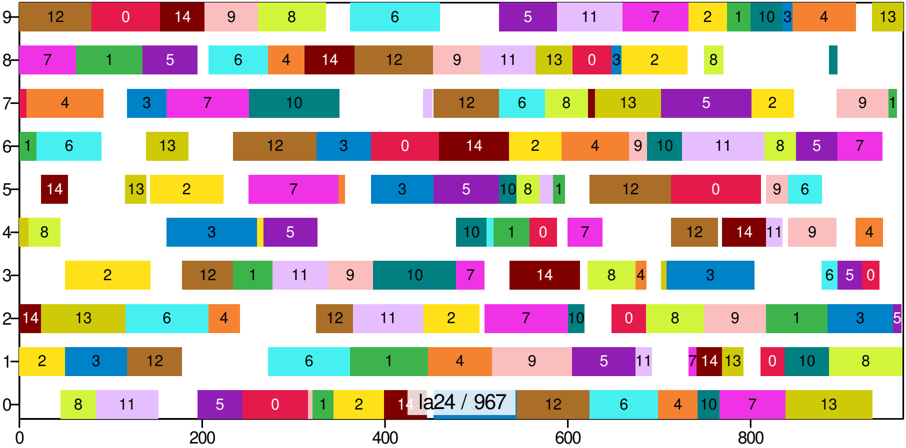
So what do we get?

ea_8192_5%_nswap: median result of 3 min of the EA with $\mu = \lambda = 8'192$ with nswap unary operator and 5% sequence recombination



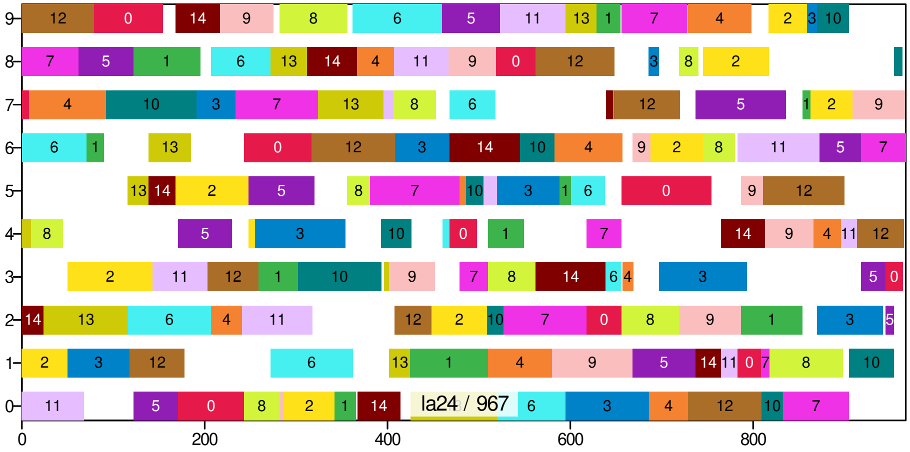
So what do we get?

`ea_16384_nswap`: median result of 3 min of the EA with $\mu = \lambda = 16'384$ with `nswap` unary operator



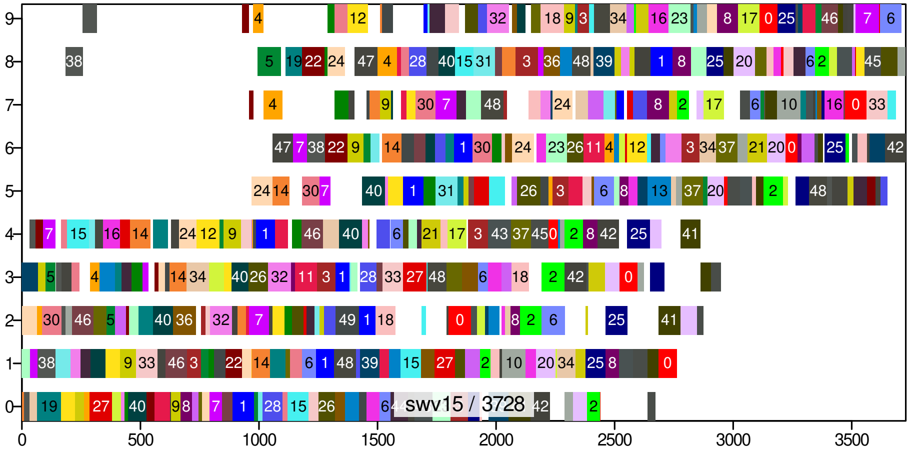
So what do we get?

`ea_8192_5%_nswap`: median result of 3 min of the EA with $\mu = \lambda = 8'192$ with `nswap` unary operator and 5% sequence recombination



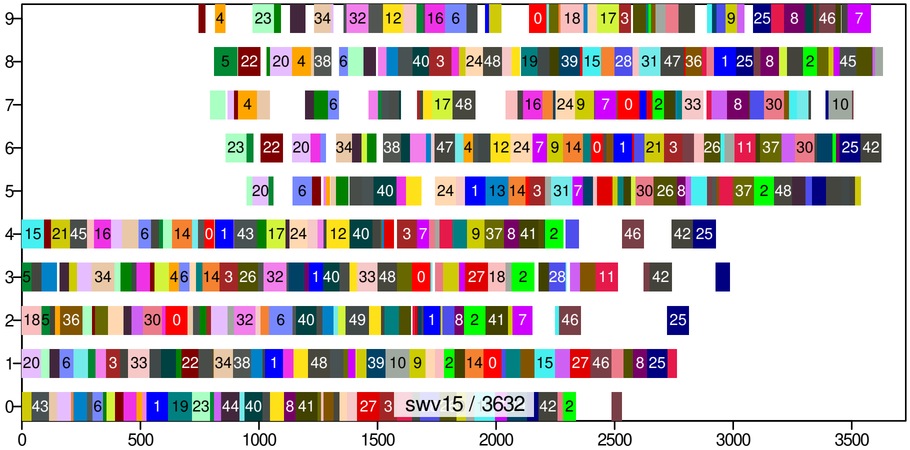
So what do we get?

`ea_16384_nswap`: median result of 3 min of the EA with $\mu = \lambda = 16'384$ with `nswap` unary operator



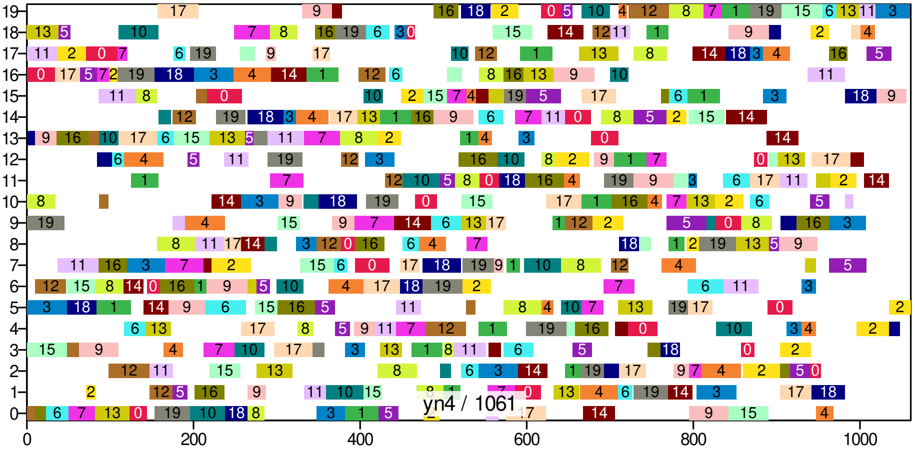
So what do we get?

