合肥學院
HEFEI UNIVERSITY

Optimization Algorithms

4. Random Sampling

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

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

Outline

Introduction

Introduction

- We will now learn our very first optimization algorithm.

Introduction

- We will now learn our very first optimization algorithm.
- We already have the basic tools: We can represent a Gantt chart for m machines and n jobs as an integer string of length $m \times n$.

Introduction

- We will now learn our very first optimization algorithm.
- We already have the basic tools: We can represent a Gantt chart for m machines and n jobs as an integer string of length $m \times n$.
- How does this help us to search?

Introduction

- We will now learn our very first optimization algorithm.
- We already have the basic tools: We can represent a Gantt chart for m machines and n jobs as an integer string of length $m \times n$.
- How does this help us to search?
- Well, we can first try the trivial thing: create a random solution!

Introduction

- We will now learn our very first optimization algorithm.
- We already have the basic tools: We can represent a Gantt chart for m machines and n jobs as an integer string of length $m \times n$.
- How does this help us to search?
- Well, we can first try the trivial thing: create a random solution!
- We can therefore
    1. put each of the numbers from 0 to $n-1$ exactly m times in an integer array of length $m * n$

Introduction

- We will now learn our very first optimization algorithm.
- We already have the basic tools: We can represent a Gantt chart for m machines and n jobs as an integer string of length $m \times n$.
- How does this help us to search?
- Well, we can first try the trivial thing: create a random solution!
- We can therefore
    1. put each of the numbers from 0 to $n-1$ exactly m times in an integer array of length $m * n$ (so we have a valid point $x_0 \in \mathbb{X}$)

Introduction

- We will now learn our very first optimization algorithm.
- We already have the basic tools: We can represent a Gantt chart for m machines and n jobs as an integer string of length $m \times n$.
- How does this help us to search?
- Well, we can first try the trivial thing: create a random solution!
- We can therefore
    1. put each of the numbers from 0 to $n-1$ exactly m times in an integer array of length $m * n$ (so we have a valid point $x_0 \in \mathbb{X}$), then
    2. randomly shuffle the values like a deck of cards

Introduction

- We will now learn our very first optimization algorithm.
- We already have the basic tools: We can represent a Gantt chart for m machines and n jobs as an integer string of length $m \times n$.
- How does this help us to search?
- Well, we can first try the trivial thing: create a random solution!
- We can therefore
  1. put each of the numbers from 0 to $n-1$ exactly m times in an integer array of length $m * n$ (so we have a valid point $x_0 \in \mathbb{X}$), then
  2. randomly shuffle the values like a deck of cards (so we get a random valid point $x \in \mathbb{X}$)

Introduction

- We will now learn our very first optimization algorithm.
- We already have the basic tools: We can represent a Gantt chart for m machines and n jobs as an integer string of length $m \times n$.
- How does this help us to search?
- Well, we can first try the trivial thing: create a random solution!
- We can therefore
  1. put each of the numbers from 0 to $n-1$ exactly m times in an integer array of length $m * n$ (so we have a valid point $x_0 \in \mathbb{X}$), then
  2. randomly shuffle the values like a deck of cards (so we get a random valid point $x \in \mathbb{X}$), and
  3. apply the representation mapping γ to get a Gantt chart $y = \gamma(x)$, $y \in \mathbb{Y}$.

Algorithm Concept

Interface for a Function to Sample 1 Point from $\mathbb{X}$

- We already have the interface that we need to implement to do such a thing

Interface for a Function to Sample 1 Point from $\mathbb{X}$

- We already have the interface that we need to implement to do such a thing: the `INullarySearchOperator`

Interface for a Function to Sample 1 Point from $\mathbb{X}$

- We already have the interface that we need to implement to do such a thing: the `INullarySearchOperator`

```
package aitoa.structure;

public interface INullarySearchOperator<X> {

  void apply(X dest, Random random);

}
```

Implementation: Create Random Point in $\mathbb{X}$

```
public class JSSPNullaryOperator {
//
// unnecessary stuff omitted here...
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
}
```

Implementation: Create Random Point in $\mathbb{X}$

```java
public class JSSPNullaryOperator implements
    INullarySearchOperator<int[]> {
// unnecessary stuff omitted here...
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
}
```

Implementation: Create Random Point in $\mathbb{X}$

```
public class JSSPNullaryOperator implements
    INullarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] dest, Random random) {
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
  }
}
```

Implementation: Create Random Point in $\mathbb{X}$

```
public class JSSPNullaryOperator implements
    INullarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] dest, Random random) {
// fill first part of array with 0, 1, 2, ..., n
//
//
//
//
//
//
//
//
//
//
//
//
//
//
  }
}
```

Implementation: Create Random Point in $\mathbb{X}$

```
public class JSSPNullaryOperator implements
    INullarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] dest, Random random) {
// fill first part of array with 0, 1, 2, ..., n
    for (int i = this.n; (--i) >= 0;) {
      dest[i] = i;
    }
//
//
//
//
//
//
//
//
//
//
//
  }
}
```

Implementation: Create Random Point in $\mathbb{X}$

```
public class JSSPNullaryOperator implements
    INullarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] dest, Random random) {
// fill first part of array with 0, 1, 2, ..., n
    for (int i = this.n; (--i) >= 0;) {
      dest[i] = i;
    }
// copy this part m-1 times
//
//
//
//
//
//
//
//
//
//
  }
}
```

Implementation: Create Random Point in $\mathbb{X}$

```
public class JSSPNullaryOperator implements
    INullarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] dest, Random random) {
// fill first part of array with 0, 1, 2, ..., n
    for (int i = this.n; (--i) >= 0;) {
      dest[i] = i;
    }
// copy this part m-1 times
    for (int i = dest.length; (i -= this.n) > 0;) {
      System.arraycopy(dest, 0, dest, i, this.n);
    }
//
//
//
//
//
//
//
  }
}
```

Implementation: Create Random Point in $\mathbb{X}$

```
public class JSSPNullaryOperator implements
    INullarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] dest, Random random) {
// fill first part of array with 0, 1, 2, ..., n
    for (int i = this.n; (--i) >= 0;) {
      dest[i] = i;
    }
// copy this part m-1 times
    for (int i = dest.length; (i -= this.n) > 0;) {
      System.arraycopy(dest, 0, dest, i, this.n);
    }
// randomly shuffle the array: Fisher-Yates shuffle[34]
//
//
//
//
//
//
  }
}
```

Implementation: Create Random Point in $\mathbb{X}$

```
public class JSSPNullaryOperator implements
    INullarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] dest, Random random) {
// fill first part of array with 0, 1, 2, ..., n
    for (int i = this.n; (--i) >= 0;) {
      dest[i] = i;
    }
// copy this part m-1 times
    for (int i = dest.length; (i -= this.n) > 0;) {
      System.arraycopy(dest, 0, dest, i, this.n);
    }
// randomly shuffle the array: Fisher-Yates shuffle[34]
    for (int i = dest.length; i > 1;) {
//
//
//
//
    }
  }
}
```

