Optimization Algorithms

5. Stochastic Hill Climbing

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

Institute of Applied Optimization (IAO)
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
3. Ingredient: Unary Search Operator
4. Experiment and Analysis
5. Improved Algorithm Concept 1
6. Experiment and Analysis
7. Improved Algorithm Concept 2
8. Experiment and Analysis
9. Improved Algorithm Concept 3
10. Experiment and Analysis
11. Summary
Introduction
Information from Good Solutions

• Our first algorithm, random sampling, was not very efficient.
Information from Good Solutions

• Our first algorithm, random sampling, was not very efficient.
• It does not make any use of the information it “sees” during the optimization process.
Information from Good Solutions

• Our first algorithm, random sampling, was not very efficient.
• It does not make any use of the information it “sees” during the optimization process.
• Each search step consists of creating an entirely new, entirely random candidate solution.
Information from Good Solutions

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

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

It does not make any use of the information it “sees” during the optimization process.

Each search step consists of creating an entirely new, entirely random candidate solution.

Each search step is thus independent of all prior steps.

Is this a good idea?

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

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

• So how we can make use of the information we have seen during the search?
Basic Idea

• So how we can make use of the information we have seen during the search?
Basic Idea

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

• So how we can make use of the information we have seen during the search?
• Instead of generating a completely random new candidate solution in each step...
• ...why can’t we try to iteratively improve the best solution we have seen so far?
Algorithm Concept
Stochastic Hill Climbing

- This is the concept of Local Search\textsuperscript{2–5} and its simplest realization is Stochastic Hill Climbing\textsuperscript{2}.
Stochastic Hill Climbing

- This is the concept of Local Search\(^2-5\) and its simplest realization is Stochastic Hill Climbing\(^2\).
- Simple Concept
Stochastic Hill Climbing

• This is the concept of Local Search\textsuperscript{2–5} and its simplest realization is Stochastic Hill Climbing\textsuperscript{2}.

• Simple Concept:
  1. create random initial solution
Stochastic Hill Climbing

• This is the concept of Local Search\textsuperscript{2–5} and its simplest realization is Stochastic Hill Climbing\textsuperscript{2}.

• Simple Concept:
  1. create random initial solution
  2. make a modified copy of best-so-far solution
Stochastic Hill Climbing

• This is the concept of **Local Search**\(^2\text{–}^5\) and its simplest realization is **Stochastic Hill Climbing**\(^2\).

• Simple Concept:
  1. create random initial solution
  2. make a modified copy of best-so-far solution
  3. if it is better, it becomes the new best-so-far solution (if it is not better, discard it).
Stochastic Hill Climbing

• This is the concept of Local Search\textsuperscript{2–5} and its simplest realization is Stochastic Hill Climbing\textsuperscript{2}.

• Simple Concept:
  1. create random initial solution
  2. make a modified copy of best-so-far solution
  3. if it is better, it becomes the new best-so-far solution (if it is not better, discard it).
  4. go back to 2. (until the time is up)
Implementation of the Stochastic Hill Climber

```java
package aitoa.algorithms;

public class HillClimber<X, Y> {
    // unnecessary stuff omitted here...
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
}
```
package aitoa.algorithms;

public class HillClimber<X, Y> extends Metaheuristic1<X, Y> {
    // unnecessary stuff omitted here...
    public void solve(IBlackBoxProcess<X, Y> process) {
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
    }
}
Implementation of the Stochastic Hill Climber

```java
package aitoa.algorithms;

public class HillClimber<X, Y> extends Metaheuristic1<X, Y> {
    // unnecessary stuff omitted here...
    public void solve(IBlackBoxProcess<X, Y> process) {
        X xCur = process.getSearchSpace().create();
    }
}
```
Implementation of the Stochastic Hill Climber

```java
package aitoa.algorithms;

public class HillClimber<X, Y> extends Metaheuristic1<X, Y> {
    // unnecessary stuff omitted here...
    public void solve(IBlackBoxProcess<X, Y> process) {
        X xCur = process.getSearchSpace().create();
        X xBest = process.getSearchSpace().create();
    }
}
```
Implementation of the Stochastic Hill Climber

```java
package aitoa.algorithms;

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

```java
package aitoa.algorithms;

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

        this.nullary.apply(xBest, random);
    }
}
```
package aitoa.algorithms;

public class HillClimber<X, Y> extends Metaheuristic1<X, Y> {

    // unnecessary stuff omitted here...

    public void solve(IBlackBoxProcess<X, Y> process) {
        X xCur = process.getSearchSpace().create();
        X xBest = process.getSearchSpace().create();
        Random random = process.getRandom();

        this.nullary.apply(xBest, random);
        double fBest = process.evaluate(xBest);
    }
}
Implementation of the Stochastic Hill Climber

```java
package aitoa.algorithms;

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

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

        //
        this.unary.apply(xBest, xCur, random);
        //
        //
        //
        //
        //
        //
        //
        //
    }
}
```
Implementation of the Stochastic Hill Climber

```java
package aitoa.algorithms;

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

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

        this.unary.apply(xBest, xCur, random);
        double fCur = process.evaluate(xCur);
    }
}
```
Implementation of the Stochastic Hill Climber

```java
package aitoa.algorithms;

public class HillClimber<X, Y> extends Metaheuristic1<X, Y> {
   // unnecessary stuff omitted here...

   public void solve(IBlackBoxProcess<X, Y> process) {
      X xCur = process.getSearchSpace().create();
      X xBest = process.getSearchSpace().create();
      Random random = process.getRandom();

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

      this.unary.apply(xBest, xCur, random);
      double fCur = process.evaluate(xCur);
      if (fCur < fBest) {
         //
         //
         //
         //
         //
         //
      }
}
```
Implementation of the Stochastic Hill Climber

```java
package aitoa.algorithms;

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

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

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

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

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

        //
        this.unary.apply(xBest, xCur, random);
        double fCur = process.evaluate(xCur);
        if (fCur < fBest) {
            fBest = fCur;
            process.getSearchSpace().copy(xCur, xBest);
        }
        //
    }
}
Implementation of the Stochastic Hill Climber

```java
package aitoa.algorithms;

public class HillClimber<X, Y> extends Metaheuristic1<X, Y> {
    // unnecessary stuff omitted here...

    public void solve(IBlackBoxProcess<X, Y> process) {
        X xCur = process.getSearchSpace().create();
        X xBest = process.getSearchSpace().create();
        Random random = process.getRandom();

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

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

• Local searches like hill climbers exploit a property of many optimization problems called causality\textsuperscript{6–9}.
Causality

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

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

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

• Local searches like hill climbers exploit a property of many optimization problems called causality\textsuperscript{6–9}.

• Causality means that small changes in the features of an object (or candidate solution) also lead to small changes in its behavior (or objective value).

• If an optimization problem exhibits causality, then there should be good solutions that are similar to other good solutions.

• The idea is that if we have a good candidate solution, then there may exist similar solutions which are better.

• We hope to find one of them and then continue trying to do the same from there.
Ingredient: Unary Search Operator
Unary Search Operator

- Our hill climber must be able to make modified copies of an existing point \( x \in X \) in order to find these better points.
Unary Search Operator

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

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

• Our hill climber must be able to make modified copies of an existing point \( x \in X \) in order to find these better points.

• A unary search operator accepts an existing point \( x \in X \) and creates a modified copy of it.

• It must make sure that the modified copy is still a valid element of \( X \).

• It should ideally be randomized, i.e., applying it twice to the same point \( x \) should yield different results.
Unary Search Operator

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