`ea_8192_5%_nswap`: median result of 3 min of the EA with $\mu = \lambda = 8'192$ with `nswap` unary operator and 5% `sequence` recombination



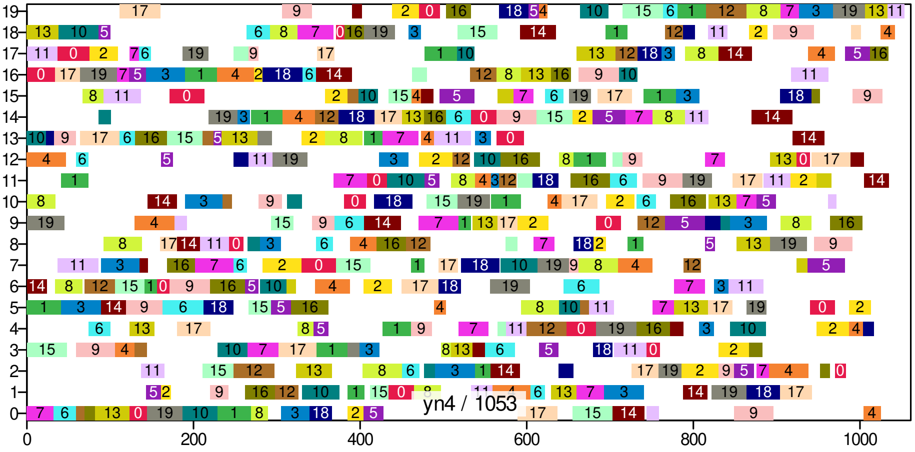
So what do we get?

`ea_16384_nswap`: median result of 3 min of the EA with $\mu = \lambda = 16'384$ with `nswap` unary operator



So what do we get?

ea_8192_5%_nswap: median result of 3 min of the EA with $\mu = \lambda = 8'192$ with nswap unary operator and 5% sequence recombination

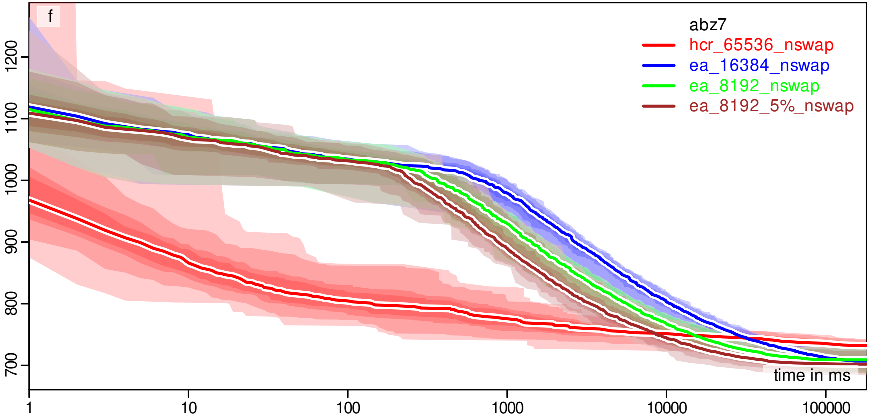


Progress over Time

What progress does the algorithm make over time?

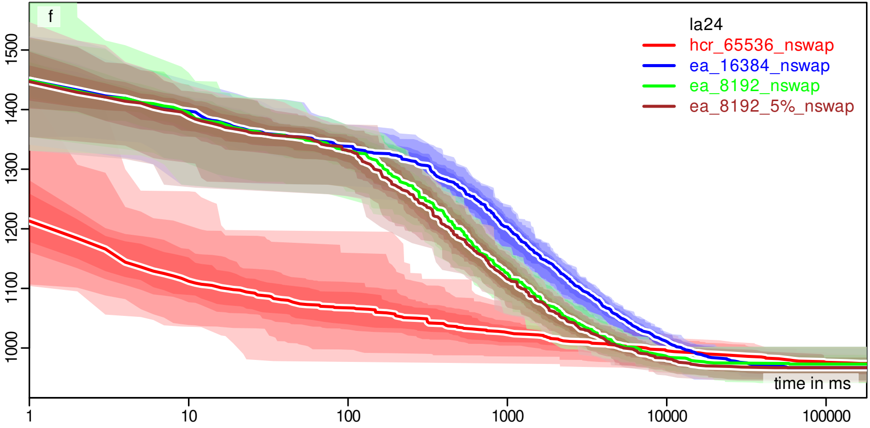
Progress over Time

What progress does the algorithm make over time?



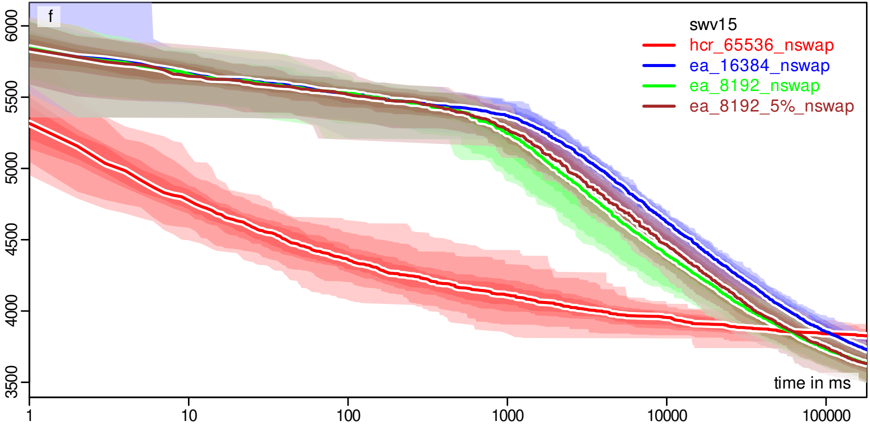
Progress over Time

What progress does the algorithm make over time?



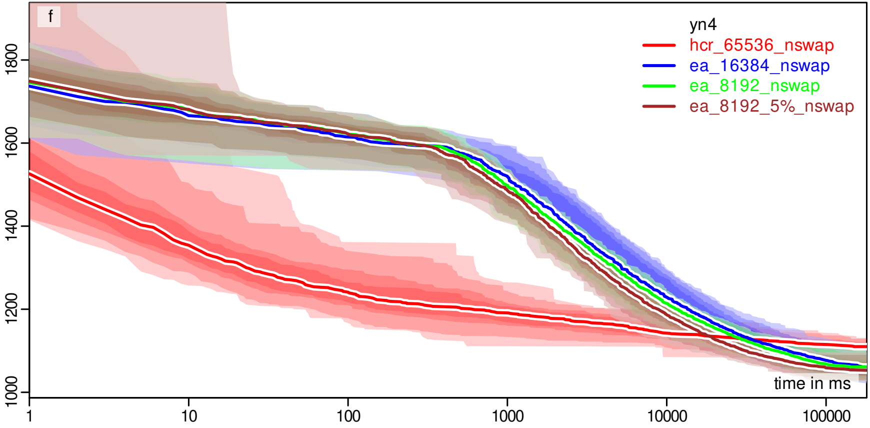
Progress over Time

What progress does the algorithm make over time?



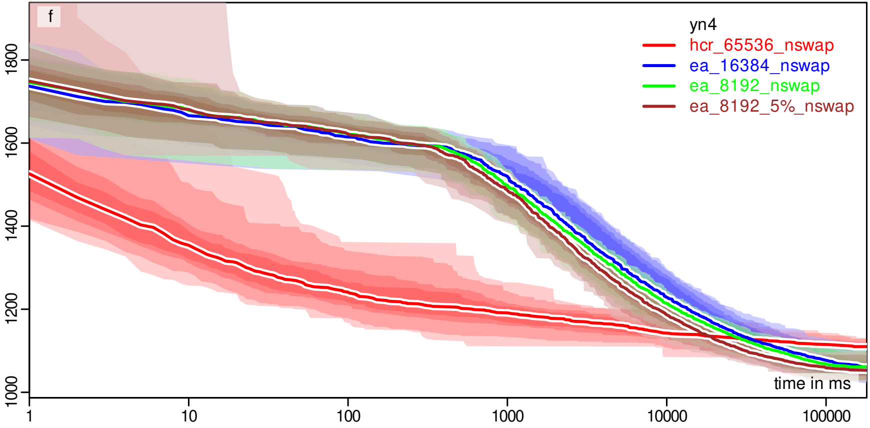
Progress over Time

What progress does the algorithm make over time?



Progress over Time

What progress does the algorithm make over time?



There is no big difference between the EA with and without recombination – but the one with recombination is a little bit better.

Recombination

- In some application areas, the binary operator can very significantly improve the result quality.

Recombination

- In some application areas, the binary operator can very significantly improve the result quality.
- Here, our idea does not work that well, although it is a bit helpful.

Algorithm Concept: Increased Diversity via Clearing

Diversity

- When is the population of the EA useless?