Implementation: Create Random Point in $\mathbb{X}$

```java
public class JSSPNullaryOperator implements
    INullarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] dest, Random random) {
// fill first part of array with 0, 1, 2, ..., n
    for (int i = this.n; (--i) >= 0;) {
      dest[i] = i;
    }
// copy this part m-1 times
    for (int i = dest.length; (i -= this.n) > 0;) {
      System.arraycopy(dest, 0, dest, i, this.n);
    }
// randomly shuffle the array: Fisher-Yates shuffle[34]
    for (int i = dest.length; i > 1;) {
      int j     = random.nextInt(i--);
      int t     = array[i];
//
//
    }
  }
}
```

Implementation: Create Random Point in $\mathbb{X}$

```java
public class JSSPNullaryOperator implements
    INullarySearchOperator<int[]> {
// unnecessary stuff omitted here...
  public void apply(int[] dest, Random random) {
// fill first part of array with 0, 1, 2, ..., n
    for (int i = this.n; (--i) >= 0;) {
      dest[i] = i;
    }
// copy this part m-1 times
    for (int i = dest.length; (i -= this.n) > 0;) {
      System.arraycopy(dest, 0, dest, i, this.n);
    }
// randomly shuffle the array: Fisher-Yates shuffle[34]
    for (int i = dest.length; i > 1;) {
      int j     = random.nextInt(i--);
      int t     = array[i];
      array[i] = array[j];
      array[j] = t;
    } // implemented as RandomUtils.shuffle in code repo
  }
}
```

Implementation: Single Random Sampling Algorithm

```java
package aitoa.algorithms;

public class SingleRandomSample<X, Y> {
//
// unnecessary stuff (e.g., constructor) omitted here...
//
//

//

//
//
}
```

Implementation: Single Random Sampling Algorithm

```
package aitoa.algorithms;

public class SingleRandomSample<X, Y> extends
    Metaheuristic0<X, Y> {
// unnecessary stuff (e.g., constructor) omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
//

//

//
  }
}
```

Implementation: Single Random Sampling Algorithm

```
package aitoa.algorithms;

public class SingleRandomSample<X, Y> extends
    Metaheuristic0<X, Y> {
// unnecessary stuff (e.g., constructor) omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
    X x = process.getSearchSpace().create(); // allocate

//

//
  }
}
```

Implementation: Single Random Sampling Algorithm

```
package aitoa.algorithms;

public class SingleRandomSample<X, Y> extends
    Metaheuristic0<X, Y> {
// unnecessary stuff (e.g., constructor) omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
    X x = process.getSearchSpace().create(); // allocate

    this.nullary.apply(x, process.getRandom());

//
  }
}
```

Implementation: Single Random Sampling Algorithm

```
package aitoa.algorithms;

public class SingleRandomSample<X, Y> extends
    Metaheuristic0<X, Y> {
// unnecessary stuff (e.g., constructor) omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
    X x = process.getSearchSpace().create(); // allocate

    this.nullary.apply(x, process.getRandom());

    process.evaluate(x); // evaluate
  }
}
```

Experiment and Analysis

So what do we get?

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

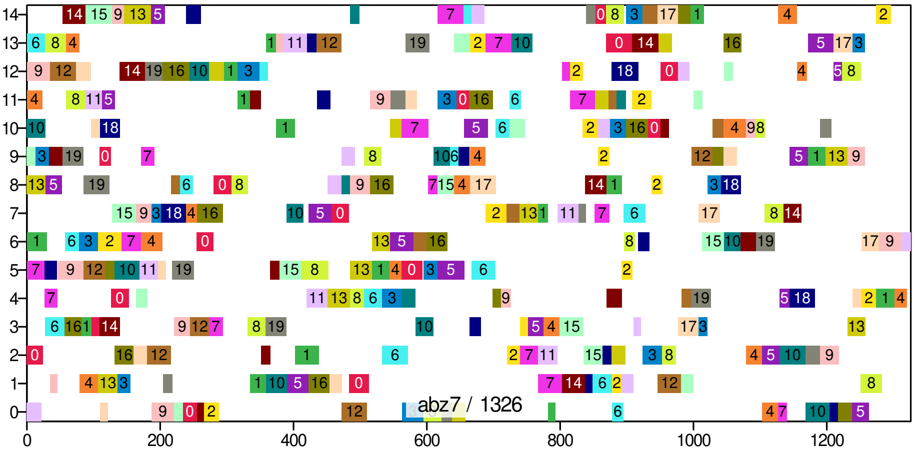
So what do we get?

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

	makespan				last improvement	
$\mathcal{I}$	best	mean	med	sd	med(t)	med(FEs)
abz7	1'131	1'334	1'326	106	0s	1
la24	1'487	1'842	1'814	165	0s	1
swv15	5'935	6'600	6'563	346	0s	1
yn4	1'754	2'036	2'039	125	0s	1

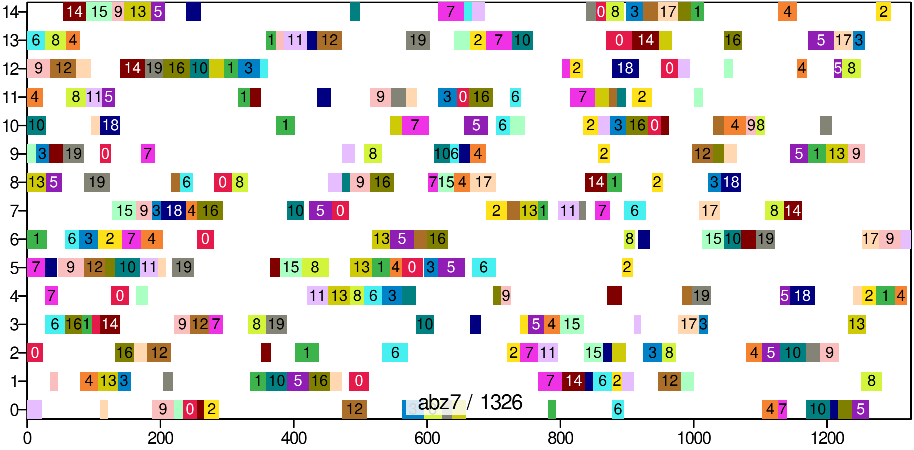
So what do we get?

Median solution for abz7



So what do we get?