```java
package aitoa.structure;

public interface IUnarySearchOperator<X> {
    void apply(X x, X dest, Random random);
}
```
Unary Search Operator

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

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

- Our hill climber must be able to make modified copies of an existing point \( x \in X \) in order to find these better points.
- A unary search operator accepts on existing point \( x \in X \) and creates a modified copy of it.
- It must make sure that the modified copy is still a valid element of \( X \).
- It should ideally be randomized, i.e., applying it twice to the same point \( x \) should yield different results.
- How can we implement this for our JSSP scenario?
  - Easy: Just swap two (different) job IDs in the string!
  - Since the numbers of occurrences of the IDs will not change, the new strings will be valid.
Example for our 1swap Operator

(2,0,1,0,1,1,2,3,2,3,
2,0,0,1,3,3,2,3,1,0)
Example for our 1swap Operator

\[ X \quad \begin{pmatrix} 2,0,1,0,1,1,2,3,2,3, \\ 2,0,0,1,3,3,2,3,1,0 \end{pmatrix} \quad Y \]
Example for our $1_{\text{swap}}$ Operator

$\gamma$

makespan: 180

$(2,0,1,0,1,1,2,3,2,3,$
$2,0,0,1,3,3,2,3,1,0)$
Example for our \textit{1swap} Operator

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

\[\gamma\]

\text{makespan: 180}
Example for our 1\text{swap} Operator

X  
(2,0,1,0,1,1,2,3,2,3,  
2,0,0,1,3,3,2,3,1,0)  \xrightarrow{1\text{swap}} (2,0,1,0,1,1,2,3,2,3,  
2,0,3,1,3,0,2,3,1,0)

\gamma \downarrow \text{makespan: 180}
Example for our 1swap Operator

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

1swap

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

 Makespan: 180

 Makespan: 195
Example for our 1swap Operator

\[ (2,0,1,0,1,1,2,3,2,3,2,0,0,1,3,3,2,3,1,0) \xrightarrow{1\text{swap}} (2,0,1,0,1,1,2,3,2,3,2,0,3,1,3,0,2,3,1,0) \]

\( \gamma \) makespan: 180

\( \gamma \) makespan: 195
Example for our 1swap Operator

X

(2,0,1,0,1,1,2,3,2,3,
2,0,0,1,3,3,2,3,1,0)

1swap

(2,0,1,0,1,1,2,3,2,3,
2,0,3,1,3,0,2,3,1,0)

\gamma

makespan: 180

Y

(2,0,1,0,1,1,2,3,2,3,
2,0,0,1,3,3,2,3,1,0)

\gamma

makespan: 195
package aitoa.examples.jssp;

public class JSSPUUnaryOperator1Swap {
    //
    // unnecessary stuff omitted here...
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
}

package aitoa.examples.jssp;

public class JSSPUnaryOperator1Swap implements IUnarySearchOperator<int[]> {
    // unnecessary stuff omitted here...
    //
    //
    //
package aitoa.examples.jssp;

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

    public void apply(int[] x, int[] dest, Random random) {
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    }
}
package aitoa.examples.jssp;

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

package aitoa.examples.jssp;

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

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

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

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

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

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

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

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

        //
        dest[i] = jobJ;
        dest[j] = jobI; // then we swap the values
        //
        //
        }
}
package aitoaexamples.jssp;

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

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

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

package aitoa.examples.jssp;

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

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

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

- I execute the program 101 times for each of the instances abz7, la24, swv15, and yn4
So what do we get?

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

<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>rs</td>
<td>895</td>
<td>947</td>
</tr>
<tr>
<td></td>
<td>hc_1swap</td>
<td>717</td>
<td>800</td>
</tr>
<tr>
<td>la24</td>
<td>rs</td>
<td>1153</td>
<td>1206</td>
</tr>
<tr>
<td></td>
<td>hc_1swap</td>
<td>999</td>
<td>1095</td>
</tr>
<tr>
<td>swv15</td>
<td>rs</td>
<td>4988</td>
<td>5166</td>
</tr>
<tr>
<td></td>
<td>hc_1swap</td>
<td>3837</td>
<td>4108</td>
</tr>
<tr>
<td>yn4</td>
<td>rs</td>
<td>1460</td>
<td>1498</td>
</tr>
<tr>
<td></td>
<td>hc_1swap</td>
<td>1109</td>
<td>1222</td>
</tr>
</tbody>
</table>
So what do we get?

rs: median result of 3 min of random sampling
So what do we get?

hc_1swap: median result of 3 min of hill climber
So what do we get?

rs: median result of 3 min of random sampling
So what do we get?

hc_1swap: median result of 3 min of hill climber
So what do we get?

rs: median result of 3 min of random sampling
So what do we get?

hc_1swap: median result of 3 min of hill climber
So what do we get?

rs: median result of 3 min of random sampling
So what do we get?

hc_1swap: median result of 3 min of hill climber
Progress over Time

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

First we have much progress...
What progress does the algorithm make over time?

First we have much progress...
What progress does the algorithm make over time?

First we have much progress...
Progress over Time

What progress does the algorithm make over time?

First we have much progress...
What progress does the algorithm make over time?

First we have much progress...

...but then the hill climber stagnates!
But we waste time...

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

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

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

Then it looks even much worse!
But we waste time...

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

Then it looks even much worse!
But we waste time...

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

Then it looks even much worse!
But we waste time...

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

Then it looks even much worse!
Indeed, we waste time!

<table>
<thead>
<tr>
<th>$\mathcal{I}$</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>rs</td>
<td>895</td>
<td>947</td>
</tr>
<tr>
<td></td>
<td>hc_1swap</td>
<td>717</td>
<td>800</td>
</tr>
<tr>
<td>la24</td>
<td>rs</td>
<td>1153</td>
<td>1206</td>
</tr>
<tr>
<td></td>
<td>hc_1swap</td>
<td>999</td>
<td>1095</td>
</tr>
<tr>
<td>swv15</td>
<td>rs</td>
<td>4988</td>
<td>5166</td>
</tr>
<tr>
<td></td>
<td>hc_1swap</td>
<td>3837</td>
<td>4108</td>
</tr>
<tr>
<td>yn4</td>
<td>rs</td>
<td>1460</td>
<td>1498</td>
</tr>
<tr>
<td></td>
<td>hc_1swap</td>
<td>1109</td>
<td>1222</td>
</tr>
</tbody>
</table>
Indeed, we waste time!

<table>
<thead>
<tr>
<th>$I$</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>rs</td>
<td>895</td>
<td>947</td>
</tr>
<tr>
<td></td>
<td>hc_1swap</td>
<td>717</td>
<td>800</td>
</tr>
<tr>
<td>la24</td>
<td>rs</td>
<td>1153</td>
<td>1206</td>
</tr>
<tr>
<td></td>
<td>hc_1swap</td>
<td>999</td>
<td>1095</td>
</tr>
<tr>
<td>swv15</td>
<td>rs</td>
<td>4988</td>
<td>5166</td>
</tr>
<tr>
<td></td>
<td>hc_1swap</td>
<td>3837</td>
<td>4108</td>
</tr>
<tr>
<td>yn4</td>
<td>rs</td>
<td>1460</td>
<td>1498</td>
</tr>
<tr>
<td></td>
<td>hc_1swap</td>
<td>1109</td>
<td>1222</td>
</tr>
</tbody>
</table>
Indeed, we waste time!

<table>
<thead>
<tr>
<th>$I$</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>rs</td>
<td>895</td>
<td>947</td>
</tr>
<tr>
<td></td>
<td>hc_1swap</td>
<td>717</td>
<td>800</td>
</tr>
<tr>
<td>la24</td>
<td>rs</td>
<td>1153</td>
<td>1206</td>
</tr>
<tr>
<td></td>
<td>hc_1swap</td>
<td>999</td>
<td>1095</td>
</tr>
<tr>
<td>swv15</td>
<td>rs</td>
<td>4988</td>
<td>5166</td>
</tr>
<tr>
<td></td>
<td>hc_1swap</td>
<td>3837</td>
<td>4108</td>
</tr>
<tr>
<td>yn4</td>
<td>rs</td>
<td>1460</td>
<td>1498</td>
</tr>
<tr>
<td></td>
<td>hc_1swap</td>
<td>1109</td>
<td>1222</td>
</tr>
</tbody>
</table>

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

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

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

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

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

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

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

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

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

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

- Our algorithm makes most of its progress early during the search.
- Later, it basically stagnates and cannot improve.
- Why is that?
- The search operator `1swap` defines a neighborhood `N(x) ⊂ X` around a point `x`.
- The hill climber can only find solutions which are in the neighborhood of the current best solution.
- Only the schedules that I can reach by swapping two operations of two different jobs.
- Clearly `|N(x)| ≪ |X|`!
- What happens if `f(γ(x)) ≤ f(γ(x)) ∀ x ∈ N(x)` but `x` is not the global optimum?
- Our algorithm gets trapped in the local optimum `x` and cannot escape!
- This is called **Premature Convergence**.
Premature Convergence

objective values $f(y(x))$

local optimum

global optimum

$X$
Improved Algorithm Concept 1
Stochastic Hill Climber with Restarts

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

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

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

• At the same time, a single run of the algorithm converges quickly.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Algorithm</th>
<th>Best</th>
<th>Mean</th>
<th>Median</th>
<th>Standard Deviation</th>
<th>Last Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>abz7</td>
<td>rs</td>
<td>895</td>
<td>947</td>
<td>949</td>
<td>12</td>
<td>85s</td>
</tr>
<tr>
<td></td>
<td>hc_1swap</td>
<td>717</td>
<td>800</td>
<td>798</td>
<td>28</td>
<td>0s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6'512'505</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16'978</td>
</tr>
<tr>
<td>la24</td>
<td>rs</td>
<td>1153</td>
<td>1206</td>
<td>1208</td>
<td>15</td>
<td>82s</td>
</tr>
<tr>
<td></td>
<td>hc_1swap</td>
<td>999</td>
<td>1095</td>
<td>1086</td>
<td>56</td>
<td>0s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15'902'911</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6'612</td>
</tr>
<tr>
<td>swv15</td>
<td>rs</td>
<td>4988</td>
<td>5166</td>
<td>5172</td>
<td>50</td>
<td>87s</td>
</tr>
<tr>
<td></td>
<td>hc_1swap</td>
<td>3837</td>
<td>4108</td>
<td>4108</td>
<td>137</td>
<td>1s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5'559'124</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>104'598</td>
</tr>
<tr>
<td>yn4</td>
<td>rs</td>
<td>1460</td>
<td>1498</td>
<td>1499</td>
<td>15</td>
<td>76s</td>
</tr>
<tr>
<td></td>
<td>hc_1swap</td>
<td>1109</td>
<td>1222</td>
<td>1220</td>
<td>48</td>
<td>0s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4'814'914</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>31'789</td>
</tr>
</tbody>
</table>
Stochastic Hill Climber with Restarts

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

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

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

• At the same time, a single *run* of the algorithm converges quickly.

• Let us exploit this variance!

• Idea: If we did not make any progress for a number $L$ of algorithm steps, we simply restart at a new random solution.

• Of course, we will always remember the overall best solution we ever had (in another variable).
Stochastic Hill Climbing Algorithm with Restarts