Diversity

- When is the population of the EA useless?
- If all the solutions in it are the same!

Diversity

- When is the population of the EA useless?
- If all the solutions in it are the same!
- When is a population of the EA useful?

Diversity

- When is the population of the EA useless?
- If all the solutions in it are the same!
- When is a population of the EA useful?
- When the elements of it represent different good solution traits.

Diversity

- When is the population of the EA useless?
- If all the solutions in it are the same!
- When is a population of the EA useful?
- When the elements of it represent different good solution traits – when they are diverse.

Diversity

- When is the population of the EA useless?
- If all the solutions in it are the same!
- When is a population of the EA useful?
- When the elements of it represent different good solution traits – when they are diverse.
- Many methods have been devised to ensure the diversity of a population.

Diversity

- When is the population of the EA useless?
- If all the solutions in it are the same!
- When is a population of the EA useful?
- When the elements of it represent different good solution traits – when they are diverse.
- Many methods have been devised to ensure the diversity of a population, to prevent the population from collapsing to a single point in the search space.[11–13]

Clearing

- We will here consider a very simple approach to preserve population diversity: clearing[11 14].

Clearing

- We will here consider a very simple approach to preserve population diversity: clearing[11 14].
- Furthermore, we will apply the simplest version of this approach.

Clearing

- We will here consider a very simple approach to preserve population diversity: clearing[11 14].
- Furthermore, we will apply the simplest version of this approach.
- Every time, when μ out of the $\mu + \lambda$ records are selected, one prior step is applied

Clearing

- We will here consider a very simple approach to preserve population diversity: clearing[11, 14].
- Furthermore, we will apply the simplest version of this approach.
- Every time, when μ out of the $\mu + \lambda$ records are selected, one prior step is applied: we ensure that there is only one record per objective value in the population.

Clearing

- We will here consider a very simple approach to preserve population diversity: clearing[11, 14].
- Furthermore, we will apply the simplest version of this approach.
- Every time, when μ out of the $\mu + \lambda$ records are selected, one prior step is applied: we ensure that there is only one record per objective value in the population.
- We call the EA with clearing and recombination eac.

Ingredient: Clearing

```
package aitoa.algorithms;

public class Utils {
//
//
//

//
//

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

Ingredient: Clearing

```
package aitoa.algorithms;

public class Utils {
// useless stuff omitted
  public static int qualityBasedClearing(Record<?>[] array, int max) {
//

//
//

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

Ingredient: Clearing

```
package aitoa.algorithms;

public class Utils {
// useless stuff omitted
  public static int qualityBasedClearing(Record<?>[] array, int max) {
    Arrays.sort(array, Record.BY_QUALITY); // best -> first

//
//

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

Ingredient: Clearing

```
package aitoa.algorithms;

public class Utils {
// useless stuff omitted
  public static int qualityBasedClearing(Record<?>[] array, int max) {
    Arrays.sort(array, Record.BY_QUALITY); // best -> first

    int    unique      = 0;
//

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

Ingredient: Clearing

```
package aitoa.algorithms;

public class Utils {
// useless stuff omitted
  public static int qualityBasedClearing(Record<?>[] array, int max) {
    Arrays.sort(array, Record.BY_QUALITY); // best -> first

    int    unique      = 0;
    double lastQuality = Double.NEGATIVE_INFINITY; // impossibly bad

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

Ingredient: Clearing

```
package aitoa.algorithms;

public class Utils {
// useless stuff omitted
  public static int qualityBasedClearing(Record<?>[] array, int max) {
    Arrays.sort(array, Record.BY_QUALITY); // best -> first

    int    unique      = 0;
    double lastQuality = Double.NEGATIVE_INFINITY; // impossibly bad

    for (int index = 0; index < array.length; index++) {
//
//
//
//
//
//
//
//
//
//
//
//
//
    }
//
  }
}
```

Ingredient: Clearing

```java
package aitoa.algorithms;

public class Utils {
// useless stuff omitted
  public static int qualityBasedClearing(Record<?>[] array, int max) {
    Arrays.sort(array, Record.BY_QUALITY); // best -> first

    int    unique      = 0;
    double lastQuality = Double.NEGATIVE_INFINITY; // impossibly bad

    for (int index = 0; index < array.length; index++) {
      Record<?> current        = array[index];
//
//
//
//
//
//
//
//
//
//
//
//
    }
//
  }
}
```

Ingredient: Clearing

```
package aitoa.algorithms;

public class Utils {
// useless stuff omitted
  public static int qualityBasedClearing(Record<?>[] array, int max) {
    Arrays.sort(array, Record.BY_QUALITY); // best -> first

    int    unique      = 0;
    double lastQuality = Double.NEGATIVE_INFINITY; // impossibly bad

    for (int index = 0; index < array.length; index++) {
      Record<?> current        = array[index];
      double    currentQuality = current.quality;
//
//
//
//
//
//
//
//
//
//
//
    }
//
  }
}
```

Ingredient: Clearing

```
package aitoa.algorithms;

public class Utils {
// useless stuff omitted
  public static int qualityBasedClearing(Record<?>[] array, int max) {
    Arrays.sort(array, Record.BY_QUALITY); // best -> first

    int    unique      = 0;
    double lastQuality = Double.NEGATIVE_INFINITY; // impossibly bad

    for (int index = 0; index < array.length; index++) {
      Record<?> current        = array[index];
      double    currentQuality = current.quality;
      if (currentQuality > lastQuality) { // unique so-far
//
//
//
//
//
//
//
//
//
      }
    }
//
  }
}
```

Ingredient: Clearing

```java
package aitoa.algorithms;

public class Utils {
// useless stuff omitted
  public static int qualityBasedClearing(Record<?>[] array, int max) {
    Arrays.sort(array, Record.BY_QUALITY); // best -> first

    int    unique      = 0;
    double lastQuality = Double.NEGATIVE_INFINITY; // impossibly bad

    for (int index = 0; index < array.length; index++) {
      Record<?> current        = array[index];
      double    currentQuality = current.quality;
      if (currentQuality > lastQuality) { // unique so-far
        if (index > unique) { // need to move forward?
//
//
//
        }
//
//
//
//
      }
    }
//
  }
}
```

Ingredient: Clearing

```java
package aitoa.algorithms;

public class Utils {
// useless stuff omitted
  public static int qualityBasedClearing(Record<?>[] array, int max) {
    Arrays.sort(array, Record.BY_QUALITY); // best -> first

    int    unique      = 0;
    double lastQuality = Double.NEGATIVE_INFINITY; // impossibly bad

    for (int index = 0; index < array.length; index++) {
      Record<?> current        = array[index];
      double    currentQuality = current.quality;
      if (currentQuality > lastQuality) { // unique so-far
        if (index > unique) { // need to move forward?
          Record<?> other = array[unique];
//
//
        }
//
//
//
//
      }
    }
//
  }
}
```

Ingredient: Clearing

```java
package aitoa.algorithms;

public class Utils {
// useless stuff omitted
  public static int qualityBasedClearing(Record<?>[] array, int max) {
    Arrays.sort(array, Record.BY_QUALITY); // best -> first

    int    unique      = 0;
    double lastQuality = Double.NEGATIVE_INFINITY; // impossibly bad

    for (int index = 0; index < array.length; index++) {
      Record<?> current        = array[index];
      double    currentQuality = current.quality;
      if (currentQuality > lastQuality) { // unique so-far
        if (index > unique) { // need to move forward?
          Record<?> other = array[unique];
          array[unique]   = current; // swap with first non-unique
//
        }
//
//
//
//
      }
    }
//
  }
}
```

Ingredient: Clearing

```java
package aitoa.algorithms;

public class Utils {
// useless stuff omitted
  public static int qualityBasedClearing(Record<?>[] array, int max) {
    Arrays.sort(array, Record.BY_QUALITY); // best -> first

    int    unique      = 0;
    double lastQuality = Double.NEGATIVE_INFINITY; // impossibly bad

    for (int index = 0; index < array.length; index++) {
      Record<?> current        = array[index];
      double    currentQuality = current.quality;
      if (currentQuality > lastQuality) { // unique so-far
        if (index > unique) { // need to move forward?
          Record<?> other = array[unique];
          array[unique]   = current; // swap with first non-unique
          array[index]    = other;
        }
//
//
//
//
      }
    }
//
  }
}
```

Ingredient: Clearing

