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)
School of Artificial Intelligence and Big Data
Hefei University
Hefei, Anhui, China

应用优化研究所
人工智能与大数据学院
合肥学院
中国安徽省合肥市
Outline

1. Introduction
2. Algorithm Concept: Population
3. Experiment and Analysis
4. Algorithm Concept: Binary Operator
5. Experiment and Analysis
6. Algorithm Concept: Increased Diversity via Clearing
7. Experiment and Analysis
8. Summary
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 $x^*$, 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 \texttt{1swap} or \texttt{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 \( x^* \), 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\textsuperscript{2–6} try to maintain a set of points in the search space which are iteratively refined.
• Population-based metaheuristics\textsuperscript{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\textsuperscript{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.\(^2\)\(^4\)\(^5\)
Evolutionary Algorithms (EAs) are the most successful family of population-based metaheuristics.\textsuperscript{2,4,5}

Here we focus on \((\mu + \lambda)\) EAs
Evolutionary Algorithms (EAs) are the most successful family of population-based metaheuristics.\textsuperscript{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).
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 \(\mu\) best points as “parents” for the next “generation” of points, discard the remaining \(\lambda\) points.
(\(\mu + \lambda\)) \textbf{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 \(\mu\) best points as “parents” for the next “generation” of points, discard the remaining \(\lambda\) points.
  3. Generate \(\lambda\) new “offspring” points by applying a unary search operator (which creates a randomly modified copy from a selected point).
$$(\mu + \lambda) \text{ EA}$$

- Evolutionary Algorithms (EAs) are the most successful family of population-based metaheuristics.\textsuperscript{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 \(\mu\) best points as “parents” for the next “generation” of points, discard the remaining \(\lambda\) points.
  3. Generate \(\lambda\) new “offspring” points by applying a unary search operator (which creates a randomly modified copy from a selected point).
  4. Evaluate the \(\lambda\) offsprings, add them to the population, and go back to step 2..
package aitoa.structure;

public class Record<X> {  

    /** The comparator to be used for sorting according to 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...
}
package aitoa.algorithms;

public class EA<X, Y> {
    // abridged code: unnecessary stuff omitted here and in function 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) {
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
  }
}
```
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
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
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
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 of solve
}  // end class
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
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
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;) {
            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
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;) {
            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
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;) {
            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
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 class
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;) {
      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
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();
            this.nullary.apply(x, random);   // allocate point
            P[i] = new Record<>(x, process.evaluate(x));   // fill with random data
            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
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[不少].length() >= -1 instanceof X
        if (process.shouldTerminate())
            return;
    }
    // end solve
    } // end class

    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;) {
        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;) {
            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;) {
            if (process.shouldTerminate()) return;
            Record<X> dest = P[index];
        }
        // the end of the offspring generation
    }
    // end solve
} // end class
```
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;)
        {
            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;)
        {
            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
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;) {
            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;) {
            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
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;) {
            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;) {
            if (process.shouldTerminate()) return;
            Record<X> dest = P[index];
            Record<X> sel = P[p1]; // step the parent 1 index
            this.unary.apply(sel.x, dest.x, random); // generate offspring
            p1 = (p1 + 1) % this.mu;
        }
        // the end of the offspring generation
    }
    // end solve
} // end class
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;) {
            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;) {
            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
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
            if (process.shouldTerminate()) return;
        } // the end of the main loop
    } // end solve
} // end class
Experiment and Analysis
Configuring the Algorithm

- Our EA has two parameters, $\mu$ and $\lambda$. 
Configuring the Algorithm

• Our EA has two parameters, $\mu$ and $\lambda$.
• Actually, it has three parameters.
Configuring the Algorithm

• Our EA has two parameters, $\mu$ and $\lambda$.
• Actually, it has three parameters: We can choose 1swap or nswap as unary search operation.
Configuring the Algorithm

• Our EA has two parameters, $\mu$ and $\lambda$.
• Actually, it has three parameters: We can choose $1\text{swap}$ or $n\text{swap}$ 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, $\mu$ and $\lambda$.
• Actually, it has three parameters: We can choose $1\text{swap}$ or $n\text{swap}$ 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

best f / lb*

ea_μ_unary

abz7 / 656
la24 / 935
swv15 / 2885
yn4 / 929
1swap
nswap

μ=λ
Configuring the Algorithm

- Our EA has two parameters, \( \mu \) and \( \lambda \).
- Actually, it has three parameters: We can choose 1\textit{swap} or \textit{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 \textit{swv15}, a setting of \( \mu = \lambda = 16'384 \) seems reasonable.
Configuring the Algorithm