```java
package aitoa.algorithms;

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

```java
```
Stochastic Hill Climbing Algorithm with Restarts

```java
package aitoa.algorithms;

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

    }
}
```
package aitoa.algorithms;

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

```java
package aitoa.algorithms;

public class HillClimberWithRestarts<X, Y> extends Metaheuristic1<X, Y> {
    // unnecessary stuff omitted here...
    public void solve(IBlackBoxProcess<X, Y> process) {
        X xCur = process.getSearchSpace().create();
        X xBest = process.getSearchSpace().create();
    }
}
```
Stochastic Hill Climbing Algorithm with Restarts

```java
package aitoa.algorithms;

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

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

      // this.nullary.apply(xBest, random); // sample random solution
   }
}
package aitoa.algorithms;

public class HillClimberWithRestarts<X, Y> extends Metaheuristic1<X, Y> {

    // unnecessary stuff omitted here...

    public void solve(IBlackBoxProcess<X, Y> process) {
        X xCur = process.getSearchSpace().create();
        X xBest = process.getSearchSpace().create();
        Random random = process.getRandom();

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

        // process has stored best-so-far result
    }
}
package aitoa.algorithms;

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

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

        //
        this.unary.apply(xBest, xCur, random); // try to improve

        // process has stored best-so-far result
    }
}
Stochastic Hill Climbing Algorithm with Restarts

package aitoa.algorithms;

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

        //
        //
        this.nullary.apply(xBest, random); // sample random solution
        double fBest = process.evaluate(xBest); // evaluate it
    //
    //
    //
    //
    this.unary.apply(xBest, xCur, random); // try to improve
    double fCur = process.evaluate(xCur); // evaluate
    //
    //
    //
    //
    //
    //
    //
    //
    //
    } // process has stored best-so-far result
}
package aitoa.algorithms;

public class HillClimberWithRestarts<X, Y> extends Metaheuristic1<X, Y> {

    // unnecessary stuff omitted here...

    public void solve(IBlackBoxProcess<X, Y> process) {
        X xCur = process.getSearchSpace().create();
        X xBest = process.getSearchSpace().create();
        Random random = process.getRandom();

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

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

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

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

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

```java
package aitoa.algorithms;

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

    //
    this.nullary.apply(xBest, random); // sample random solution
    double fBest = process.evaluate(xBest); // evaluate it
    //
    //
    this.unary.apply(xBest, xCur, random); // try to improve
    double fCur = process.evaluate(xCur); // evaluate
    if (fCur < fBest) {
      fBest = fCur; // we found a better solution
      process.getSearchSpace().copy(xCur, xBest); // copy
      //
    }
    //
    //
    //
    //
    // process has stored best-so-far result
  }
}
```
Stochastic Hill Climbing Algorithm with Restarts

```java
package aitoa.algorithms;

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

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

        while (!process.shouldTerminate()) { // inner loop
            this.unary.apply(xBest, xCur, random); // try to improve
            double fCur = process.evaluate(xCur); // evaluate
            if (fCur < fBest) { // we found a better solution
                fBest = fCur; // remember best quality
                process.getSearchSpace().copy(xCur, xBest); // copy
            }
        }
    }
}
```
Stochastic Hill Climbing Algorithm with Restarts

```java
package aitoa.algorithms;

public class HillClimberWithRestarts<X, Y> extends Metaheuristic1<X, Y> {
   // unnecessary stuff omitted here...