```java
package aitoa.algorithms;

public class Utils {
// useless stuff omitted
  public static int qualityBasedClearing(Record<?>[] array, int max) {
    Arrays.sort(array, Record.BY_QUALITY); // best -> first

    int    unique      = 0;
    double lastQuality = Double.NEGATIVE_INFINITY; // impossibly bad

    for (int index = 0; index < array.length; index++) {
      Record<?> current        = array[index];
      double    currentQuality = current.quality;
      if (currentQuality > lastQuality) { // unique so-far
        if (index > unique) { // need to move forward?
          Record<?> other = array[unique];
          array[unique]   = current; // swap with first non-unique
          array[index]    = other;
        }
        lastQuality = currentQuality; // update new quality
//
//
//
      }
    }
//
  }
}
```

Ingredient: Clearing

```
package aitoa.algorithms;

public class Utils {
// useless stuff omitted
  public static int qualityBasedClearing(Record<?>[] array, int max) {
    Arrays.sort(array, Record.BY_QUALITY); // best -> first

    int    unique      = 0;
    double lastQuality = Double.NEGATIVE_INFINITY; // impossibly bad

    for (int index = 0; index < array.length; index++) {
      Record<?> current        = array[index];
      double    currentQuality = current.quality;
      if (currentQuality > lastQuality) { // unique so-far
        if (index > unique) { // need to move forward?
          Record<?> other = array[unique];
          array[unique]   = current; // swap with first non-unique
          array[index]    = other;
        }
        lastQuality = currentQuality; // update new quality
        if ((++unique) >= max) { // are we finished?
//
        }
      }
    }
//
  }
}
```

Ingredient: Clearing

```java
package aitoa.algorithms;

public class Utils {
// useless stuff omitted
  public static int qualityBasedClearing(Record<?>[] array, int max) {
    Arrays.sort(array, Record.BY_QUALITY); // best -> first

    int    unique      = 0;
    double lastQuality = Double.NEGATIVE_INFINITY; // impossibly bad

    for (int index = 0; index < array.length; index++) {
      Record<?> current        = array[index];
      double    currentQuality = current.quality;
      if (currentQuality > lastQuality) { // unique so-far
        if (index > unique) { // need to move forward?
          Record<?> other = array[unique];
          array[unique]   = current; // swap with first non-unique
          array[index]    = other;
        }
        lastQuality = currentQuality; // update new quality
        if ((++unique) >= max) { // are we finished?
          return unique; // then quit: unique == max
        }
      }
    }
//
  }
}
```

Ingredient: Clearing

```java
package aitoa.algorithms;

public class Utils {
// useless stuff omitted
  public static int qualityBasedClearing(Record<?>[] array, int max) {
    Arrays.sort(array, Record.BY_QUALITY); // best -> first

    int    unique      = 0;
    double lastQuality = Double.NEGATIVE_INFINITY; // impossibly bad

    for (int index = 0; index < array.length; index++) {
      Record<?> current        = array[index];
      double    currentQuality = current.quality;
      if (currentQuality > lastQuality) { // unique so-far
        if (index > unique) { // need to move forward?
          Record<?> other = array[unique];
          array[unique]   = current; // swap with first non-unique
          array[index]    = other;
        }
        lastQuality = currentQuality; // update new quality
        if ((++unique) >= max) { // are we finished?
          return unique; // then quit: unique == max
        }
      }
    }
    return unique; // return number of unique: 1<=unique<=max
  }
}
```

Implementation: EA with Recombination and Clearing

```java
package aitoa.algorithms;

public class EA<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

    for (;;) { // main loop: one iteration = one generation
//
      Arrays.sort(P, Record.BY_QUALITY); // sort the population: mu best at front
      RandomUtils.shuffle(random, P, 0, this.mu); // shuffle parents for fairness
      int p1 = -1; // index to iterate over first parent
      for (int index = P.length; (--index) >= this.mu;) { // overwrite lambda worst
        if (process.shouldTerminate()) return;
        Record<X> dest = P[index];
        p1 = (p1 + 1) % this.mu; // step the parent 1 index
        Record<X> sel  = P[p1];
        if (random.nextDouble() <= this.cr) { // crossover!
          int p2;
          do { // find a second, different record
            p2 = random.nextInt(this.mu);
          } while (p2 == p1); // repeat until p1 != p2
          this.binary.apply(sel.x, P[p2].x, dest.x, random); // perform recombination
        } else this.unary.apply(sel.x, dest.x, random); // generate offspring via unary
        dest.quality = process.evaluate(dest.x); // evaluate offspring
      } // the end of the offspring generation
    } // the end of the main loop
  } // end solve
} // end class
```

Implementation: EA with Recombination and Clearing

```java
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

    for (;;) { // main loop: one iteration = one generation
      RandomUtils.shuffle(random, P, 0, P.length); // make fair
      int u = Utils.qualityBasedClearing(P, this.mu);
      RandomUtils.shuffle(random, P, 0, u); // for fairness
      int p1 = -1; // index to iterate over first parent
      for (int index = P.length; (--index) >= u;) { // overwrite non-unique and worst
        if (process.shouldTerminate()) return;
        Record<X> dest = P[index];
        p1 = (p1 + 1) % u; // step the parent 1 index
        Record<X> sel  = P[p1];
        if (random.nextDouble() <= this.cr) { // crossover!
          int p2;
          do { // find a second, different record
            p2 = random.nextInt(u);
          } while (p2 == p1); // repeat until p1 != p2
          this.binary.apply(sel.x, P[p2].x, dest.x, random); // perform recombination
        } else this.unary.apply(sel.x, dest.x, random); // generate offspring via unary
        dest.quality = process.evaluate(dest.x); // evaluate offspring
      } // the end of the offspring generation
    } // the end of the main loop
  } // end solve
} // end class
```

Implementation: EA with Recombination and Clearing

```
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

//
//
//
//

//
//
//
//
//
//

//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
} // end class
```

Implementation: EA with Recombination and Clearing

```
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
//
//
//

//
//
//
//
//
//

//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
  } // end solve
} // end class
```

Implementation: EA with Recombination and Clearing

```
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
//
//

//
//
//
//
//
//

//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
  } // end solve
} // end class
```

Implementation: EA with Recombination and Clearing

```
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
//

//
//
//
//
//
//

//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
  } // end solve
} // end class
```

Implementation: EA with Recombination and Clearing

```
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

//
//
//
//
//
//

//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
  } // end solve
} // end class
```

Implementation: EA with Recombination and Clearing

```java
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
//
//
//
//
    } // end of filling the first population

//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
  } // end solve
} // end class
```

Implementation: EA with Recombination and Clearing

```
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create(); // allocate point
//
//
//
    } // end of filling the first population

//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
  } // end solve
} // end class
```

Implementation: EA with Recombination and Clearing

```
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
//
//
    } // end of filling the first population

//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
  } // end solve
} // end class
```

Implementation: EA with Recombination and Clearing

```
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
//
    } // end of filling the first population

//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
  } // end solve
} // end class
```

Implementation: EA with Recombination and Clearing

```
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
  } // end solve
} // end class
```

Implementation: EA with Recombination and Clearing

```java
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
    RandomUtils.shuffle(random, P, 0, P.length); // make fair
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
  } // end solve
} // end class
```

Implementation: EA with Recombination and Clearing

```
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
    RandomUtils.shuffle(random, P, 0, P.length); // make fair
    int u = Utils.qualityBasedClearing(P, this.mu);
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
  } // end solve
} // end class
```

Implementation: EA with Recombination and Clearing

```
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
    RandomUtils.shuffle(random, P, 0, P.length); // make fair
    int u = Utils.qualityBasedClearing(P, this.mu);
    RandomUtils.shuffle(random, P, 0, u); // for fairness
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
  } // end solve
} // end class
```

Implementation: EA with Recombination and Clearing

```java
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
    RandomUtils.shuffle(random, P, 0, P.length); // make fair
    int u = Utils.qualityBasedClearing(P, this.mu);
    RandomUtils.shuffle(random, P, 0, u); // for fairness
    int p1 = -1; // index to iterate over first parent
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
  } // end solve
} // end class
```

Implementation: EA with Recombination and Clearing