- Our EA has two parameters, $\mu$ and $\lambda$.
- 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, $\mu$ and $\lambda$.
- Actually, it has three parameters: We can choose $\text{1swap}$ or $\text{nswap}$ 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 $\text{1swap}$ and $\text{nswap}$, but we pick $\text{nswap}$ because it tends to be the better choice more often.
Configuring the Algorithm

- Our EA has two parameters, $\mu$ and $\lambda$.
- 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

<table>
<thead>
<tr>
<th>$\mathcal{I}$</th>
<th>algo</th>
<th>makespan</th>
<th>last improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>best</td>
<td>mean</td>
</tr>
<tr>
<td>abz7</td>
<td>hcr_65536_nswap</td>
<td>712</td>
<td>731</td>
</tr>
<tr>
<td></td>
<td>ea_16384_nswap</td>
<td>691</td>
<td>707</td>
</tr>
<tr>
<td>la24</td>
<td>hcr_65536_nswap</td>
<td>942</td>
<td>973</td>
</tr>
<tr>
<td></td>
<td>ea_16384_nswap</td>
<td>945</td>
<td>968</td>
</tr>
<tr>
<td>swv15</td>
<td>hcr_65536_nswap</td>
<td>3740</td>
<td>3818</td>
</tr>
<tr>
<td></td>
<td>ea_16384_nswap</td>
<td>3577</td>
<td>3723</td>
</tr>
<tr>
<td>yn4</td>
<td>hcr_65536_nswap</td>
<td>1068</td>
<td>1109</td>
</tr>
<tr>
<td></td>
<td>ea_16384_nswap</td>
<td>1022</td>
<td>1063</td>
</tr>
</tbody>
</table>
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

The diagram shows the median result of the restarted hill climber with $L = 65'536$ and nswap for various values of nswap from 0 to 3500.
So what do we get?

ea_{16384\_nswap}: median result of 3 min of the EA with $\mu = \lambda = 16'384$ with 

\textit{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.
Progress over Time

What progress does the algorithm make 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.
However, on the linear time scale we can see that it keeps improving slowly but surely during all the time.
• 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
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.
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.
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 \(\mu + \lambda \rightarrow +\infty\), 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 \(\eta\) large enough to completely exhaust our computational budget (here: 3 min), the EA is a random sampling algorithm.
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 \(\eta\) 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 $\mu$ 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 $\mu$ 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 $\mu$ 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 $\mu$ basically allows us to “tune” between these two behaviors.

• 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 $\mu$ 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 $\mu$ basically allows us to “tune” between these two behaviors.
- 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 $\mu$, 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 $\mu$ basically allows us to “tune” between these two behaviors.

• 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 $\mu$, 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 $\mu$ 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 $\mu$ 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 $\mu$, 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 $\mu$ 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.”
• We now have more than one candidate solution in our “population.”
• But we only use one existing point from $X$ 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 $X$ 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 $X$ 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 $X$ 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...
We now have more than one candidate solution in our “population.”

But we only use one existing point from \( X \) as “blueprint” when we create a new one.

Why can’t we use one 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.
(\(\mu + \lambda\)) 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).
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 $\mu$ best points as “parents” for the next “generation,” discard the remaining $\lambda$ points.
The \((\mu + \lambda)\) EAs with recombination work as follows:

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

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

1. Generate a population of \(\mu + \lambda\) random points in the search space (and map them to solutions and evaluate them).
2. From the complete population, select the \(\mu\) best points as “parents” for the next “generation,” discard the remaining \(\lambda\) points.
3. Generate \(\lambda\) 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 \(\mu\) best points as "parents" for the next "generation," discard the remaining \(\lambda\) points.
  3. Generate \(\lambda\) 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.
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 \(\mu\) best points as “parents” for the next “generation,” discard the remaining \(\lambda\) points.
3. Generate \(\lambda\) 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 \(\lambda\) offsprings, add them to the population, and go back to step 2.
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;) {
            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;
        }