   public void solve(IBlackBoxProcess<X, Y> process) {
      X xCur = process.getSearchSpace().create();
      X xBest = process.getSearchSpace().create();
      Random random = process.getRandom();

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

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

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

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

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

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

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

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

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

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

```java
package aitoa.algorithms;

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

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

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

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

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

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

```java
package aitoa.algorithms;

public class HillClimberWithRestarts<X, Y> extends Metaheuristic1<X, Y> {

    // unnecessary stuff omitted here...

    public void solve(IBlackBoxProcess<X, Y> process) {
        X xCur = process.getSearchSpace().create();
        X xBest = process.getSearchSpace().create();
        Random random = process.getRandom();

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

            while (!process.shouldTerminate()) {
                // inner loop
                this.unary.apply(xBest, xCur, random); // try to improve
                double fCur = process.evaluate(xCur); // evaluate
                if (fCur < fBest) {
                    // we found a better solution
                    fBest = fCur; // remember best quality
                    process.getSearchSpace().copy(xCur, xBest); // copy
                    failCounter = 0L; // reset number of unsuccessful steps
                } else {
                    // ok, we did not find an improvement
                    if ( (++failCounter) >= this.failsBeforeRestart ) {
                        break; // jump back to outer loop for restart
                    } // increase fail counter
                } // failure
            } // inner loop
        } // outer loop
    } // process has stored best-so-far result
```
Experiment and Analysis
We now have an algorithm which, in theory, should be able to utilize some of the variance that we observe in the results of hc_1swap.
Configuring the Algorithm: Parameter $L$

- We now have an algorithm which, in theory, should be able to utilize some of the variance that we observe in the results of $hc_{1\text{swap}}$.
- We got one problem, though …
Configuring the Algorithm: Parameter $L$

- We now have an algorithm which, in theory, should be able to utilize some of the variance that we observe in the results of $\text{hc}_1\text{swap}$.
- We got one problem, though . . . . . actually, it is not just one algorithm, it is an algorithm with a parameter $L$: $\text{hcr}_L\text{_1swap}$. 
Configuring the Algorithm: Parameter $L$

- We now have an algorithm which, in theory, should be able to utilize some of the variance that we observe in the results of $hc_{\text{1swap}}$.
- We got one problem, though . . . . . . actually, it is not just one algorithm, it is an algorithm with a parameter $L$: $hcr_{\text{L_{1swap}}}$.
- What do we do with that?
Configuring the Algorithm: Parameter $L$

- We now have an algorithm which, in theory, should be able to utilize some of the variance that we observe in the results of $hc_{1\text{swap}}$.
- We got one problem, though . . . . . actually, it is not just one algorithm, it is an algorithm with a parameter $L$: $hcr_{L\_1\text{swap}}$.
- What do we do with that?
- Let’s take a look.
Configuring the Algorithm: Parameter $L$

![Graph showing the relationship between $L$ and different datasets](image-url)
Configuring the Algorithm: Parameter $L$

- We now have an algorithm which, in theory, should be able to utilize some of the variance that we observe in the results of $hc_1\text{swap}$.
- We got one problem, though ... actually, it is not just one algorithm, it is an algorithm with a parameter $L$: $hcr_L_1\text{swap}$.
- What do we do with that?
- Let's take a look.
- If we choose $L$ too small, we will restart the algorithm too early
We now have an algorithm which, in theory, should be able to utilize some of the variance that we observe in the results of \( \text{hc}_1\text{swap} \).

We got one problem, though . . . . actually, it is not just one algorithm, it is an algorithm with a parameter \( L \): \( \text{hcr}_L\text{_1swap} \).

What do we do with that?

Let’s take a look.

If we choose \( L \) too small, we will restart the algorithm too early, before it even arrives in a local optimum.
We now have an algorithm which, in theory, should be able to utilize some of the variance that we observe in the results of \texttt{hc_1swap}.

We got one problem, though . . . . . . actually, it is not just one algorithm, it is an algorithm with a parameter $L$: \texttt{hcr\_L\_1swap}.

What do we do with that?

Let’s take a look.

If we choose $L$ too small, we will restart the algorithm too early, before it even arrives in a local optimum.

If we choose $L$ too large, we will restart too late and thus waste time.
Configuring the Algorithm: Parameter $L$

• We now have an algorithm which, in theory, should be able to utilize some of the variance that we observe in the results of $hc_{1swap}$.

• We got one problem, though . . . . . . actually, it is not just one algorithm, it is an algorithm with a parameter $L$: $hcr_{L_{1swap}}$.

• What do we do with that?

• Let’s take a look.

• If we choose $L$ too small, we will restart the algorithm too early, before it even arrives in a local optimum

• If we choose $L$ too large, we will restart too late and thus waste time, that we could have used for more restarts
Configuring the Algorithm: Parameter $L$

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

- I execute the program 101 times for each of the instances abz7, la24, swv15, and yn4
So what do we get?

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

<table>
<thead>
<tr>
<th>Instance</th>
<th>Algorithm</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>rs</td>
<td>895</td>
<td>947</td>
</tr>
<tr>
<td></td>
<td>hc_1swap</td>
<td>717</td>
<td>800</td>
</tr>
<tr>
<td></td>
<td>hcr_16384_1swap</td>
<td>714</td>
<td>732</td>
</tr>
<tr>
<td>la24</td>
<td>rs</td>
<td>1153</td>
<td>1206</td>
</tr>
<tr>
<td></td>
<td>hc_1swap</td>
<td>999</td>
<td>1095</td>
</tr>
<tr>
<td></td>
<td>hcr_16384_1swap</td>
<td>953</td>
<td>976</td>
</tr>
<tr>
<td>swv15</td>
<td>rs</td>
<td>4988</td>
<td>5166</td>
</tr>
<tr>
<td></td>
<td>hc_1swap</td>
<td>3837</td>
<td>4108</td>
</tr>
<tr>
<td></td>
<td>hcr_16384_1swap</td>
<td>3752</td>
<td>3859</td>
</tr>
<tr>
<td>yn4</td>
<td>rs</td>
<td>1460</td>
<td>1498</td>
</tr>
<tr>
<td></td>
<td>hc_1swap</td>
<td>1109</td>
<td>1222</td>
</tr>
<tr>
<td></td>
<td>hcr_16384_1swap</td>
<td>1081</td>
<td>1115</td>
</tr>
</tbody>
</table>
So what do we get?

**hc_1swap**: median result of 3 min of hill climber
So what do we get?

hcr_16384_1swap: median result of 3 min of hill climber which restarts after $L = 16'384$ search steps without improvement
So what do we get?

hc_1swap: median result of 3 min of hill climber
So what do we get?

hcr_16384_1swap: median result of 3 min of hill climber which restarts after $L = 16'384$ search steps without improvement
So what do we get?

hc_1swap: median result of 3 min of hill climber

swv15 / 4108
So what do we get?

**hcr_16384_1swap**: median result of 3 min of hill climber which restarts after $L = 16'384$ search steps without improvement.
So what do we get?

hc_1swap: median result of 3 min of hill climber
So what do we get?

hcr_16384_1swap: median result of 3 min of hill climber which restarts after
\[ L = 16'384 \] search steps without improvement
What progress does the algorithm make over time?
Progress over Time

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

What progress does the algorithm make over time?

- First it behaves like the normal hill climber
Progress over Time

What progress does the algorithm make over time?

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

What progress does the algorithm make over time?

- First it behaves like the normal hill climber
- But it keeps improving.
What progress does the algorithm make over time?

- First it behaves like the normal hill climber
- But it keeps improving.
What progress does the algorithm make over time?

- First it behaves like the normal hill climber
- But it keeps improving.
What progress does the algorithm make over time?

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

What progress does the algorithm make over time?

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

What progress does the algorithm make over time?

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

What progress does the algorithm make over time?

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

What progress does the algorithm make over time?

• First it behaves like the normal hill climber
• But it keeps improving.
• Although we still do not use the available time very well...
Improved Algorithm Concept 2
Drawbacks of Restarts

• A restarted algorithm is still the same algorithm, just restarted.
Drawbacks of Restarts

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

• A restarted algorithm is still the same algorithm, just restarted.
• If there are many more local optima than global optima, every restart will probably end again in a local optimum.
• If there are many more “bad” local optima than “good” local optima, every restart will probably end in a “bad” local optimum.
Drawbacks of Restarts

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

• A restarted algorithm is still the same algorithm, just restarted.
• If there are many more local optima than global optima, every restart will probably end again in a local optimum.
• If there are many more “bad” local optima than “good” local optima, every restart will probably end in a “bad” local optimum.
• While restarts improve the chance to find better solutions, they cannot solve the intrinsic shortcomings of an algorithm.