```
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
    RandomUtils.shuffle(random, P, 0, P.length); // make fair
    int u = Utils.qualityBasedClearing(P, this.mu);
    RandomUtils.shuffle(random, P, 0, u); // for fairness
    int p1 = -1; // index to iterate over first parent
    for (int index = P.length; (--index) >= u;) { // overwrite non-unique and worst
//
//
//
//
//
//
//
//
//
//
//
//
    } // the end of the offspring generation
//
  } // end solve
} // end class
```

Implementation: EA with Recombination and Clearing

```
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
    RandomUtils.shuffle(random, P, 0, P.length); // make fair
    int u = Utils.qualityBasedClearing(P, this.mu);
    RandomUtils.shuffle(random, P, 0, u); // for fairness
    int p1 = -1; // index to iterate over first parent
    for (int index = P.length; (--index) >= u;) { // overwrite non-unique and worst
      if (process.shouldTerminate()) return;
//
//
//
//
//
//
//
//
//
//
//
    } // the end of the offspring generation
//
  } // end solve
} // end class
```

Implementation: EA with Recombination and Clearing

```
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
    RandomUtils.shuffle(random, P, 0, P.length); // make fair
    int u = Utils.qualityBasedClearing(P, this.mu);
    RandomUtils.shuffle(random, P, 0, u); // for fairness
    int p1 = -1; // index to iterate over first parent
    for (int index = P.length; (--index) >= u;) { // overwrite non-unique and worst
      if (process.shouldTerminate()) return;
      Record<X> dest = P[index];
//
//
//
//
//
//
//
//
//
//
    } // the end of the offspring generation
//
  } // end solve
} // end class
```

Implementation: EA with Recombination and Clearing

```java
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
    RandomUtils.shuffle(random, P, 0, P.length); // make fair
    int u = Utils.qualityBasedClearing(P, this.mu);
    RandomUtils.shuffle(random, P, 0, u); // for fairness
    int p1 = -1; // index to iterate over first parent
    for (int index = P.length; (--index) >= u;) { // overwrite non-unique and worst
      if (process.shouldTerminate()) return;
      Record<X> dest = P[index];
      p1 = (p1 + 1) % u; // step the parent 1 index
//
//
//
//
//
//
//
//
//
    } // the end of the offspring generation
//
  } // end solve
} // end class
```

Implementation: EA with Recombination and Clearing

```java
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
    RandomUtils.shuffle(random, P, 0, P.length); // make fair
    int u = Utils.qualityBasedClearing(P, this.mu);
    RandomUtils.shuffle(random, P, 0, u); // for fairness
    int p1 = -1; // index to iterate over first parent
    for (int index = P.length; (--index) >= u;) { // overwrite non-unique and worst
      if (process.shouldTerminate()) return;
      Record<X> dest = P[index];
      p1 = (p1 + 1) % u; // step the parent 1 index
      Record<X> sel  = P[p1];
//
//
//
//
//
//
//
//
    } // the end of the offspring generation
//
  } // end solve
} // end class
```

Implementation: EA with Recombination and Clearing

```java
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
    RandomUtils.shuffle(random, P, 0, P.length); // make fair
    int u = Utils.qualityBasedClearing(P, this.mu);
    RandomUtils.shuffle(random, P, 0, u); // for fairness
    int p1 = -1; // index to iterate over first parent
    for (int index = P.length; (--index) >= u;) { // overwrite non-unique and worst
      if (process.shouldTerminate()) return;
      Record<X> dest = P[index];
      p1 = (p1 + 1) % u; // step the parent 1 index
      Record<X> sel  = P[p1];
      if (random.nextDouble() <= this.cr) { // crossover!
//
//
//
//
//
      }
//
    } // the end of the offspring generation
//
  } // end solve
} // end class
```

Implementation: EA with Recombination and Clearing

```
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
    RandomUtils.shuffle(random, P, 0, P.length); // make fair
    int u = Utils.qualityBasedClearing(P, this.mu);
    RandomUtils.shuffle(random, P, 0, u); // for fairness
    int p1 = -1; // index to iterate over first parent
    for (int index = P.length; (--index) >= u;) { // overwrite non-unique and worst
      if (process.shouldTerminate()) return;
      Record<X> dest = P[index];
      p1 = (p1 + 1) % u; // step the parent 1 index
      Record<X> sel  = P[p1];
      if (random.nextDouble() <= this.cr) { // crossover!
        int p2 = random.nextInt(u);
//
//
//
//
      }
//
    } // the end of the offspring generation
//
  } // end solve
} // end class
```

Implementation: EA with Recombination and Clearing

```java
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
    RandomUtils.shuffle(random, P, 0, P.length); // make fair
    int u = Utils.qualityBasedClearing(P, this.mu);
    RandomUtils.shuffle(random, P, 0, u); // for fairness
    int p1 = -1; // index to iterate over first parent
    for (int index = P.length; (--index) >= u;) { // overwrite non-unique and worst
      if (process.shouldTerminate()) return;
      Record<X> dest = P[index];
      p1 = (p1 + 1) % u; // step the parent 1 index
      Record<X> sel  = P[p1];
      if (random.nextDouble() <= this.cr) { // crossover!
        int p2;
        do { // find a second, different record
          p2 = random.nextInt(u);
        } while (p2 == p1); // repeat until p1 != p2
//
      }
//
    } // the end of the offspring generation
//
  } // end solve
} // end class
```

Implementation: EA with Recombination and Clearing

```java
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
    RandomUtils.shuffle(random, P, 0, P.length); // make fair
    int u = Utils.qualityBasedClearing(P, this.mu);
    RandomUtils.shuffle(random, P, 0, u); // for fairness
    int p1 = -1; // index to iterate over first parent
    for (int index = P.length; (--index) >= u;) { // overwrite non-unique and worst
      if (process.shouldTerminate()) return;
      Record<X> dest = P[index];
      p1 = (p1 + 1) % u; // step the parent 1 index
      Record<X> sel  = P[p1];
      if (random.nextDouble() <= this.cr) { // crossover!
        int p2;
        do { // find a second, different record
          p2 = random.nextInt(u);
        } while (p2 == p1); // repeat until p1 != p2
        this.binary.apply(sel.x, P[p2].x, dest.x, random); // perform recombination
      }
//
    } // the end of the offspring generation
//
  } // end solve
} // end class
```

Implementation: EA with Recombination and Clearing

```java
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
    RandomUtils.shuffle(random, P, 0, P.length); // make fair
    int u = Utils.qualityBasedClearing(P, this.mu);
    RandomUtils.shuffle(random, P, 0, u); // for fairness
    int p1 = -1; // index to iterate over first parent
    for (int index = P.length; (--index) >= u;) { // overwrite non-unique and worst
      if (process.shouldTerminate()) return;
      Record<X> dest = P[index];
      p1 = (p1 + 1) % u; // step the parent 1 index
      Record<X> sel  = P[p1];
      if (random.nextDouble() <= this.cr) { // crossover!
        int p2;
        do { // find a second, different record
          p2 = random.nextInt(u);
        } while (p2 == p1); // repeat until p1 != p2
        this.binary.apply(sel.x, P[p2].x, dest.x, random); // perform recombination
      } else this.unary.apply(sel.x, dest.x, random); // generate offspring via unary
//
    } // the end of the offspring generation
//
  } // end solve
} // end class
```

Implementation: EA with Recombination and Clearing

```
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

//
    RandomUtils.shuffle(random, P, 0, P.length); // make fair
    int u = Utils.qualityBasedClearing(P, this.mu);
    RandomUtils.shuffle(random, P, 0, u); // for fairness
    int p1 = -1; // index to iterate over first parent
    for (int index = P.length; (--index) >= u;) { // overwrite non-unique and worst
      if (process.shouldTerminate()) return;
      Record<X> dest = P[index];
      p1 = (p1 + 1) % u; // step the parent 1 index
      Record<X> sel  = P[p1];
      if (random.nextDouble() <= this.cr) { // crossover!
        int p2;
        do { // find a second, different record
          p2 = random.nextInt(u);
        } while (p2 == p1); // repeat until p1 != p2
        this.binary.apply(sel.x, P[p2].x, dest.x, random); // perform recombination
      } else this.unary.apply(sel.x, dest.x, random); // generate offspring via unary
      dest.quality = process.evaluate(dest.x); // evaluate offspring
    } // the end of the offspring generation
//
  } // end solve
} // end class
```

Implementation: EA with Recombination and Clearing