        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;) {
                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();
            this.nullary.apply(x, random); // allocate point
            P[i] = new Record<>(x, process.evaluate(x)); // evaluate
            if (process.shouldTerminate()) return;
        }

        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;) {
                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
        }
    }
}
```
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;) {
            X x = searchSpace.create();
            this.nullary.apply(x, random); // allocate point
            P[i] = new Record<>(x, process.evaluate(x)); // evaluate
        }

        for (;;) {
            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;) {
                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[X][this.mu + this.lambda];

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

        for (;;) {
            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;) {
                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

```
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
```
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.
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 $x_1$ or $x_2$ 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 $x_1$ or $x_2$ 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 \( x_1 \) or \( x_2 \) 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 $x_1$ or $x_2$ 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 $i = n \times m$, 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 $x_1$ or $x_2$ 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 $i = n \ast m$, then all operations have been assigned. We exit and returning $x'$.
   3.6 Mark the first unmarked occurrence of $J$ as “already assigned” in $x_1$. 
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 $x_1$ or $x_2$ 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 $i = n \times m$, then all operations have been assigned. We exit and returning $x'$.
   3.6 Mark the first unmarked occurrence of $J$ as “already assigned” in $x_1$.
   3.7 Mark the first unmarked occurrence of $J$ as “already assigned” in $x_2$. 

Example for Sequence Recombination

\[ x_1 = (2,0,3,1,1,1,0,0,0,3,0,2,1,1,3,2,2,3,2,3) \quad x_2 = (3,1,1,2,0,2,1,2,2,1,0,3,1,0,0,3,2,3,3,0) \]