• Another problem is: With every restart we throw away all accumulated knowledge and information of the current run.
Drawbacks of Restarts

- A restarted algorithm is still the same algorithm, just restarted.
- If there are many more local optima than global optima, every restart will probably end again in a local optimum.
- If there are many more “bad” local optima than “good” local optima, every restart will probably end in a “bad” local optimum.
- While restarts improve the chance to find better solutions, they cannot solve the intrinsic shortcomings of an algorithm.
- Another problem is: With every restart we throw away all accumulated knowledge and information of the current run.
- Restarts are therefore also wasteful.
How else can we stop premature convergence?

- Our \texttt{hc\_1swap} hill climber will stop improving if it can no longer find better solutions.
How else can we stop premature convergence?

- Our \texttt{hc\_1swap} hill climber will stop improving if it can no longer find better solutions.
- This happens when it reaches a local optimum.
How else can we stop premature convergence?

- Our \texttt{hc\_1swap} hill climber will stop improving if it can no longer finder better solutions.
- This happens when it reaches a local optimum.
- A local optimum is a point $x^\times$ in $\mathbb{X}$ where no \texttt{1swap}-move can yield any improvement.
How else can we stop premature convergence?

• Our \texttt{hc\_1swap} hill climber will stop improving if it can no longer find better solutions.
• This happens when it reaches a local optimum.
• A local optimum is a point $x^\times$ in $\mathbb{X}$ where no 1\texttt{swap}-move can yield any improvement.
• It does not matter which two job ids I exchange in the current best string $x^\times$, the result is not better than $x^\times$. 
How else can we stop premature convergence?

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

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

- Two solutions \( x_1 \) and \( x_2 \) are “neighbors” if I can reach \( x_2 \) by applying the search operator one time to \( x_1 \).
Making the neighborhood bigger

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

Making the neighborhood bigger

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

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

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

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

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

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

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

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

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

- Idea: Let’s most often swap 2 jobs
Making the neighborhood bigger

- Idea: Let’s most often swap 2 jobs, but sometimes 3
Making the neighborhood bigger

• Idea: Let’s most often swap 2 jobs, but sometimes 3, less often 4
Making the neighborhood bigger

• Idea: Let’s most often swap 2 jobs, but sometimes 3, less often 4, from time to time 5
Making the neighborhood bigger

- Idea: Let’s most often swap 2 jobs, but sometimes 3, less often 4, from time to time 5, rarely 6
Making the neighborhood bigger

• Idea: Let’s most often swap 2 jobs, but sometimes 3, less often 4, from time to time 5, rarely 6, hardly ever 7
Making the neighborhood bigger

• Idea: Let’s most often swap 2 jobs, but sometimes 3, less often 4, from time to time 5, rarely 6, hardly ever 7, …
Making the neighborhood bigger

• Idea: Let’s most often swap 2 jobs, but sometimes 3, less often 4, from time to time 5, rarely 6, hardly ever 7, . . .
• nswap operator idea
Making the neighborhood bigger

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

• nswap operator idea:
  1. flip a coin: if it is heads (50% probability), we will swap 2 job ids (and stop).
Making the neighborhood bigger

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

• nswap operator idea:
  1. flip a coin: if it is heads (50% probability), we will swap 2 job ids (and stop).
  2. otherwise (it was tail), we again flip a coin.
Making the neighborhood bigger

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

• nswap operator idea:
  1. flip a coin: if it is heads (50% probability), we will swap 2 job ids (and stop).
  2. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 25% in total), we will swap 3 job ids (and stop).
Making the neighborhood bigger

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

• nswap operator idea:
  1. flip a coin: if it is heads (50% probability), we will swap 2 job ids (and stop).
  2. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 25% in total), we will swap 3 job ids (and stop).
  3. otherwise (it was tail), we again flip a coin.
Making the neighborhood bigger

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

• nswap operator idea:
  1. flip a coin: if it is heads (50% probability), we will swap 2 job ids (and stop).
  2. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 25% in total), we will swap 3 job ids (and stop).
  3. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 12.5% in total), we will swap 4 job ids (and stop).
Making the neighborhood bigger

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

• nswap operator idea:
  1. flip a coin: if it is heads (50% probability), we will swap 2 job ids (and stop).
  2. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 25% in total), we will swap 3 job ids (and stop).
  3. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 12.5% in total), we will swap 4 job ids (and stop).
  4. otherwise (it was tail), we again flip a coin.
Making the neighborhood bigger

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

- nswap operator idea:
  1. flip a coin: if it is heads (50% probability), we will swap 2 job ids (and stop).
  2. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 25% in total), we will swap 3 job ids (and stop).
  3. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 12.5% in total), we will swap 4 job ids (and stop).
  4. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 6.25% in total), we will swap 5 job ids (and stop).
Making the neighborhood bigger

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

• nswap operator idea:
  1. flip a coin: if it is heads (50% probability), we will swap 2 job ids (and stop).
  2. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 25% in total), we will swap 3 job ids (and stop).
  3. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 12.5% in total), we will swap 4 job ids (and stop).
  4. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 6.25% in total), we will swap 5 job ids (and stop).
  5. otherwise (it was tail), we again flip a coin.
Making the neighborhood bigger

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

• nswap operator idea:
  1. flip a coin: if it is heads (50% probability), we will swap 2 job ids (and stop).
  2. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 25% in total), we will swap 3 job ids (and stop).
  3. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 12.5% in total), we will swap 4 job ids (and stop).
  4. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 6.25% in total), we will swap 5 job ids (and stop).
  5. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 3.125% in total), we will swap 6 job ids (and stop).
Making the neighborhood bigger

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

- nswap operator idea:
  1. flip a coin: if it is heads (50% probability), we will swap 2 job ids (and stop).
  2. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 25% in total), we will swap 3 job ids (and stop).
  3. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 12.5% in total), we will swap 4 job ids (and stop).
  4. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 6.25% in total), we will swap 5 job ids (and stop).
  5. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 3.125% in total), we will swap 6 job ids (and stop).
  6. and so on.
Making the neighborhood bigger

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

- **nswap operator idea:**
  1. Flip a coin: if it is heads (50% probability), we will swap 2 job ids (and stop).
  2. Otherwise (it was tail), we again flip a coin. If it is heads (50% probability, now 25% in total), we will swap 3 job ids (and stop).
  3. Otherwise (it was tail), we again flip a coin. If it is heads (50% probability, now 12.5% in total), we will swap 4 job ids (and stop).
  4. Otherwise (it was tail), we again flip a coin. If it is heads (50% probability, now 6.25% in total), we will swap 5 job ids (and stop).
  5. Otherwise (it was tail), we again flip a coin. If it is heads (50% probability, now 3.125% in total), we will swap 6 job ids (and stop).
  6. And so on.

- **We most often make small moves, but sometimes bigger ones.**
Making the neighborhood bigger

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

• nswap operator idea:
  1. flip a coin: if it is heads (50% probability), we will swap 2 job ids (and stop).
  2. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 25% in total), we will swap 3 job ids (and stop).
  3. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 12.5% in total), we will swap 4 job ids (and stop).
  4. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 6.25% in total), we will swap 5 job ids (and stop).
  5. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 3.125% in total), we will swap 6 job ids (and stop).
  6. and so on.

• We most often make small moves, but sometimes bigger ones.

• Theoretically, we could always escape from any local optimum.
Making the neighborhood bigger

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

• nswap operator idea:
  1. flip a coin: if it is heads (50% probability), we will swap 2 job ids (and stop).
  2. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 25% in total), we will swap 3 job ids (and stop).
  3. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 12.5% in total), we will swap 4 job ids (and stop).
  4. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 6.25% in total), we will swap 5 job ids (and stop).
  5. otherwise (it was tail), we again flip a coin. if it is heads (50% probability, now 3.125% in total), we will swap 6 job ids (and stop).
  6. and so on.

• We most often make small moves, but sometimes bigger ones.

• Theoretically, we could always escape from any local optimum, but the probability may sometimes be very very small.
Implementation of the \texttt{nswap} Operator