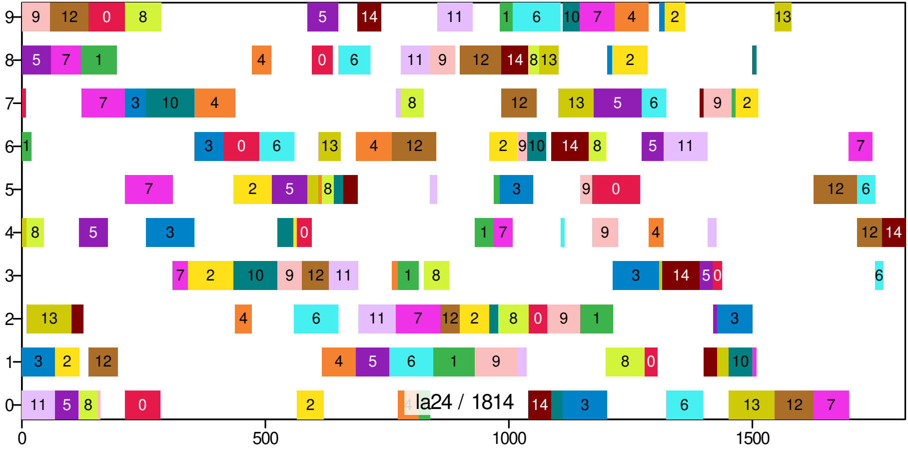
Median solution for abz7



. . . there is lots of white space between the operations. . .

So what do we get?

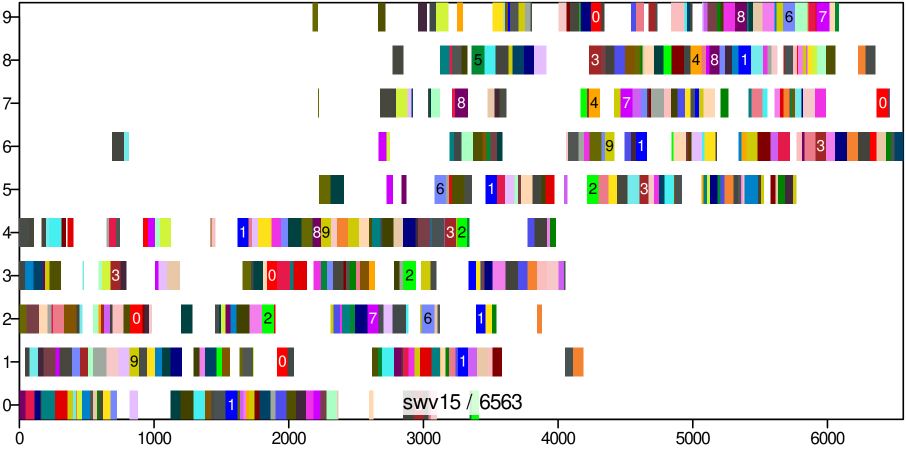
Median solution for `la24`



. . . there is lots of white space between the operations. . .

So what do we get?

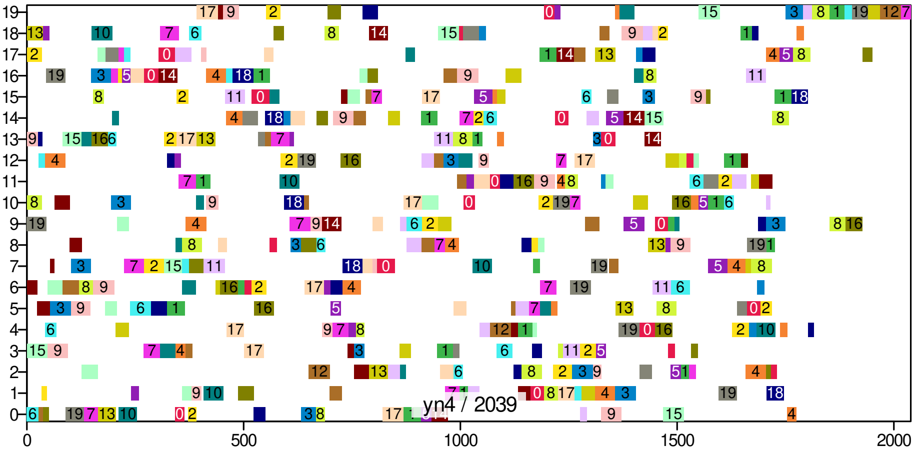
Median solution for `swv15`



. . . there is lots of white space between the operations. . .

So what do we get?

Median solution for `yn4`



. . . there is lots of white space between the operations. . .

So what do we get?

- I execute the program 101 times for each of the instances `abz7`, `la24`, `swv15`, and `yn4`
- The results are not good, there is lots of white space $\equiv$ wasted time.

	makespan				**last improvement**	
$\mathcal{I}$	best	mean	med	sd	med(t)	med(FEs)
`abz7`	1'131	1'334	1'326	106	0s	1
`la24`	1'487	1'842	1'814	165	0s	1
`swv15`	5'935	6'600	6'563	346	0s	1
`yn4`	1'754	2'036	2'039	125	0s	1

So what do we get?

- I execute the program 101 times for each of the instances `abz7`, `la24`, `swv15`, and `yn4`
- The results are not good, there is lots of white space $\equiv$ wasted time. That was expected: Our solutions are random.

	makespan				last improvement	
$\mathcal{I}$	best	mean	med	sd	med(t)	med(FEs)
`abz7`	1'131	1'334	1'326	106	0s	1
`la24`	1'487	1'842	1'814	165	0s	1
`swv15`	5'935	6'600	6'563	346	0s	1
`yn4`	1'754	2'036	2'039	125	0s	1

So what do we get?

- I execute the program 101 times for each of the instances `abz7`, `la24`, `swv15`, and `yn4`
- The results are not good, there is lots of white space $\equiv$ wasted time. That was expected: Our solutions are random.
- Notice 1. We can create and test the schedules very very fast (much faster than 3min).

	makespan				last improvement	
$\mathcal{I}$	best	mean	med	sd	med(t)	med(FEs)
`abz7`	1'131	1'334	1'326	106	0s	1
`la24`	1'487	1'842	1'814	165	0s	1
`swv15`	5'935	6'600	6'563	346	0s	1
`yn4`	1'754	2'036	2'039	125	0s	1

So what do we get?

- I execute the program 101 times for each of the instances `abz7`, `la24`, `swv15`, and `yn4`
- The results are not good, there is lots of white space $\equiv$ wasted time. That was expected: Our solutions are random.
- Notice 1. We can create and test the schedules very very fast (much faster than 3min).
- Notice 2. There is a high variance in the results due to randomness.

	makespan				**last improvement**	
$\mathcal{I}$	best	mean	med	sd	med(t)	med(FEs)
abz7	1'131	1'334	1'326	106	0s	1
la24	1'487	1'842	1'814	165	0s	1
swv15	5'935	6'600	6'563	346	0s	1
yn4	1'754	2'036	2'039	125	0s	1

Improved Algorithm Concept

Exploit Variance: Random Sampling

- If we can generate solutions fast ($med(t) \approx 0$) and sometimes are lucky, sometimes not ($sd \gg 0$)...

Exploit Variance: Random Sampling

- If we can generate solutions fast ($med(t) \approx 0$) and sometimes are lucky, sometimes not ($sd \gg 0$)...
- ...then why don't we keep generating schedules until the 3 minutes are up and keep the best one?