Example for Sequence Recombination

\[ f(y_1) = 202 \]
\[ f(y_2) = 182 \]

\[ x_1 = (2, 0, 3, 1, 1, 1, 0, 0, 0, 3, 0, 2, 1, 1, 3, 2, 2, 3, 2, 3) \]

\[ x_2 = (3, 1, 1, 2, 0, 2, 1, 2, 2, 1, 0, 3, 1, 0, 0, 3, 2, 3, 3, 0) \]
Example for Sequence Recombination

\[ x' = (2, 0, 3, 1, 1, 0, 2, 2, 2, 0, 1, 3, 1, 0, 0, 3, 3, 2, 3) \]

\[ f(y_1) = 202 \]

\[ f(y_2) = 182 \]

Sub-jobs are picked in a random sequence from both parents.
Example for Sequence Recombination

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

\(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, 3, 0)

\(f(y_1) = 202\)

\(f(y_2) = 182\)

random sequence in which the sub-jobs are picked:

\(x_1\)
Example for Sequence Recombination

\[ x' = (2, 0, 3, 1, 1, 1, 0, 0, 0, 3, 0, 2, 1, 1, 3, 2, 2, 3, 2, 3) \]

\[ f(y_1) = 202 \]

\[ f(y_2) = 182 \]

\[ x_1 = (2, 0, 3, 1, 1, 1, 0, 0, 0, 3, 0, 2, 1, 1, 3, 2, 2, 3, 2, 3) \]

\[ x_2 = (3, 1, 1, 2, 0, 2, 1, 2, 2, 1, 0, 3, 1, 0, 0, 3, 2, 3, 3, 0) \]

random sequence in which the sub-jobs are picked:

\[ x_1, x_1 \]
Example for Sequence Recombination

\[ x' = (2, 0, 3, 1, 1, 1, 0, 0, 0, 3, 0, 2, 1, 1, 3, 2, 2, 3, 2, 3) \]

\[ f(y_1) = 202 \]

\[ x_1 = (2, 0, 3, 1, 1, 1, 0, 0, 0, 3, 0, 2, 1, 1, 3, 2, 2, 3, 2, 3) \]

\[ x_2 = (3, 1, 1, 2, 0, 2, 1, 2, 2, 1, 0, 3, 1, 0, 0, 3, 2, 3, 3, 0) \]

\[ f(y_2) = 182 \]

random sequence in which the sub-jobs are picked:

\[ x_1, x_1, x_1 \]
Example for Sequence Recombination

\[ x' = (2, 0, 3, 1, 1, 1, 0, 0, 0, 3, 0, 2, 1, 1, 3, 2, 2, 3, 2, 3) \]

\[ x_1 = (2, 0, 3, 1, 1, 1, 0, 0, 0, 3, 0, 2, 1, 1, 3, 2, 2, 3, 2, 3) \]

\[ x_2 = (3, 1, 2, 0, 2, 1, 2, 2, 1, 0, 3, 1, 0, 0, 3, 2, 3, 3, 0) \]

\[ f(y_1) = 202 \]

\[ f(y_2) = 182 \]

random sequence in which the sub-jobs are picked:

\[ x_1, x_1, x_1, x_2 \]
Example for Sequence Recombination

\[ x' = (2, 0, 3, 1, 1, 1, 1, 0, 0, 0, 3, 0, 2, 1, 1, 3, 2, 2, 3, 2, 3) \]

\[ x_1 = (2, 0, 3, 1, 1, 1, 0, 0, 0, 3, 0, 2, 1, 1, 3, 2, 2, 3, 2, 3) \]

\[ x_2 = (3, 1, 1, 2, 0, 2, 1, 2, 2, 1, 0, 3, 1, 0, 0, 3, 2, 3, 3, 0) \]

\[ f(y_1) = 202 \]
\[ f(y_2) = 182 \]

(random sequence in which the sub-jobs are picked: \( x_1, x_1, x_1, x_2, x_1 \))
Example for Sequence Recombination

\[ x' = (2, 0, 3, 1, 1, 0, 0, 0, 3, 0, 2, 1, 1, 3, 2, 2, 3, 2, 3) \]

\[ x_1 = (2, 0, 3, 1, 1, 0, 0, 0, 3, 0, 2, 1, 1, 3, 2, 2, 3, 2, 3) \]

\[ x_2 = (3, 1, 1, 2, 0, 2, 1, 2, 2, 1, 0, 3, 1, 0, 0, 3, 0, 2, 3, 3, 0) \]

\[ f(y_1) = 202 \]