```java
package aitoa.examples.jssp;

public class JSSPUnaryOperatorNSwap {
  // unnecessary stuff omitted here...
  //
  //
  //
  //
  //
  //
  //
  //
  //
  //
  //
  //
  //
  //
  //
  //
  //
  //
  //
  //
  //
  //
  //
  //
  //
  //
  //
  //
}
```
package aitoa.examples.jssp;

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

}
Implementation of the `nswap` Operator

```java
package aitoa.examples.jssp;

public class JSSPUnaryOperatorNSwap implements IUnarySearchOperator<int[]> {
    // unnecessary stuff omitted here...
    public void apply(int[] x, int[] dest, Random random) {
        //
    }
}
```
package aitoa.examples.jssp;

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

```java
package aitoa.examples.jssp;

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

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

Implementation of the *nswap* Operator

```java
package aitoa.examples.jssp;

public class JSSPUnaryOperatorNSwap implements IUnarySearchOperator<int[]> {
    // unnecessary stuff omitted here...
    public void apply(int[] x, int[] dest, Random random) {
        System.arraycopy(x, 0, dest, 0, x.length); // copy x to dest
        int i = random.nextInt(dest.length); // index of first job to swap
        int first = dest[i];
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        dest[i] = first; // write back first id to last copied index
    }
}
```
package aitoa.examples.jssp;

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

      int j = random.nextInt(dest.length);

      dest[i] = first; // write back first id to last copied index
   }
}
Implementation of the nswap Operator

```java
default package aitoa.examples.jssp;

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

    public void apply(int[] x, int[] dest, Random random) {
        System.arraycopy(x, 0, dest, 0, x.length); // copy x to dest
        int i = random.nextInt(dest.length); // index of first job to swap
        int first = dest[i];

        int j = random.nextInt(dest.length);
        int jobJ = dest[j];

        dest[i] = first; // write back first id to last copied index
    }
}
```
Implementation of the nswap Operator

```java
package aitoa.examples.jssp;

public class JSSPUnaryOperatorNSwap implements IUnarySearchOperator<int[]> {
    // unnecessary stuff omitted here...
    public void apply(int[] x, int[] dest, Random random) {
        System.arraycopy(x, 0, dest, 0, x.length); // copy x to dest
        int i = random.nextInt(dest.length); // index of first job to swap
        int first = dest[i];
        //
        int j = random.nextInt(dest.length);
        int jobJ = dest[j];
        if (first != jobJ) {
            //
        }
        //
        dest[i] = first; // write back first id to last copied index
    }
}
```
Implementation of the \texttt{nswap} Operator

```java
package aitoa.examples.jssp;

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

```java
package aitoa.examples.jssp;

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

```java
package aitoa.examples.jssp;

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

        //
        //
        //
        // inner: for (; ;) { // find a location with a different job
        int j = random.nextInt(dest.length);
        int jobJ = dest[j];
        if (first != jobJ) {
            //        dest[i] = jobJ; // overwrite job at index i with jobJ
            i = j; // remember index j: we will overwrite it next
            //        break inner;
        }
        //
        //
        dest[i] = first; // write back first id to last copied index
    }
}
```
Implementation of the nswap Operator

```java
package aitoa.examples.jssp;

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

    public void apply(int[] x, int[] dest, Random random) {
        System.arraycopy(x, 0, dest, 0, x.length); // copy x to dest
        int i = random.nextInt(dest.length); // index of first job to swap
        int first = dest[i];

        for(;;) {
            for(;;) { // find a location with a different job
                int j = random.nextInt(dest.length);
                int jobJ = dest[j];
                if (first != jobJ) {
                    dest[i] = jobJ; // overwrite job at index i with jobJ
                    i = j; // remember index j: we will overwrite it next
                    break inner;
                }
            }
        }
        dest[i] = first; // write back first id to last copied index
    }
}
```
Implementation of the nswap Operator

```java
package aitoa.examples.jssp;

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

```java
package aitoa.examples.jssp;

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

    public void apply(int[] x, int[] dest, Random random) {
        System.arraycopy(x, 0, dest, 0, x.length); // copy x to dest
        int i = random.nextInt(dest.length); // index of first job to swap
        int first = dest[i];
        int last = first; // last stores the job id to "swap in"
        boolean hasNext;
        do {
            hasNext = random.nextBoolean();
            inner: for (;;) { // find a location with a different job
                int j = random.nextInt(dest.length);
                int jobJ = dest[j];
                if (first != jobJ) {
                    dest[i] = jobJ; // overwrite job at index i with jobJ
                    i = j; // remember index j: we will overwrite it next
                    break inner;
                }
            }
        } while (hasNext); // Bernoulli process
        dest[i] = first; // write back first id to last copied index
    }
}
```
package aitoa.examples.jssp;

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

    public void apply(int[] x, int[] dest, Random random) {
        System.arraycopy(x, 0, dest, 0, x.length); // copy x to dest
        int i = random.nextInt(dest.length); // index of first job to swap
        int first = dest[i];
        int last = first; // last stores the job id to "swap in"
        boolean hasNext;
        do {
            hasNext = random.nextBoolean();
            inner: for (; ;) {
                int j = random.nextInt(dest.length);
                int jobJ = dest[j];
                if (first != jobJ) {
                    dest[i] = jobJ; // overwrite job at index i with jobJ
                    i = j; // remember index j: we will overwrite it next
                    last = jobJ; // but not with the same value jobJ...
                    break inner;
                }
            }
        } while (hasNext); // Bernoulli process
        dest[i] = first; // write back first id to last copied index
    }
}
package aitoa.examples.jssp;

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

```java
package aitoa.examples.jssp;