Exploit Variance: Random Sampling

- If we can generate solutions fast ($med(t) \approx 0$) and sometimes are lucky, sometimes not ($sd \gg 0$)...
- ...then why don't we keep generating schedules until the 3 minutes are up and keep the best one?
- New idea

Exploit Variance: Random Sampling

- If we can generate solutions fast ($med(t) \approx 0$) and sometimes are lucky, sometimes not ($sd \gg 0$)...
- ...then why don't we keep generating schedules until the 3 minutes are up and keep the best one?
- New idea: The Random sampling algorithm (also called random search) repeats creating random solutions until the computational budget is exhausted[5].

Exploit Variance: Random Sampling

- If we can generate solutions fast ($med(t) \approx 0$) and sometimes are lucky, sometimes not ($sd \gg 0$)…
- …then why don't we keep generating schedules until the 3 minutes are up and keep the best one?
- New idea: The Random sampling algorithm (also called random search) repeats creating random solutions until the computational budget is exhausted[5].
- It works as follows

Exploit Variance: Random Sampling

- If we can generate solutions fast ($med(t) \approx 0$) and sometimes are lucky, sometimes not ($sd \gg 0$)...
- ...then why don't we keep generating schedules until the 3 minutes are up and keep the best one?
- New idea: The Random sampling algorithm (also called random search) repeats creating random solutions until the computational budget is exhausted[5].
- It works as follows:
  1. create new random candidate solution y (via random sampling from the search space)

Exploit Variance: Random Sampling

- If we can generate solutions fast ($med(t) \approx 0$) and sometimes are lucky, sometimes not ($sd \gg 0$)...
- ...then why don't we keep generating schedules until the 3 minutes are up and keep the best one?
- New idea: The Random sampling algorithm (also called random search) repeats creating random solutions until the computational budget is exhausted[5].
- It works as follows:
  1. create new random candidate solution y (via random sampling from the search space)
  2. remember best solution ever encountered

Exploit Variance: Random Sampling

- If we can generate solutions fast ($med(t) \approx 0$) and sometimes are lucky, sometimes not ($sd \gg 0$)...
- ...then why don't we keep generating schedules until the 3 minutes are up and keep the best one?
- New idea: The Random sampling algorithm (also called random search) repeats creating random solutions until the computational budget is exhausted[5].
- It works as follows:
  1. create new random candidate solution y (via random sampling from the search space)
  2. remember best solution ever encountered
  3. repeat until 3 min are exhausted

Random Sampling Algorithm

```
package aitoa.algorithms;

public class RandomSampling<X, Y> {
//
// unnecessary stuff (e.g., constructor) omitted here...
//
//

//

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

Random Sampling Algorithm

```
package aitoa.algorithms;

public class RandomSampling<X, Y> extends Metaheuristic0<X,
    Y> {
// unnecessary stuff (e.g., constructor) omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
//

//

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

Random Sampling Algorithm

```java
package aitoa.algorithms;

public class RandomSampling<X, Y> extends Metaheuristic0<X,
    Y> {
// unnecessary stuff (e.g., constructor) omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
    X x = process.getSearchSpace().create();

//

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

Random Sampling Algorithm

```
package aitoa.algorithms;

public class RandomSampling<X, Y> extends Metaheuristic0<X,
    Y> {
// unnecessary stuff (e.g., constructor) omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
    X x = process.getSearchSpace().create();

    Random random = process.getRandom();

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

Random Sampling Algorithm

```java
package aitoa.algorithms;

public class RandomSampling<X, Y> extends Metaheuristic0<X,
    Y> {
// unnecessary stuff (e.g., constructor) omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
    X x = process.getSearchSpace().create();

    Random random = process.getRandom();

    do {
//
//
    } while (!process.shouldTerminate());
  }
}
```

Random Sampling Algorithm

```java
package aitoa.algorithms;

public class RandomSampling<X, Y> extends Metaheuristic0<X,
    Y> {
// unnecessary stuff (e.g., constructor) omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
    X x = process.getSearchSpace().create();

    Random random = process.getRandom();

    do {
      this.nullary.apply(x, random);
//
    } while (!process.shouldTerminate());
  }
}
```

Random Sampling Algorithm

```java
package aitoa.algorithms;

public class RandomSampling<X, Y> extends Metaheuristic0<X,
    Y> {
// unnecessary stuff (e.g., constructor) omitted here...
  public void solve(IBlackBoxProcess<X, Y> process) {
    X x = process.getSearchSpace().create();

    Random random = process.getRandom();

    do {
      this.nullary.apply(x, random);
      process.evaluate(x);
    } while (!process.shouldTerminate());
  }
}
```

Experiment and Analysis 2

So what do we get?

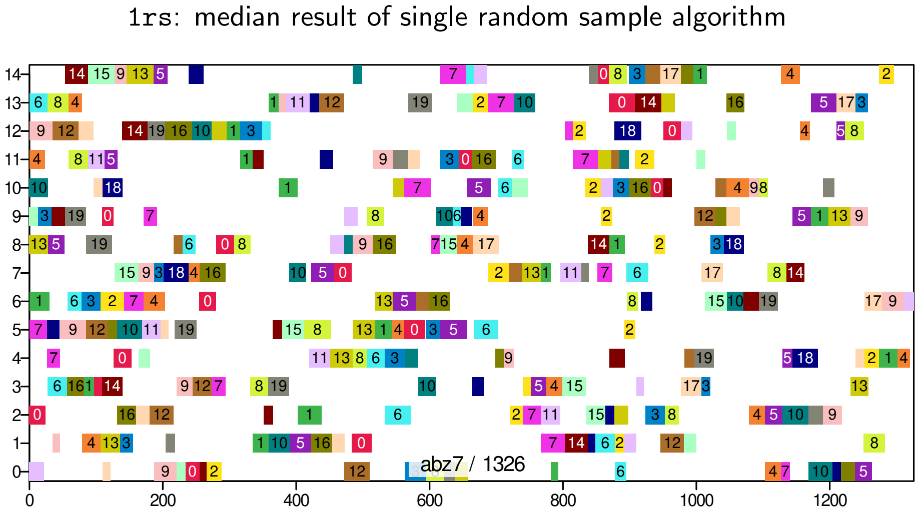
- I execute the program 101 times for each of the instances `abz7`, `la24`, `swv15`, and `yn4`

So what do we get?