\[ f(y_2) = 182 \]
Example for Sequence Recombination

\[ x'=(2,0,3,1,1,1,0,2,2,2,0,1,3,1,0,0,3,3,2,3) \]

random sequence in which the sub-jobs are picked:

- x1, x1, x1, x2, x1, x1, x1, x1
Example for Sequence Recombination

$f(y_1) = 202$

\[ x_1 = (2, 0, 3, 1, 1, 0, 0, 0, 3, 0, 2, 1, 1, 3, 2, 2, 3, 2) \]

$f(y_2) = 182$

\[ x_2 = (3, 1, 1, 2, 0, 2, 1, 2, 2, 1, 0, 3, 1, 0, 0, 3, 2, 3, 3) \]

\[ x' = (2, 0, 3, 1, 1, 0, 2, 2, 2, 0, 1, 3, 1, 0, 0, 3, 3, 2) \]

random sequence in which the sub-jobs are picked:
- $x_1, x_1, x_1, x_2, x_1, x_1, x_1, x_1, x_1, x_2$
Example for Sequence Recombination

\[ x' = (2, 0, 3, 1, 1, 0, 2, 2, 2, 0, 1, 3, 1, 0, 0, 3, 3, 2, 3) \]

\[ f(y_1) = 202 \]

\[ f(y_2) = 182 \]

Random sequence in which the sub-jobs are picked:

\[ x_1, x_1, x_1, x_2, x_1, x_1, x_1, x_2, x_2 \]
Example for Sequence Recombination

\[ x' = (2, 0, 3, 1, 1, 0, 2, 2, 2, 0, 1, 3, 1, 0, 0, 3, 3, 2, 3) \]

\[ f(y_1) = 202 \]

\[ f(y_2) = 182 \]

random sequence in which the sub-jobs are picked:

\[ x_1, x_1, x_1, x_2, x_1, x_1, x_1, x_2, x_2, x_2 \]
Example for Sequence Recombination

\[ x' = (2, 0, 3, 1, 1, 0, 2, 2, 2, 0, 1, 3, 1, 0, 0, 3, 3, 2, 3) \]

random sequence in which the sub-jobs are picked:

\[ x_1, x_1, x_1, x_2, x_1, x_1, x_1, x_2, x_2, x_2, x_1 \]
Example for Sequence Recombination

\[ x' = (2, 0, 3, 1, 1, 0, 0, 0, 0, 3, 0, 2, 1, 1, 3, 2, 2, 3, 2, 3) \]

\[ x = (3, 1, 1, 2, 0, 2, 1, 2, 2, 1, 0, 3, 1, 3, 0, 0, 3, 2, 3, 3) \]

\[ f(y_1) = 202 \]
\[ f(y_2) = 182 \]
Example for Sequence Recombination

\[ x' = (2, 0, 3, 1, 1, 0, 2, 2, 0, 1, 3, 1, 0, 0, 3, 3, 2, 3) \]

random sequence in which the sub-jobs are picked:

\[ x_1, x_1, x_1, x_2, x_1, x_1, x_1, x_1, x_2, x_2, x_2, x_1, x_2, x_2, x_2, x_2 \]
Example for Sequence Recombination

\[ x' = (2, 0, 3, 1, 1, 0, 2, 2, 0, 1, 3, 1, 0, 0, 3, 3, 2, 3) \]

\[ x_1 = (2, 0, 3, 1, 1, 0, 0, 0, 3, 0, 2, 1, 1, 3, 2, 2, 3, 2, 3) \]

\[ x_2 = (3, 1, 1, 2, 0, 2, 1, 2, 2, 1, 0, 3, 1, 0, 0, 3, 2, 3, 3, 0) \]

\[ f(y_1) = 202 \]

\[ f(y_2) = 182 \]
Example for Sequence Recombination

\[ x_1 = (2, 0, 3, 1, 1, 1, 0, 0, 0, 0, 3, 0, 2, 1, 3, 2, 2, 3, 2, 3) \]

\[ x_2 = (3, 1, 1, 2, 0, 2, 1, 2, 2, 1, 0, 3, 1, 0, 0, 3, 2, 3, 3, 0) \]

\[ f(y_1) = 202 \]

\[ f(y_2) = 182 \]

\[ x' = (2, 0, 3, 1, 1, 1, 0, 2, 2, 2, 0, 1, 3, 1, 0, 0, 3, 3, 2, 3) \]
Example for Sequence Recombination

\[ x' = (2, 0, 3, 1, 1, 1, 0, 2, 2, 2, 0, 1, 3, 1, 0, 0, 3, 3, 2, 3, 3) \]