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

    public void apply(int[] x, int[] dest, Random random) {
        System.arraycopy(x, 0, dest, 0, x.length); // copy x to dest
        int i = random.nextInt(dest.length); // index of first job to swap
        int first = dest[i];
        int last = first; // last stores the job id to "swap in"
        boolean hasNext;
        do { // we repeat a geometrically distributed number of times
            hasNext = random.nextBoolean();
            inner: for (;;) { // find a location with a different job
                int j = random.nextInt(dest.length);
                int jobJ = dest[j];
                if ((last != jobJ) && // don't swap job with itself
                    (hasNext || (first != jobJ))) { // also not at end
                    dest[i] = jobJ; // overwrite job at index i with jobJ
                    i = j; // remember index j: we will overwrite it next
                    last = jobJ; // but not with the same value jobJ...
                    break inner;
                }
            }
        } while (hasNext); // Bernoulli process
        dest[i] = first; // write back first id to last copied index
    }
}
```
Experiment and Analysis
So what do we get?

• I execute the program 101 times for each of the instances abz7, 1a24, 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>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>hc_1swap</td>
<td>717</td>
<td>800</td>
</tr>
<tr>
<td></td>
<td>hcr_16384_1swap</td>
<td>714</td>
<td>732</td>
</tr>
<tr>
<td></td>
<td>hc_nswap</td>
<td>724</td>
<td>758</td>
</tr>
<tr>
<td>la24</td>
<td>hc_1swap</td>
<td>999</td>
<td>1095</td>
</tr>
<tr>
<td></td>
<td>hcr_16384_1swap</td>
<td>953</td>
<td>976</td>
</tr>
<tr>
<td></td>
<td>hc_nswap</td>
<td>945</td>
<td>1018</td>
</tr>
<tr>
<td>swv15</td>
<td>hc_1swap</td>
<td>3837</td>
<td>4108</td>
</tr>
<tr>
<td></td>
<td>hcr_16384_1swap</td>
<td>3752</td>
<td>3859</td>
</tr>
<tr>
<td></td>
<td>hc_nswap</td>
<td>3602</td>
<td>3880</td>
</tr>
<tr>
<td>yn4</td>
<td>hc_1swap</td>
<td>1109</td>
<td>1222</td>
</tr>
<tr>
<td></td>
<td>hcr_16384_1swap</td>
<td>1081</td>
<td>1115</td>
</tr>
<tr>
<td></td>
<td>hc_nswap</td>
<td>1095</td>
<td>1162</td>
</tr>
</tbody>
</table>
So what do we get?

hc_1swap: median result of 3 min of hill climber using 1swap
So what do we get?

hc_nswap: median result of 3 min of hill climber using nswap
So what do we get?

hc_1swap: median result of 3 min of hill climber using 1swap
So what do we get?

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

---

![Diagram showing median result of 3 min of hill climber using nswap.](chart.png)
So what do we get?

**hc_1swap**: median result of 3 min of hill climber using 1swap
So what do we get?

hc_nswap: median result of 3 min of hill climber using nswap
So what do we get?

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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>19</td>
<td>15</td>
<td>18</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>5</td>
<td>11</td>
<td>19</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>4</td>
<td>10</td>
<td>3</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>19</td>
<td>16</td>
<td>15</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>8</td>
<td>2</td>
<td>15</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>6</td>
<td>11</td>
<td>19</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>5</td>
<td>10</td>
<td>17</td>
<td>19</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>9</td>
<td>18</td>
<td>11</td>
<td>16</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>12</td>
<td>6</td>
<td>13</td>
<td>15</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>7</td>
<td>13</td>
<td>0</td>
<td>19</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>2</td>
<td>0</td>
<td>8</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

yn4 / 1220
So what do we get?

hc_nswap: median result of 3 min of hill climber using nswap
What progress does the algorithm make over time?
What progress does the algorithm make over time?
What progress does the algorithm make over time?

- `hc_nswap` first behaves like `hc_1swap`, because most of the `nswap` moves are the same as `1swap` moves.
What progress does the algorithm make over time?

- The rare larger moves allow it to escape from local optima that would trap `hc_1swap`.
What progress does the algorithm make over time?

- The rare larger moves allow it to escape from local optima that would trap hsc_1swap.
- The hill climber with restarts seems to improve longer.
What progress does the algorithm make over time?

- The rare larger moves allow it to escape from local optima that would trap \texttt{hc\_1swap}.
- The hill climber with restarts seems to improve longer.
What progress does the algorithm make over time?

- The rare larger moves allow it to escape from local optima that would trap `hc_1swap`.
- The hill climber with restarts seems to improve longer.
What progress does the algorithm make over time?

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

- The hill climber with restarts seems to improve longer.
Improved Algorithm Concept 3
Combining the Two Improvements

• Now we know two ways to improve the performance of our hill climber.
Combining the Two Improvements

• Now we know two ways to improve the performance of our hill climber:
  1. we can restart it
Combining the Two Improvements

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

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

• It is only natural to try to combine these two improvements.
## Configuring the Algorithm

<table>
<thead>
<tr>
<th>$I$</th>
<th>algo</th>
<th>best</th>
<th>mean</th>
<th>med</th>
<th>sd</th>
<th>med(t)</th>
<th>med(FEs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>abz7</td>
<td>hc_1swap</td>
<td>717</td>
<td>800</td>
<td>798</td>
<td>28</td>
<td>0s</td>
<td>16'978</td>
</tr>
<tr>
<td></td>
<td>hc_nswap</td>
<td>724</td>
<td>758</td>
<td>758</td>
<td>17</td>
<td>35s</td>
<td>7'781'762</td>
</tr>
<tr>
<td>la24</td>
<td>hc_1swap</td>
<td>999</td>
<td>1095</td>
<td>1086</td>
<td>56</td>
<td>0s</td>
<td>6'612</td>
</tr>
<tr>
<td></td>
<td>hc_nswap</td>
<td>945</td>
<td>1018</td>
<td>1016</td>
<td>29</td>
<td>25s</td>
<td>9'072'935</td>
</tr>
<tr>
<td>swv15</td>
<td>hc_1swap</td>
<td>3837</td>
<td>4108</td>
<td>4108</td>
<td>137</td>
<td>1s</td>
<td>104'598</td>
</tr>
<tr>
<td></td>
<td>hc_nswap</td>
<td>3602</td>
<td>3880</td>
<td>3872</td>
<td>112</td>
<td>70s</td>
<td>8'351'112</td>
</tr>
<tr>
<td>yn4</td>
<td>hc_1swap</td>
<td>1109</td>
<td>1222</td>
<td>1220</td>
<td>48</td>
<td>0s</td>
<td>31'789</td>
</tr>
<tr>
<td></td>