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

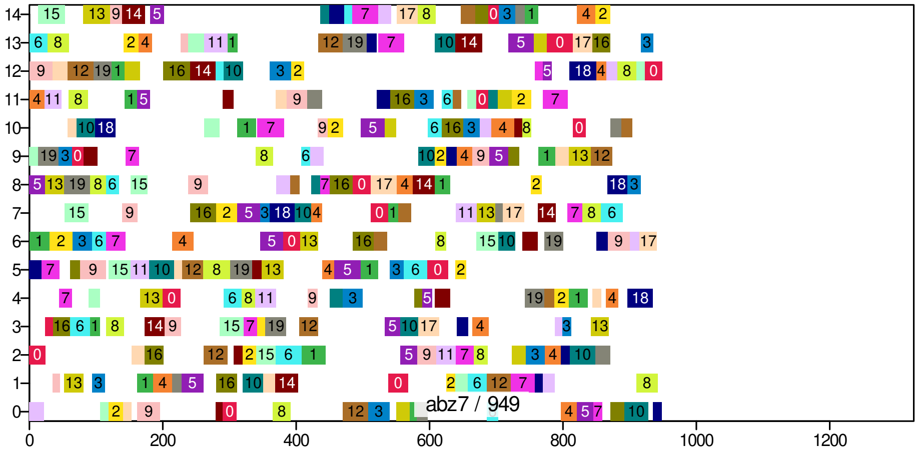
		makespan				last improvement	
$\mathcal{I}$	algo	best	mean	med	sd	med(t)	med(FEs)
`abz7`	`1rs`	1131	1334	1326	106	0s	1
	`rs`	**895**	**947**	**949**	**12**	85s	6'512'505
`la24`	`1rs`	1487	1842	1814	165	0s	1
	`rs`	**1153**	**1206**	**1208**	**15**	82s	15'902'911
`swv15`	`1rs`	5935	6600	6563	346	0s	1
	`rs`	**4988**	**5166**	**5172**	**50**	87s	5'559'124
`yn4`	`1rs`	1754	2036	2039	125	0s	1
	`rs`	**1460**	**1498**	**1499**	**15**	76s	4'814'914

So what do we get?



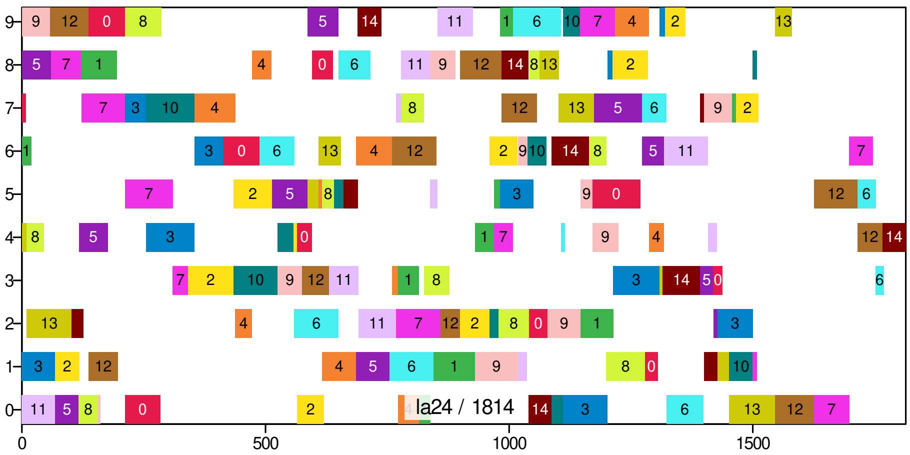
So what do we get?

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



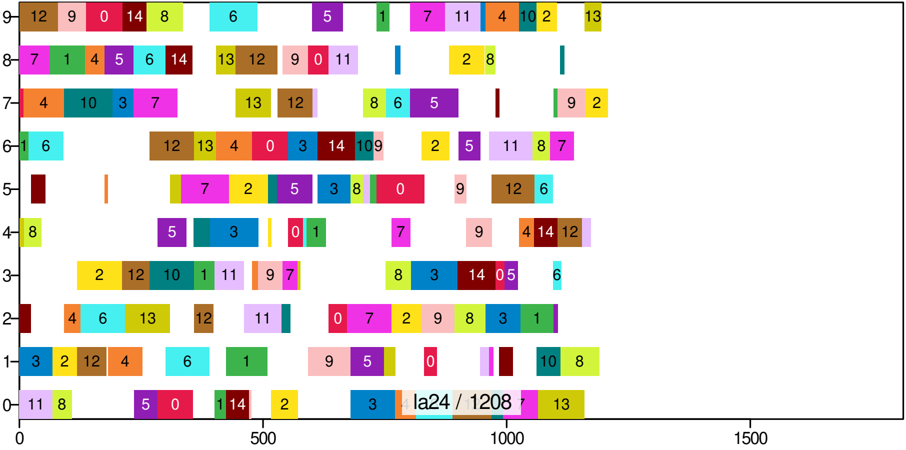
So what do we get?

1rs: median result of single random sample algorithm

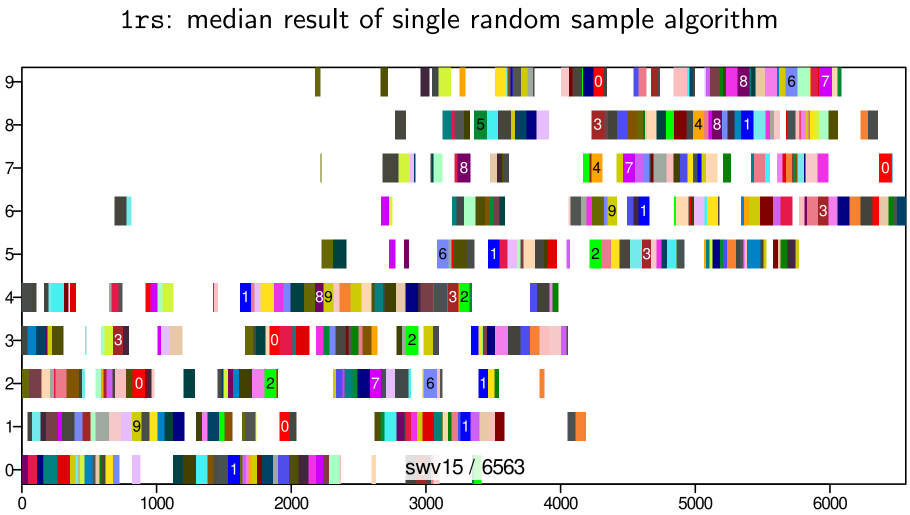


So what do we get?

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

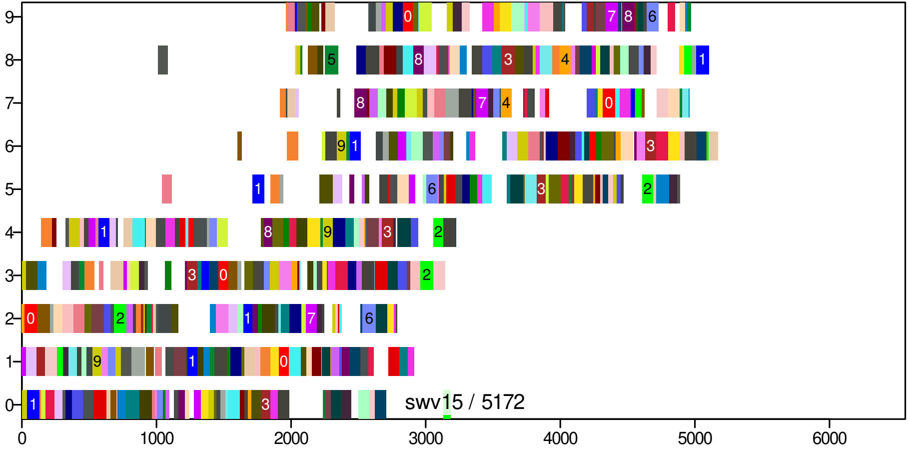


So what do we get?