\[ f(y_1) = 202 \]

\[ f(y_2) = 182 \]

\[ x_1 = (2, 0, 3, 1, 1, 1, 0, 0, 0, 3, 0, 2, 1, 1, 3, 2, 2, 3, 2, 3) \]

\[ x_2 = (3, 1, 1, 2, 0, 2, 1, 2, 1, 2, 1, 0, 3, 1, 0, 0, 3, 2, 3, 3, 0) \]

random sequence in which the sub-jobs are picked:

X1, X1, X1, X2, X1, X1, X1, X2, X2, X2, X2, X1, X2, X2, X2, X1, X1
Example for Sequence Recombination

\[ x' = (2, 0, 3, 1, 1, 0, 2, 2, 2, 0, 1, 3, 1, 0, 0, 3, 3, 2, 3) \]

\[ f(y_1) = 202 \]

\[ f(y_2) = 182 \]

\[ x_1 = (2, 0, 3, 1, 1, 0, 0, 0, 3, 0, 2, 1, 1, 3, 2, 2, 3, 2, 3) \]
\[ x_2 = (3, 1, 1, 2, 0, 2, 1, 2, 1, 0, 3, 1, 0, 0, 3, 2, 3, 3, 0) \]

random sequence in which the sub-jobs are picked:

\[ x_1, x_1, x_1, x_2, x_1, x_1, x_1, x_2, x_2, x_2, x_2, x_1, x_1, x_1, x_2, x_2, x_1, x_1, x_1 \]
**Example for Sequence Recombination**

\[ x' = (2,0,3,1,1,0,0,0,0,3,0,2,1,1,3,2,2,3,2,3) \]

\[ x_1 = (2,0,3,1,1,0,0,0,0,3,0,2,1,1,3,2,2,3,2,3) \]

\[ x_2 = (3,1,1,2,0,2,1,2,2,1,0,3,1,0,0,3,2,3,3,0) \]

\[ f(y_1) = 202 \]

\[ f(y_2) = 182 \]
Example for Sequence Recombination

\[ f(y_1) = 202 \]
\[ x' = (2, 0, 3, 1, 1, 1, 0, 2, 2, 2, 0, 1, 3, 1, 0, 3, 3, 2, 3) \]

\[ f(y_2) = 182 \]
\[ x_2 = (3, 1, 1, 2, 0, 2, 1, 2, 2, 1, 0, 3, 1, 0, 3, 2, 3, 3, 0) \]

Random sequence in which the sub-jobs are picked:
\[ x_1, x_1, x_1, x_2, x_1, x_1, x_1, x_2, x_2, x_2, x_1, x_2, x_2, x_2, x_1, x_1, x_1, x_2, x_2 \]
Example for Sequence Recombination

\[ x' = (2, 0, 3, 1, 1, 0, 2, 2, 2, 0, 1, 3, 1, 0, 0, 3, 3, 2, 3) \]

\[ f(y_1) = 202 \]

\[ x_1 = (2, 0, 3, 1, 1, 0, 0, 0, 3, 0, 2, 1, 1, 3, 2, 2, 3, 2, 3) \]

\[ f(y_2) = 182 \]

\[ x_2 = (3, 1, 1, 2, 0, 2, 1, 2, 2, 1, 0, 3, 1, 0, 0, 3, 2, 3, 3, 0) \]

Random sequence in which the sub-jobs are picked:

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

\[ x' = (2, 0, 3, 1, 1, 0, 2, 2, 2, 0, 1, 3, 1, 0, 0, 3, 3, 2, 3) \]

\[ f(y') = 192 \]
Example for Sequence Recombination

\[ x' = (2, 0, 3, 1, 1, 0, 2, 2, 2, 0, 1, 3, 1, 0, 0, 3, 3, 2, 3) \]

\[ f(y') = 192 \]

\[ x = (2, 0, 3, 1, 1, 0, 0, 0, 3, 0, 2, 1, 1, 3, 2, 2, 3, 2, 3) \]

\[ f(y_1) = 202 \]

\[ x_1 = (2, 0, 3, 1, 1, 0, 0, 0, 3, 0, 2, 1, 1, 3, 2, 2, 3, 2, 3) \]

\[ x_2 = (3, 1, 1, 2, 0, 2, 1, 2, 2, 1, 0, 3, 1, 0, 0, 3, 2, 3, 3, 0) \]

\[ f(y_2) = 182 \]

random sequence in which the sub-jobs are picked:

\[ x_1, x_1, x_1, x_2, x_1, x_1, x_1, x_1, x_2, x_2, x_1, x_1, x_1, x_1, x_2, x_1, x_1 \]
Implementing Sequence Recombination