```java
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {

  public void solve(IBlackBoxProcess<X, Y> process) {
    Random      random      = process.getRandom();
    ISpace<X>   searchSpace = process.getSearchSpace();
    Record<X>[] P           = new Record[this.mu + this.lambda];

    for (int i = P.length; (--i) >= 0;) { // first generation: fill P with random points
      X x  = searchSpace.create();   // allocate point
      this.nullary.apply(x, random); // fill with random data
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
      if (process.shouldTerminate()) return;
    } // end of filling the first population

    for (;;) { // main loop: one iteration = one generation
      RandomUtils.shuffle(random, P, 0, P.length); // make fair
      int u = Utils.qualityBasedClearing(P, this.mu);
      RandomUtils.shuffle(random, P, 0, u); // for fairness
      int p1 = -1; // index to iterate over first parent
      for (int index = P.length; (--index) >= u;) { // overwrite non-unique and worst
        if (process.shouldTerminate()) return;
        Record<X> dest = P[index];
        p1 = (p1 + 1) % u; // step the parent 1 index
        Record<X> sel  = P[p1];
        if (random.nextDouble() <= this.cr) { // crossover!
          int p2;
          do { // find a second, different record
            p2 = random.nextInt(u);
          } while (p2 == p1); // repeat until p1 != p2
          this.binary.apply(sel.x, P[p2].x, dest.x, random); // perform recombination
        } else this.unary.apply(sel.x, dest.x, random); // generate offspring via unary
        dest.quality = process.evaluate(dest.x); // evaluate offspring
      } // the end of the offspring generation
    } // the end of the main loop
  } // end solve
} // end class
```

Experiment and Analysis

So what do we get?

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

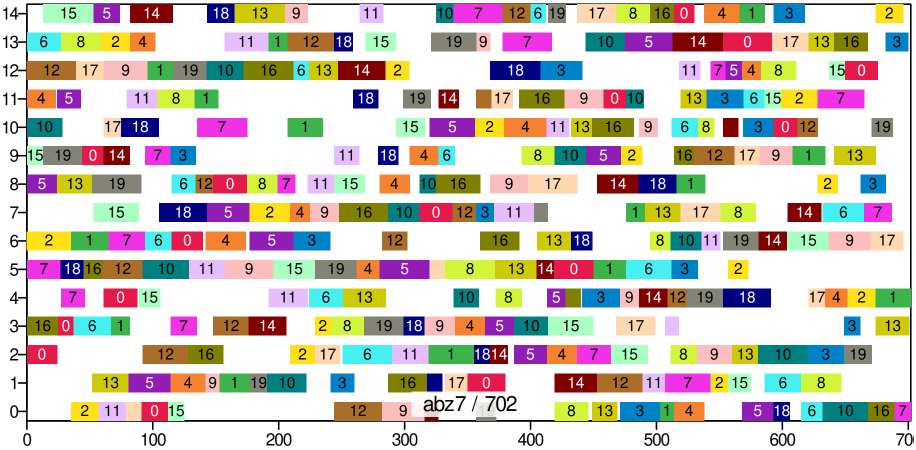
So what do we get?

- 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`	`ea_8192_5%_nswap`	684	703	702	**8**	54s	10'688'314
	`eac_4_5%_nswap`	**672**	**690**	**690**	9	68s	12'474'571
`la24`	`ea_8192_5%_nswap`	943	967	967	**11**	18s	4'990'002
	`eac_4_5%_nswap`	**935**	**963**	**961**	16	30s	9'175'579
`swv15`	`ea_8192_5%_nswap`	3498	3631	3632	**65**	178s	17'747'983
	`eac_4_5%_nswap`	**3102**	**3220**	**3224**	65	168s	18'245'534
`yn4`	`ea_8192_5%_nswap`	1026	1056	1053	**17**	114s	13'206'552
	`eac_4_5%_nswap`	**1000**	**1038**	**1037**	18	118s	15'382'072

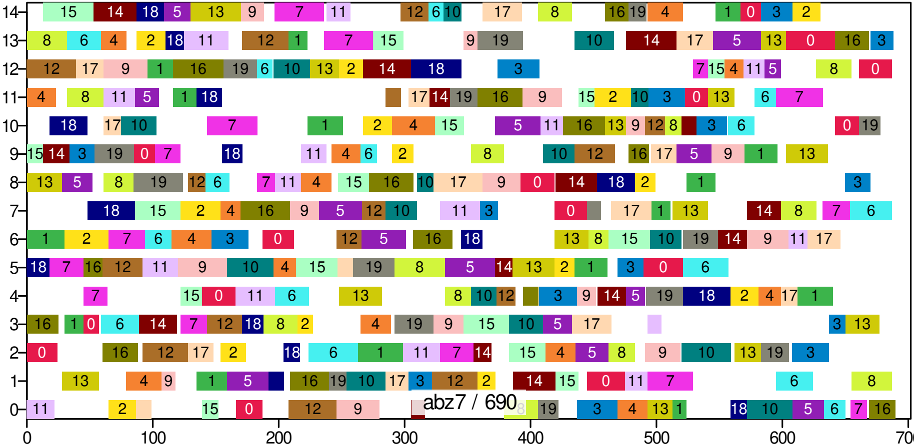
So what do we get?

`ea_8192_5%_nswap`: median result of 3 min of the EA with $\mu = \lambda = 8'192$ with `nswap` unary operator and 5% sequence recombination