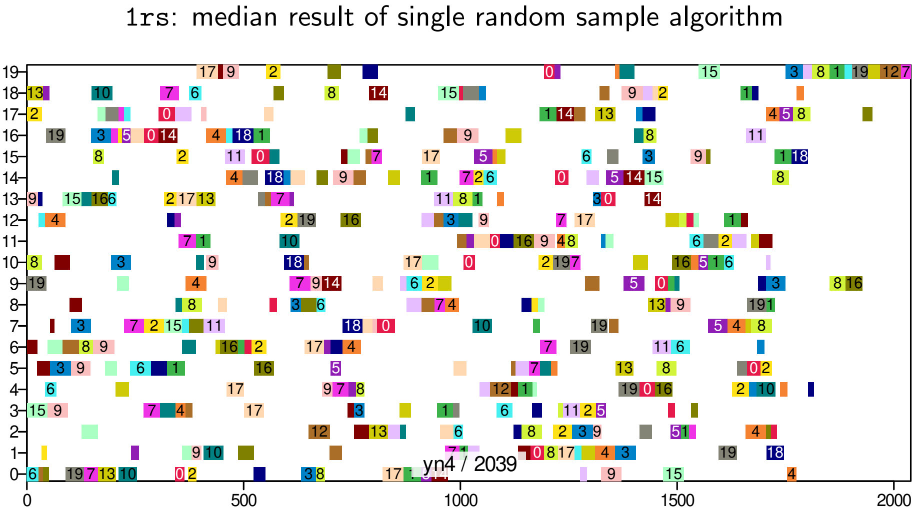


So what do we get?

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

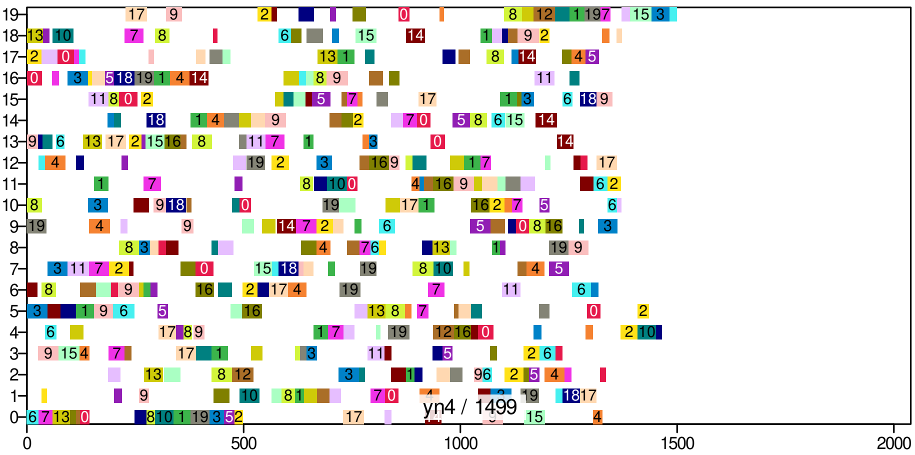


So what do we get?



So what do we get?

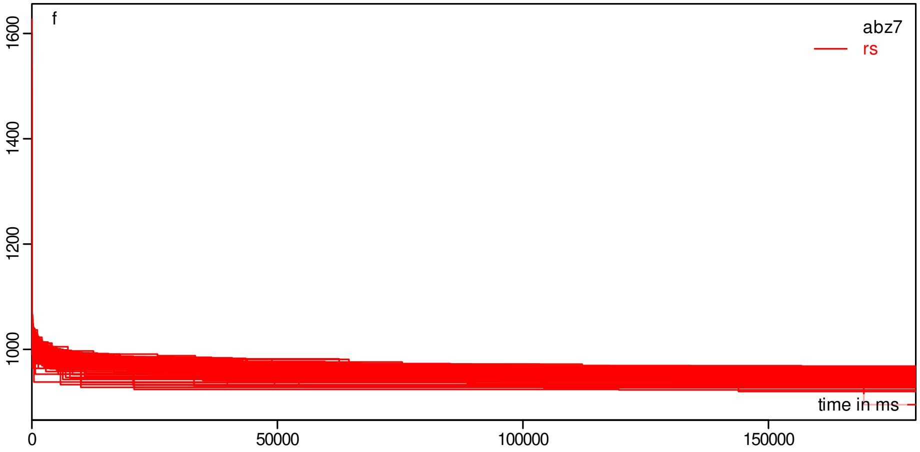
`rs`: median result of 3 min of random sampling algorithm



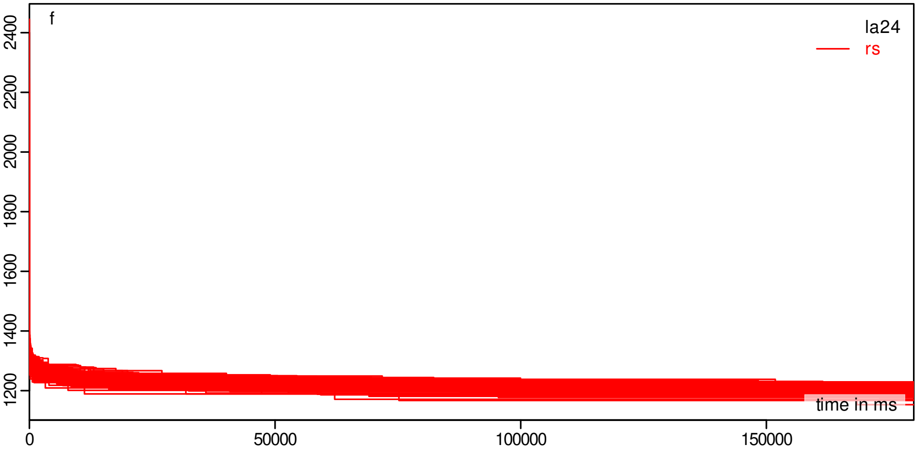
Progress over Time

What progress does the algorithm make over time?