```java
package aitoa.examples.jssp;

public class JSSPBinaryOperatorSequence {
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
}
```
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) {
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
```
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

```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];
        boolean[] doneX1 = new boolean[x0.length];
    }
}
```
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 reusable
        boolean[] doneX1 = new boolean[x0.length]; // member variable instead
    }
}
} // 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
    }
}
```
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 reusable
        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

```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
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //

    } // end of function
}
```

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 reusable
        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 reusable
        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
    }
}
```
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 reusable
        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

```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 reusable
        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
            }
        }
    }
}
```
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++ ) {
            if ((x0[i] == add) && (!doneX0[i])) {
                doneX0[i] = true; // found it and marked it
                break; // quit operation finding loop
            }
        }
    }
}
```
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 reusable
        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
    }
}
```
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

        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 reusable
        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 < length; i++) {
            if ((x0[i] == add) && (!doneX0[i])) {
                doneX0[i] = true; // found it and marked it
                break; // quit operation finding loop
            }
        }
        while (doneX0[x0i]) x0i++;

        for (int i = x1i++; i < length; i++) {
            if ((x1[i] == add) && (!doneX1[i])) {
                doneX1[i] = true; // found it and marked it
                break; // quit operation finding loop
            }
        }
    }
}
```
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 reusable
        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

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 reusable
        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 reusable
        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 $\mu$. 
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 $\mu, \lambda$. 
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 $\mu, \lambda$, 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 $\mu, \lambda$, 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 $\mu, \lambda$, 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 $\mu$, $\lambda$, the unary operator, the binary operator, and the crossover rate $cr$.
- Let’s stick with $\mu = \lambda$, $\text{nswap}$, 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 $\mu$, $\lambda$, 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 $\lambda$ and $cr$. 
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 $\mu$, $\lambda$, 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 $\lambda$ and $cr$.

The improvements that the binary operator offered us in this scenario are quite small.
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 $\mu$, $\lambda$, 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 $\lambda$ 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

<table>
<thead>
<tr>
<th>I</th>
<th>algo</th>
<th>makespan</th>
<th>last improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>best</td>
<td>mean</td>
<td>med</td>
</tr>
<tr>
<td>abz7</td>
<td>hcr_65536_nswap</td>
<td>712</td>
<td>731</td>
</tr>
<tr>
<td></td>
<td>ea_16384_nswap</td>
<td>691</td>
<td>707</td>
</tr>
<tr>
<td></td>
<td>ea_8192_5%_nswap</td>
<td>684</td>
<td>703</td>
</tr>
<tr>
<td>la24</td>
<td>hcr_65536_nswap</td>
<td>942</td>
<td>973</td>
</tr>
<tr>
<td></td>
<td>ea_16384_nswap</td>
<td>945</td>
<td>968</td>
</tr>
<tr>
<td></td>
<td>ea_8192_5%_nswap</td>
<td>943</td>
<td>967</td>
</tr>
<tr>
<td>swv15</td>
<td>hcr_65536_nswap</td>
<td>3740</td>
<td>3818</td>
</tr>
<tr>
<td></td>
<td>ea_16384_nswap</td>
<td>3577</td>
<td>3723</td>
</tr>
<tr>
<td></td>
<td>ea_8192_5%_nswap</td>
<td>3498</td>
<td>3631</td>
</tr>
<tr>
<td>yn4</td>
<td>hcr_65536_nswap</td>
<td>1068</td>
<td>1109</td>
</tr>
<tr>
<td></td>
<td>ea_16384_nswap</td>
<td>1022</td>
<td>1063</td>
</tr>
<tr>
<td></td>
<td>ea_8192_5%_nswap</td>
<td>1026</td>
<td>1056</td>
</tr>
</tbody>
</table>
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?

\text{ea\_16384\_nswap: median result of 3 min of the EA with } \mu = \lambda = 16'384 \text{ 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 $\mu = \lambda = 16^{\prime}384$ 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

<table>
<thead>
<tr>
<th>0</th>
<th>500</th>
<th>1000</th>
<th>1500</th>
<th>2000</th>
<th>2500</th>
<th>3000</th>
<th>3500</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>6</td>
<td>1</td>
<td>19</td>
<td>23</td>
<td>44</td>
<td>40</td>
<td>8</td>
</tr>
<tr>
<td>27</td>
<td>3</td>
<td>swv15 / 3632</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

yn4 / 1061
So what do we get?

e

\[\text{ea}_{8192}\_5\%\_\text{nswap}: \text{median result of 3 min of the EA with } \mu = \lambda = 8'192 \text{ with nswap unary operator and 5\% sequence recombination}\]
Progress over Time

What progress does the algorithm make over time?
Progress over Time

What progress does the algorithm make over time?

![Graph showing progress over time for different algorithms and conditions.](image)
Progress over Time

What progress does the algorithm make over time?

[Graph showing progress over time with different algorithms and labels: la24, hcr_65536_nswap, ea_16384_nswap, ea_8192_nswap, ea_8192_5%_nswap, time in ms]
Progress over Time

What progress does the algorithm make over time?

- swv15
- hcr_65536_nswap
- ea_16384_nswap
- ea_8192_nswap
- ea_8192_5%_nswap

(time in ms)
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.
• 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.\textsuperscript{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: \textit{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\textsuperscript{11, 14}.
- Furthermore, we will apply the simplest version of this approach.
- Every time, when $\mu$ out of the $\mu + \lambda$ records are selected, one prior step is applied.
• We will here consider a very simple approach to preserve population diversity: clearing\textsuperscript{11,14}.
• Furthermore, we will apply the simplest version of this approach.
• Every time, when $\mu$ 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\textsuperscript{11,14}.
- Furthermore, we will apply the simplest version of this approach.
- Every time, when $\mu$ 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.
package aitoa.algorithms;

public class Utils {
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
package aitoa.algorithms;

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

      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
   }
}

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;
    }
}
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

      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
   }
}
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++) {
            //
            //
            //
            //
            //
            //
            //
            //
            //
            //
            //
            //
            //
            //
            //
            //
        }
    }
}
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];
    }
  }
}
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;
    }
  }
}
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
            //
            //
            //
            //
            //
            //
            //
            //
            //
            //
            }
        }
    }
}
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?
               //
               //
               //
               //
               //
               //
               //
               //
               //
            }
         }
      }
   }
}
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];
                    //
                    //
                    //
                    //
                    //
                    //
                }
            }
        }
    }
}
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
            }
         }
      }
   }
}
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;
                }
            }
        }
    }
}
package aitoa.algorithms;

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
        //
        //
        //
      }
    }
}
}
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?
                //
                //
                //
            }
        }
    }
}
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
            }
        }
    }
}

//
//}
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
         }
      }
      return unique; // return number of unique: 1<=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