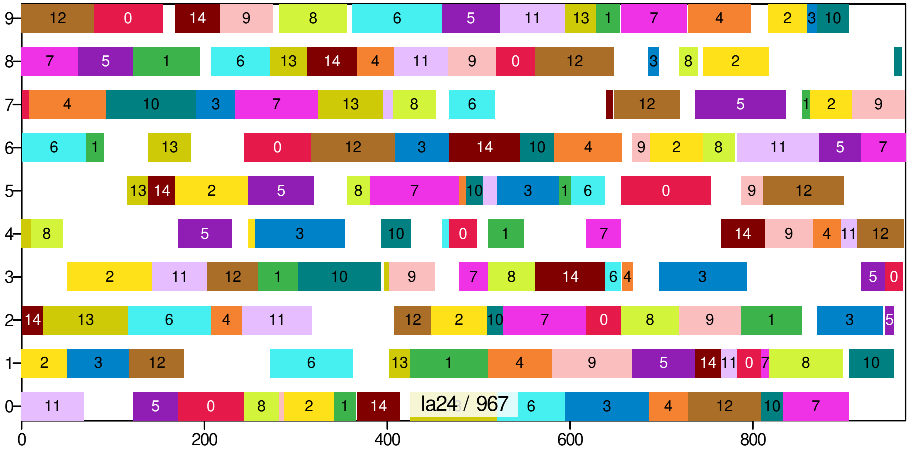
So what do we get?

`eac_4_5%_nswap`: median result of 3 min of the EA with clearing and $\mu = \lambda = 4$ with `nswap` unary operator and 5% sequence recombination



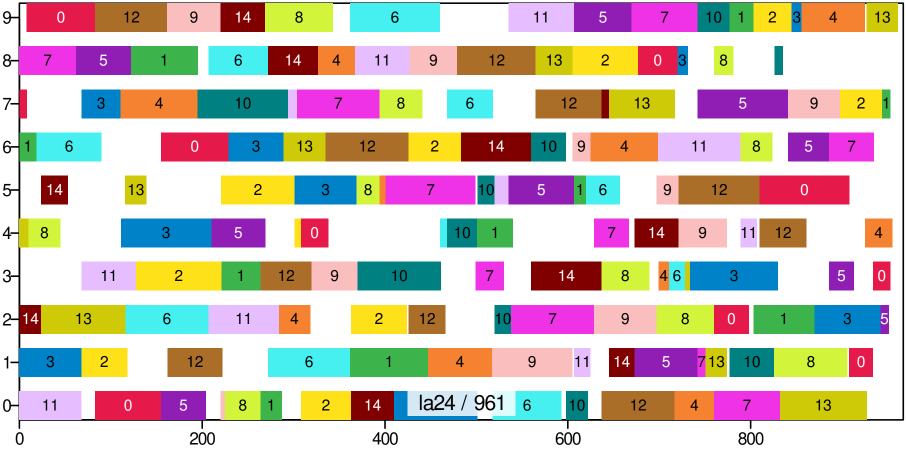
So what do we get?

ea_8192_5%_nswap: median result of 3 min of the EA with $\mu = \lambda = 8'192$ with nswap unary operator and 5% sequence recombination



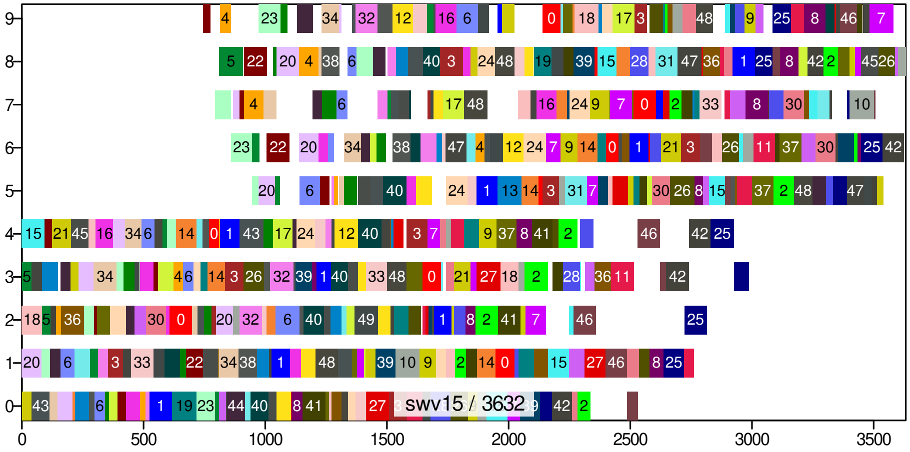
So what do we get?

`eac_4_5%_nswap`: median result of 3 min of the EA with clearing and $\mu = \lambda = 4$ with `nswap` unary operator and 5% `sequence` recombination



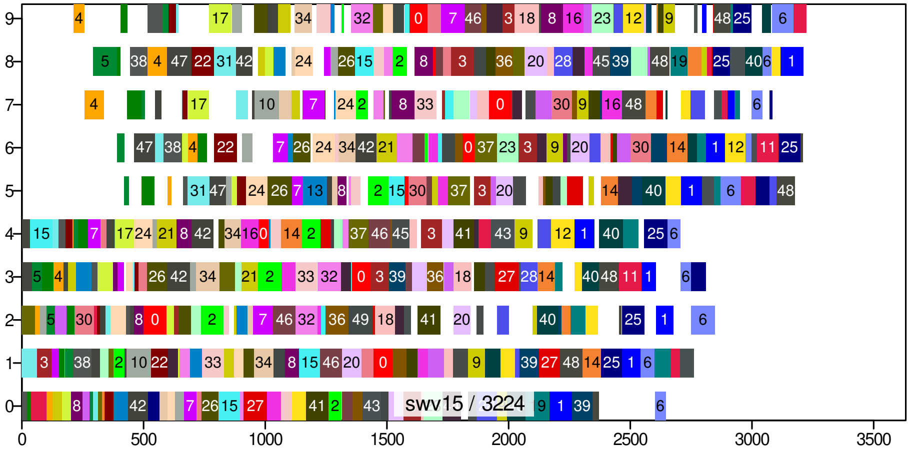
So what do we get?

ea_8192_5%_nswap: median result of 3 min of the EA with $\mu = \lambda = 8'192$ with nswap unary operator and 5% sequence recombination



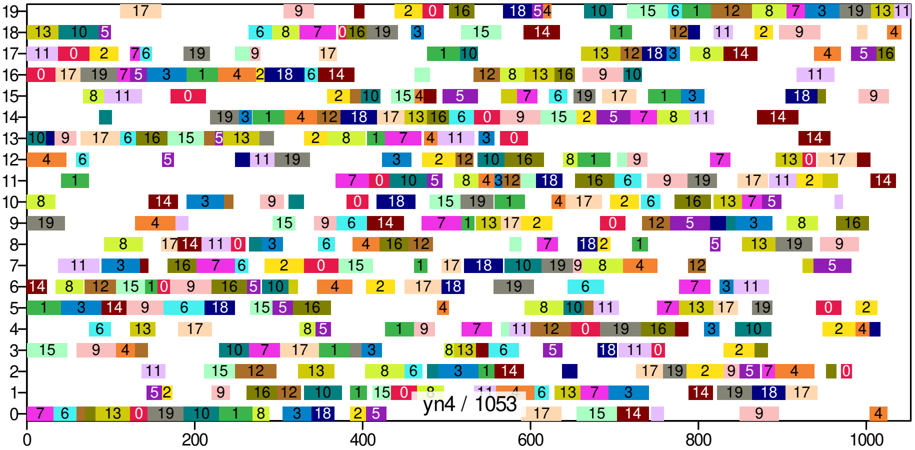
So what do we get?

`eac_4_5%_nswap`: median result of 3 min of the EA with clearing and $\mu = \lambda = 4$ with `nswap` unary operator and 5% sequence recombination



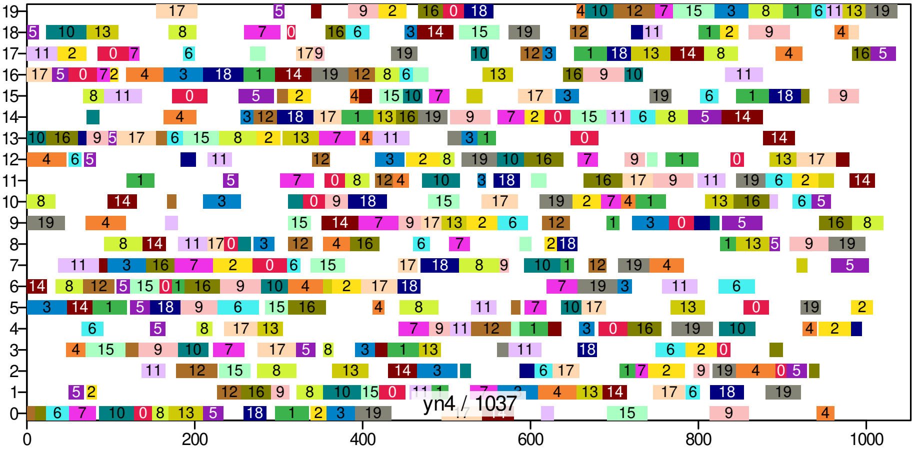
So what do we get?

`ea_8192_5%_nswap`: median result of 3 min of the EA with $\mu = \lambda = 8'192$ with `nswap` unary operator and 5% sequence recombination



So what do we get?

eac_4_5%_nswap: median result of 3 min of the EA with clearing and $\mu = \lambda = 4$ with nswap unary operator and 5% sequence recombination

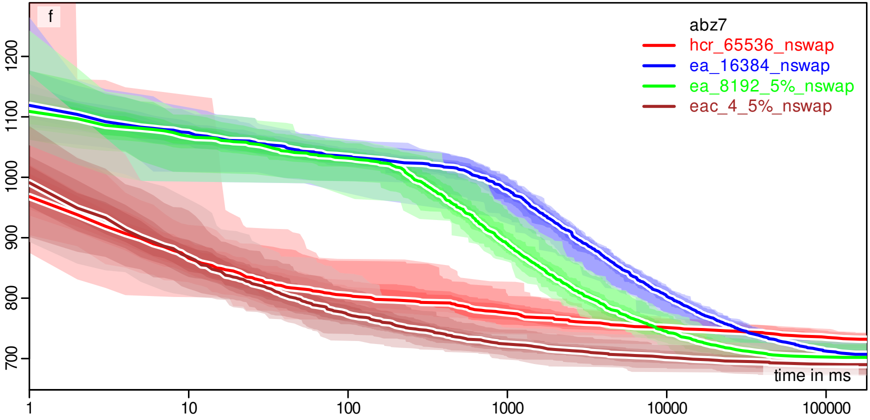


Progress over Time

What progress does the algorithm make over time?

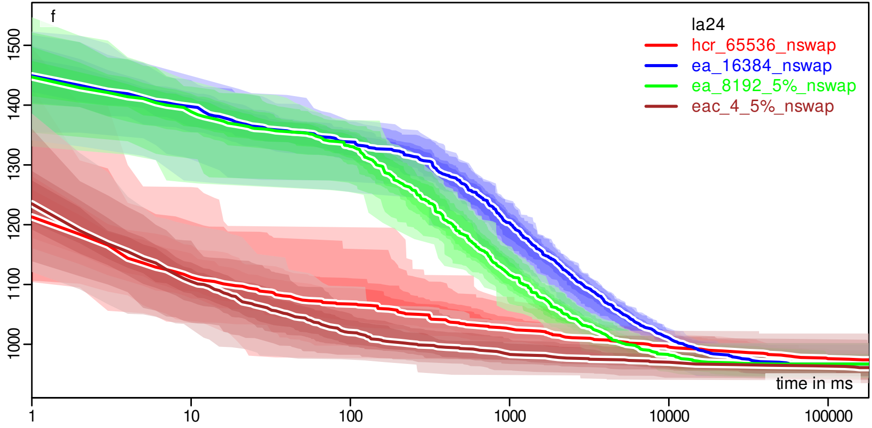
Progress over Time

What progress does the algorithm make over time?



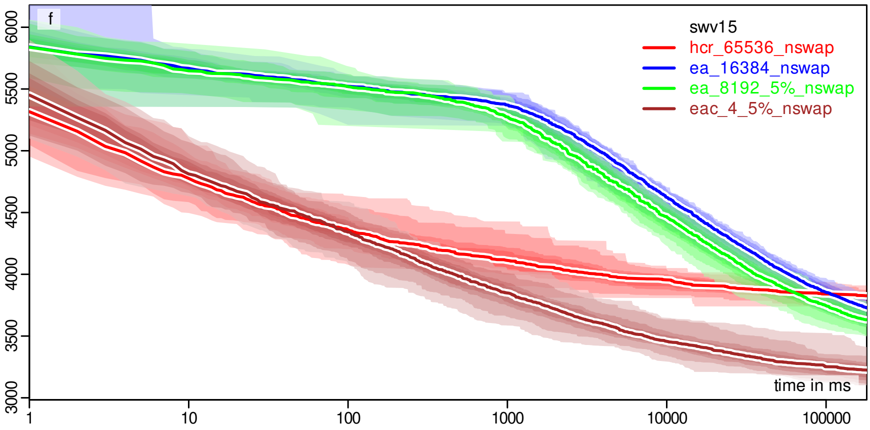
Progress over Time

What progress does the algorithm make over time?



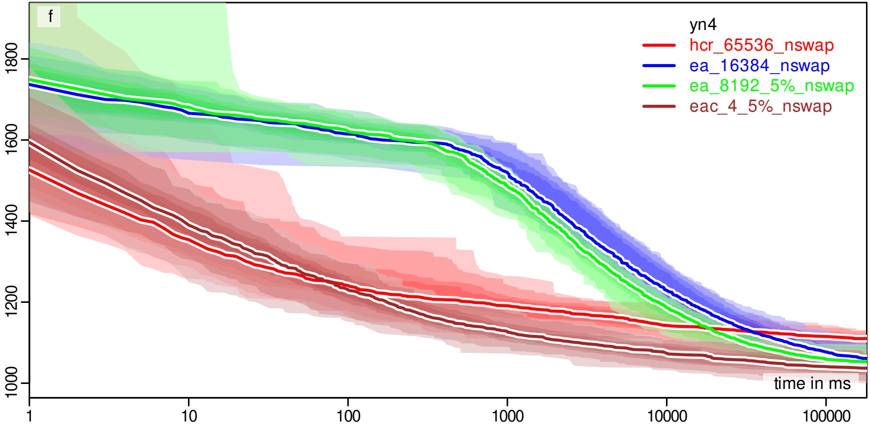
Progress over Time

What progress does the algorithm make over time?



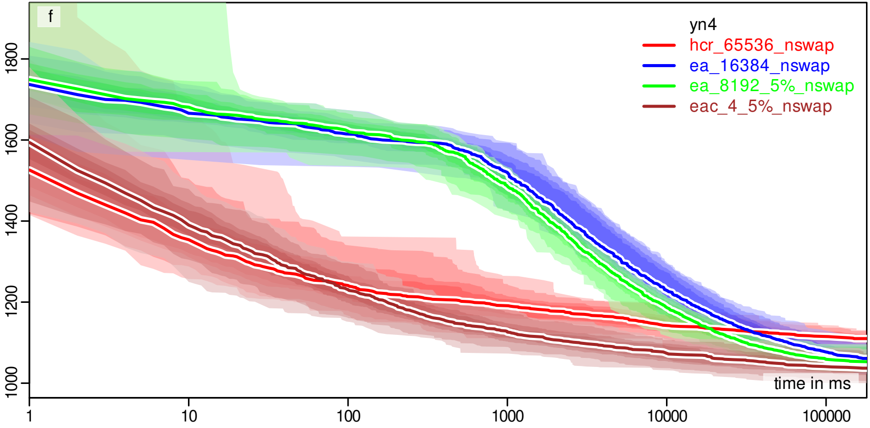
Progress over Time

What progress does the algorithm make over time?



Progress over Time

What progress does the algorithm make over time?



The EA with clearing performs much better than the EA without, at a much smaller population size.

Summary

Summary

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

Summary

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

Summary

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

Summary

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

Summary

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

Summary

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

Summary

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

Summary

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