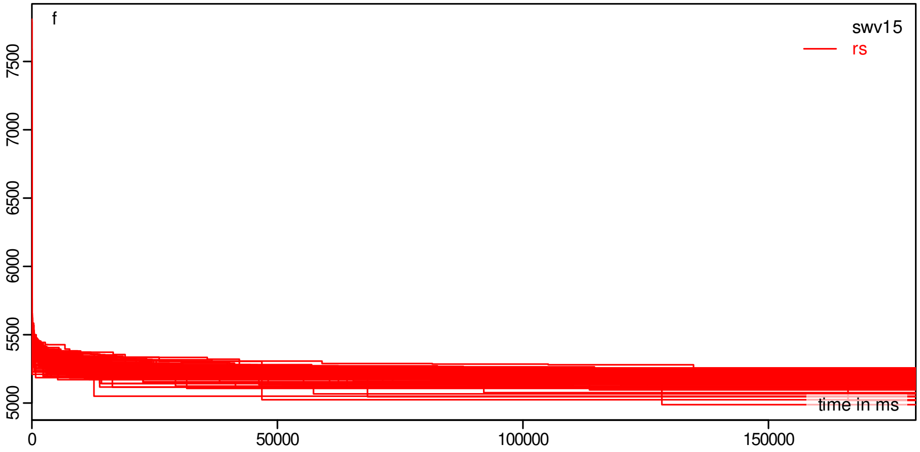
Progress over Time



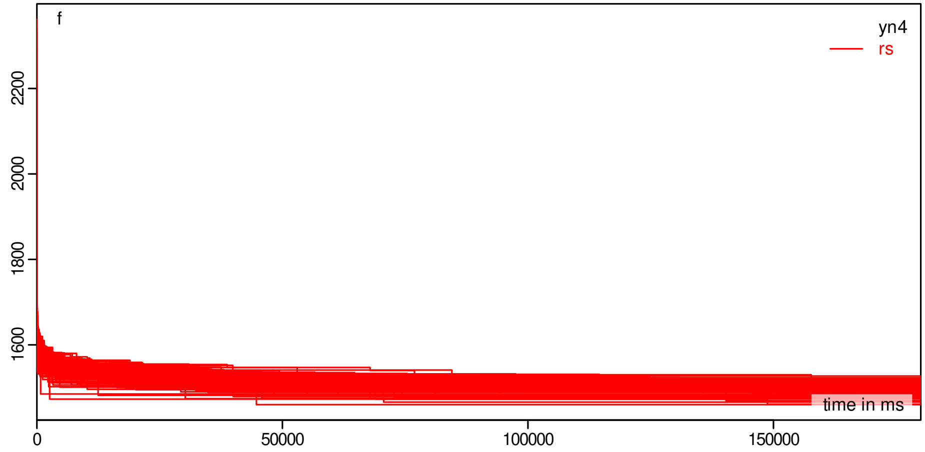
Progress over Time



Progress over Time



Progress over Time



Progress over Time

- Law of Diminishing Returns[6]

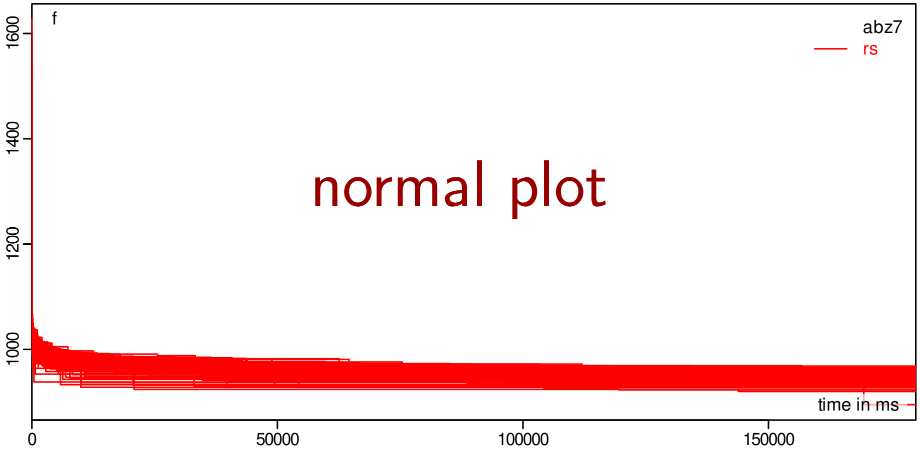
Progress over Time

- Law of Diminishing Returns[6]: Most improvements (of the makespan) are achieved with the initial, small investment (of runtime).

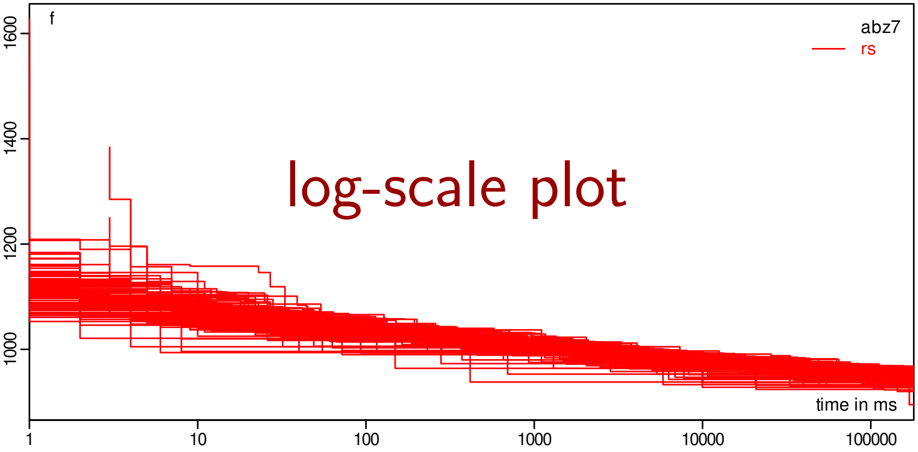
Progress over Time

- Law of Diminishing Returns[6]: Most improvements (of the makespan) are achieved with the initial, small investment (of runtime). Further improvements will cost more and more (time) and will be smaller and smaller.

Progress over Time

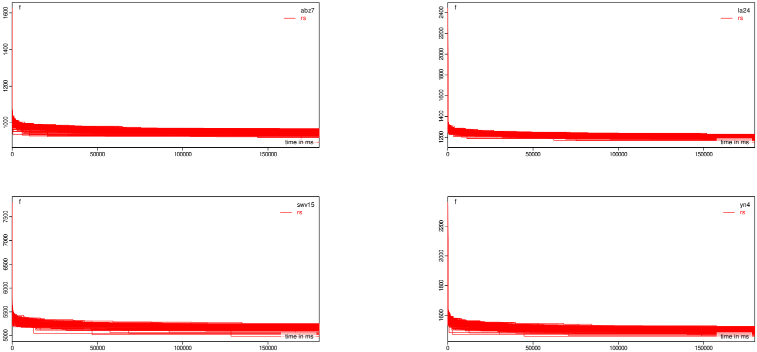


Progress over Time



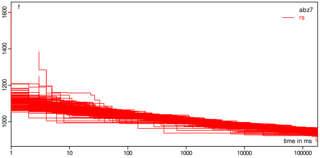
Progress over Time

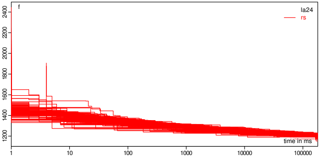
- Law of Diminishing Returns[6]: Most improvements (of the makespan) are achieved with the initial, small investment (of runtime). Further improvements will cost more and more (time) and will be smaller and smaller.