```java
package aitoa.algorithms;

public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
```
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) {
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //

    } // 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();
    }
}
```
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 class

} // end class
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 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
    }
}
```

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

            // end of filling the first population
        }
    }

} // end class
```
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
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 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

    } // end solve

} // end class
```
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
            this.lambda.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 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; // evaluate
        } // 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

```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;) {
            X x = searchSpace.create();
            this.nullary.apply(x, random); // 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
```
```
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
    }
}
```
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
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
      //
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
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;) {
            X x = searchSpace.create();
            this.nullary.apply(x, random); // allocate point
            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;) {
            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
def package aitoa.algorithms;

def public class EAWithClearing<X, Y> extends Metaheuristic2<X, Y> {
    def 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;) {
            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;) {
            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
```
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;) {
            X x = searchSpace.create();
            this.nullary.apply(x, random); // allocate point
            P[i] = new Record<>(x, process.evaluate(x)); // evaluate
            if (process.shouldTerminate()) return;
        }

        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;) {
            if (process.shouldTerminate()) return;
            Record<X> dest = P[index];
            p1 = (p1 + 1) % u; // step the parent 1 index
            Record<X> sel = P[p1];
        }
    }
}

} // 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;) {
      X x = searchSpace.create();
      this.nullary.apply(x, random); // allocate point
      P[i] = new Record<>(x, process.evaluate(x)); // evaluate
    }
    if (process.shouldTerminate()) return;

    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;) {
      if (process.shouldTerminate()) return;
      Record<X> dest = P[index];
      Record<X> sel = P[(p1 + 1) % u]; // step the parent 1
      if (random.nextDouble() <= this.cr) { // crossover!
        //
        //
        //
        //
        //
      }
    }
  }
}
```
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;) {
            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;) {
            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;) {
            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;) {
            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

```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;) {
            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;) {
            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 {
                    int 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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>makespan</th>
<th>last improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>best</td>
<td>mean</td>
</tr>
<tr>
<td>I</td>
<td>algo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>abz7</td>
<td>ea_8192_5%_nswap</td>
<td>684</td>
<td>703</td>
</tr>
<tr>
<td></td>
<td>eac_4_5%_nswap</td>
<td>672</td>
<td>690</td>
</tr>
<tr>
<td>la24</td>
<td>ea_8192_5%_nswap</td>
<td>943</td>
<td>967</td>
</tr>
<tr>
<td></td>
<td>eac_4_5%_nswap</td>
<td>935</td>
<td>963</td>
</tr>
<tr>
<td>swv15</td>
<td>ea_8192_5%_nswap</td>
<td>3498</td>
<td>3631</td>
</tr>
<tr>
<td></td>
<td>eac_4_5%_nswap</td>
<td>3102</td>
<td>3220</td>
</tr>
<tr>
<td>yn4</td>
<td>ea_8192_5%_nswap</td>
<td>1026</td>
<td>1056</td>
</tr>
<tr>
<td></td>
<td>eac_4_5%_nswap</td>
<td>1000</td>
<td>1038</td>
</tr>
</tbody>
</table>
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?

\texttt{ea\_8192\_5\%\_nswap}: median result of 3 min of the EA with $\mu = \lambda = 8'192$ with \texttt{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

---

The diagram shows a sequence of elements with a visual representation of sequence recombination. The elements are colored and positioned along a linear scale, indicating the outcome of the EA process under specified parameters.
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?
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