<td>hc_nswap</td>
<td>1095</td>
<td>1162</td>
<td>1160</td>
<td>34</td>
<td>71s</td>
<td>11'016'757</td>
</tr>
</tbody>
</table>
Configuring the Algorithm

- The `hc_nswap` improves longer than `hc_1swap`

<table>
<thead>
<tr>
<th>$\mathcal{I}$</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>hc_1swap</td>
<td>717</td>
<td>800</td>
</tr>
<tr>
<td></td>
<td>hc_nswap</td>
<td>724</td>
<td>758</td>
</tr>
<tr>
<td>la24</td>
<td>hc_1swap</td>
<td>999</td>
<td>1095</td>
</tr>
<tr>
<td></td>
<td>hc_nswap</td>
<td>945</td>
<td>1018</td>
</tr>
<tr>
<td>swv15</td>
<td>hc_1swap</td>
<td>3837</td>
<td>4108</td>
</tr>
<tr>
<td></td>
<td>hc_nswap</td>
<td>3602</td>
<td>3880</td>
</tr>
<tr>
<td>yn4</td>
<td>hc_1swap</td>
<td>1109</td>
<td>1222</td>
</tr>
<tr>
<td></td>
<td>hc_nswap</td>
<td>1095</td>
<td>1162</td>
</tr>
</tbody>
</table>
Configuring the Algorithm

- The \textit{hc\_nswap} improves longer than \textit{hc\_1swap}
- We can expect that the number $L$ of unsuccessful steps before a restart should be higher now.

<table>
<thead>
<tr>
<th>$I$</th>
<th>algo</th>
<th>best</th>
<th>mean</th>
<th>med</th>
<th>sd</th>
<th>med(t)</th>
<th>med(FEs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>abz7</td>
<td>hc_1swap</td>
<td>717</td>
<td>800</td>
<td>798</td>
<td>28</td>
<td>0s</td>
<td>16'978</td>
</tr>
<tr>
<td></td>
<td>hc_nswap</td>
<td>724</td>
<td>758</td>
<td>758</td>
<td>17</td>
<td>35s</td>
<td>7'781'762</td>
</tr>
<tr>
<td>la24</td>
<td>hc_1swap</td>
<td>999</td>
<td>1095</td>
<td>1086</td>
<td>56</td>
<td>0s</td>
<td>6'612</td>
</tr>
<tr>
<td></td>
<td>hc_nswap</td>
<td>945</td>
<td>1018</td>
<td>1016</td>
<td>29</td>
<td>25s</td>
<td>9'072'935</td>
</tr>
<tr>
<td>swv15</td>
<td>hc_1swap</td>
<td>3837</td>
<td>4108</td>
<td>4108</td>
<td>137</td>
<td>1s</td>
<td>104'598</td>
</tr>
<tr>
<td></td>
<td>hc_nswap</td>
<td>3602</td>
<td>3880</td>
<td>3872</td>
<td>112</td>
<td>70s</td>
<td>8'351'112</td>
</tr>
<tr>
<td>yn4</td>
<td>hc_1swap</td>
<td>1109</td>
<td>1222</td>
<td>1220</td>
<td>48</td>
<td>0s</td>
<td>31'789</td>
</tr>
<tr>
<td></td>
<td>hc_nswap</td>
<td>1095</td>
<td>1162</td>
<td>1160</td>
<td>34</td>
<td>71s</td>
<td>11'016'757</td>
</tr>
</tbody>
</table>
Configuring the Algorithm
Configuring the Algorithm

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

<table>
<thead>
<tr>
<th>$I$</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><code>hc_1swap</code></td>
<td>717</td>
<td>800</td>
</tr>
<tr>
<td></td>
<td><code>hc_nswap</code></td>
<td>724</td>
<td>758</td>
</tr>
<tr>
<td>la24</td>
<td><code>hc_1swap</code></td>
<td>999</td>
<td>1095</td>
</tr>
<tr>
<td></td>
<td><code>hc_nswap</code></td>
<td>945</td>
<td>1018</td>
</tr>
<tr>
<td>swv15</td>
<td><code>hc_1swap</code></td>
<td>3837</td>
<td>4108</td>
</tr>
<tr>
<td></td>
<td><code>hc_nswap</code></td>
<td>3602</td>
<td>3880</td>
</tr>
<tr>
<td>yn4</td>
<td><code>hc_1swap</code></td>
<td>1109</td>
<td>1222</td>
</tr>
<tr>
<td></td>
<td><code>hc_nswap</code></td>
<td>1095</td>
<td>1162</td>
</tr>
</tbody>
</table>
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>$\mathcal{I}$</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><strong>hcr_16384_1swap</strong></td>
<td>714</td>
<td>732</td>
</tr>
<tr>
<td></td>
<td><strong>hc_nswap</strong></td>
<td>724</td>
<td>758</td>
</tr>
<tr>
<td></td>
<td><strong>hcr_65536_nswap</strong></td>
<td>712</td>
<td>731</td>
</tr>
<tr>
<td>la24</td>
<td><strong>hcr_16384_1swap</strong></td>
<td>953</td>
<td>976</td>
</tr>
<tr>
<td></td>
<td><strong>hc_nswap</strong></td>
<td>945</td>
<td>1018</td>
</tr>
<tr>
<td></td>
<td><strong>hcr_65536_nswap</strong></td>
<td>942</td>
<td>973</td>
</tr>
<tr>
<td>swv15</td>
<td><strong>hcr_16384_1swap</strong></td>
<td>3752</td>
<td>3859</td>
</tr>
<tr>
<td></td>
<td><strong>hc_nswap</strong></td>
<td>3602</td>
<td>3880</td>
</tr>
<tr>
<td></td>
<td><strong>hcr_65536_nswap</strong></td>
<td>3740</td>
<td>3818</td>
</tr>
<tr>
<td>yn4</td>
<td><strong>hcr_16384_1swap</strong></td>
<td>1081</td>
<td>1115</td>
</tr>
<tr>
<td></td>
<td><strong>hc_nswap</strong></td>
<td>1095</td>
<td>1162</td>
</tr>
<tr>
<td></td>
<td><strong>hcr_65536_nswap</strong></td>
<td>1068</td>
<td>1109</td>
</tr>
</tbody>
</table>
So what do we get?

hcr_16384_1swap: median result of 3 min of hcr_16384_1swap
So what do we get?

hcr_65536_nswap: median result of 3 min of hcr_65536_nswap
So what do we get?

**hcr_16384_1swap**: median result of 3 min of **hcr_16384_1swap**
So what do we get?

hcr_65536_nswap: median result of 3 min of hcr_65536_nswap
So what do we get?

hcr_16384_1swap: median result of 3 min of hcr_16384_1swap
So what do we get?

hcr_65536_nswap: median result of 3 min of hcr_65536_nswap
So what do we get?

hcr_16384_1swap: median result of 3 min of hcr_16384_1swap
So what do we get?

hcr_65536_nswap: median result of 3 min of hcr_65536_nswap
Progress over Time

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

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

What progress does the algorithm make over time?

hcr_nswap tends to be a tiny little bit better than hcr_1swap ... but not much
hcr_nswap tends to be a tiny little bit better than hcr_1swap ... but not much
Progress over Time

What progress does the algorithm make over time?

hcr_nswap tends to be a tiny little bit better than hcr_1swap . . . but not much
Progress over Time

What progress does the algorithm make over time?

hcr_nswap tends to be a tiny little bit better than hcr_1swap ... but not much
Progress over Time

What progress does the algorithm make over time?

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

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

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

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

• By making use of the best point in the search space we have seen so far and iteratively trying to improve it, we can dramatically improve the results compared to random sampling.

• Like random sampling, we can apply it to all sorts of problems, as long as we provide the basic structural ingredients.
• We now have learned a second, more efficient metaheuristic optimization algorithm: stochastic hill climber.

• By making use of the best point in the search space we have seen so far and iteratively trying to improve it, we can dramatically improve the results compared to random sampling.

• Like random sampling, we can apply it to all sorts of problems, as long as we provide the basic structural ingredients.

• Hill climbing is a local search and vulnerable to get trapped in local optima.
Summary

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

Thank you
References