Summary

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

Summary

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

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. John Henry Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. University of Michigan Press, Ann Arbor, MI, USA, 1975. ISBN 0-472-08460-7.
4. Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz, editors. *Handbook of Evolutionary Computation*. Computational Intelligence Library. Oxford University Press, Inc., New York, NY, USA, 1997. ISBN 0-7503-0392-1.
5. Zbigniew Michalewicz and David B. Fogel. *How to Solve It: Modern Heuristics*. Springer-Verlag, Berlin/Heidelberg, 2nd edition, 2004. ISBN 3-540-22494-7.
6. David Edward Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989. ISBN 0-201-15767-5.
7. Thomas Weise, Yuezhong Wu, Raymond Chiong, Ke Tang, and Jörg Lässig. Global versus local search: The impact of population sizes on evolutionary algorithm performance. *Journal of Global Optimization*, 66:511–534, November 2016. doi:10.1007/s10898-016-0417-5.
8. Ágoston Endre Eiben and C. A. Schippers. On evolutionary exploration and exploitation. *Fundamenta Informaticae – Annales Societatis Mathematicae Polonae, Series IV*, 35(1-2):35–50, July–August 1998. doi:10.3233/FI-1998-35123403. URL http://www.cs.vu.nl/~gusz/papers/FunInf98-Eiben-Schippers.ps.
9. 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.
10. 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.
11. Ofer M. Shir. Niching in evolutionary algorithms. In Grzegorz Rozenberg, Thomas Bäck, and Joost N. Kok, editors, *Handbook of Natural Computing*, pages 1035–1069. Springer, Berlin/Heidelberg, Germany, 2012. ISBN 978-3-540-92909-3. doi:10.1007/978-3-540-92910-9_32.
12. Matej Črepinšek, Shih-Hsi Liu, and Marjan Mernik. Exploration and exploitation in evolutionary algorithms: A survey. *ACM Computing Surveys*, 45(3):35:1–35:33, July 2013. doi:10.1145/2480741.2480752. URL http://www.researchgate.net/publication/243055445.

References II

13. Giovanni Squillero and Alberto Paolo Tonda. Divergence of character and premature convergence: A survey of methodologies for promoting diversity in evolutionary optimization. *Information Sciences*, 329:782–799, February 2016. doi:10.1016/j.ins.2015.09.056.
14. Tobias Friedrich, Nils Hebbinghaus, and Frank Neumann. Rigorous analyses of simple diversity mechanisms. In Hod Lipson, editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'07), July 7-11, 2007, London, England*, pages 1219–1225. ACM, 2007. doi:10.1145/1276958.1277194. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.217.2494&rep=rep1&type=pdf`.