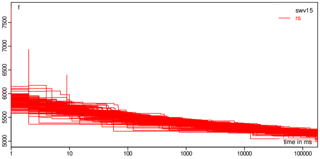


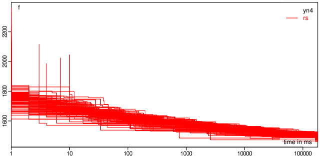
Progress over Time

- Law of Diminishing Returns[6]: Most improvements (of the makespan) are achieved with the initial, small investment (of runtime). Further improvements will cost more and more (time) and will be smaller and smaller.



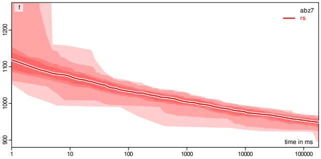


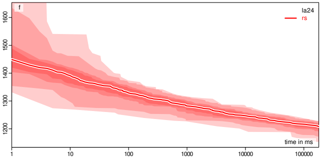


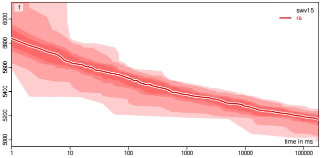


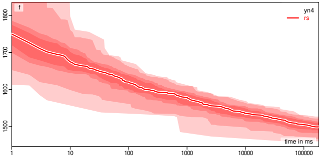
Progress over Time

- Law of Diminishing Returns[6]: Most improvements (of the makespan) are achieved with the initial, small investment (of runtime). Further improvements will cost more and more (time) and will be smaller and smaller.
- This holds for runtime, but also for improvements of algorithms.









Summary

Summary

- In this lesson, we have learned three things

Summary

- In this lesson, we have learned three things
    1. a first algorithm for solving optimization: random sampling.

Summary

- In this lesson, we have learned three things
    1. a first algorithm for solving optimization: random sampling.
    2. a tool to improve algorithm performance: restarts.

Summary

- In this lesson, we have learned three things
  1. a first algorithm for solving optimization: random sampling.
  2. a tool to improve algorithm performance: restarts.
  3. an inherent nature of optimization processes: much progress early, fewer and smaller improvements later.

Summary: Random Sampling

- With random sampling, we now have a basic algorithm that provides some solutions.

Summary: Random Sampling

- With random sampling, we now have a basic algorithm that provides some solutions.
- But it is ... well ... quite stupid.

Summary: Random Sampling

- With random sampling, we now have a basic algorithm that provides some solutions.
- But it is ... well ... quite stupid.
- It just makes random guesses.

Summary: Random Sampling

- With random sampling, we now have a basic algorithm that provides some solutions.
- But it is . . . well . . . quite stupid.
- It just makes random guesses.
- It does not make any use of the information it has seen during the search.

Summary: Random Sampling

- With random sampling, we now have a basic algorithm that provides some solutions.
- But it is . . . well . . . quite stupid.
- It just makes random guesses.
- It does not make any use of the information it has seen during the search.
- Random Sampling has two very important uses, though

Summary: Random Sampling

- With random sampling, we now have a basic algorithm that provides some solutions.
- But it is ... well ... quite stupid.
- It just makes random guesses.
- It does not make any use of the information it has seen during the search.
- Random Sampling has two very important uses, though:
    1. If an optimization problem has no structure whatsoever, if knowledge of existing good solutions is not helpful to find new good solutions in any way, then we cannot really do better than Random Sampling!

Summary: Random Sampling

- With random sampling, we now have a basic algorithm that provides some solutions.
- But it is ... well ... quite stupid.
- It just makes random guesses.
- It does not make any use of the information it has seen during the search.
- Random Sampling has two very important uses, though:
    1. If an optimization problem has no structure whatsoever, if knowledge of existing good solutions is not helpful to find new good solutions in any way, then we cannot really do better than Random Sampling!
    2. In most relevant optimization problems, however, such information is helpful.

Summary: Random Sampling

- With random sampling, we now have a basic algorithm that provides some solutions.
- But it is . . . well . . . quite stupid.
- It just makes random guesses.
- It does not make any use of the information it has seen during the search.
- Random Sampling has two very important uses, though:
    1. If an optimization problem has no structure whatsoever, if knowledge of existing good solutions is not helpful to find new good solutions in any way, then we cannot really do better than Random Sampling!
    2. In most relevant optimization problems, however, such information is helpful. An optimization algorithm is only reasonable if it is significantly better than Random Sampling.

Summary: Restarts

- We started with an algorithm that created a single random solution.

Summary: Restarts

- We started with an algorithm that created a single random solution. Let's call this algorithm $\mathcal{A}$.

Summary: Restarts

- We started with an algorithm that created a single random solution. Let's call this algorithm $\mathcal{A}$.
- We then wrapped a loop around $\mathcal{A}$, we restarted $\mathcal{A}$ again and again until the time was up.

Summary: Restarts

- We started with an algorithm that created a single random solution. Let's call this algorithm $\mathcal{A}$.
- We then wrapped a loop around $\mathcal{A}$, we restarted $\mathcal{A}$ again and again until the time was up (and of course, remembered the best solution).

Summary: Restarts

- We started with an algorithm that created a single random solution. Let's call this algorithm $\mathcal{A}$.
- We then wrapped a loop around $\mathcal{A}$, we restarted $\mathcal{A}$ again and again until the time was up (and of course, remembered the best solution).
- This is actually basic strategy of "algorithm $\mathcal{B} =$ a restarted algorithm $\mathcal{A}$", a tool that we have available from now on!

Summary: Restarts

- We started with an algorithm that created a single random solution. Let's call this algorithm $\mathcal{A}$.
- We then wrapped a loop around $\mathcal{A}$, we restarted $\mathcal{A}$ again and again until the time was up (and of course, remembered the best solution).
- This is actually basic strategy of "algorithm $\mathcal{B}$ = a restarted algorithm $\mathcal{A}$", a tool that we have available from now on!
- It can be applied in many scenarios, but has the following limitations

Summary: Restarts

- We started with an algorithm that created a single random solution. Let's call this algorithm $\mathcal{A}$.
- We then wrapped a loop around $\mathcal{A}$, we restarted $\mathcal{A}$ again and again until the time was up (and of course, remembered the best solution).
- This is actually basic strategy of "algorithm $\mathcal{B} =$ a restarted algorithm $\mathcal{A}$", a tool that we have available from now on!
- It can be applied in many scenarios, but has the following limitations:
    1. It only works if there is a reasonably-large variance, i.e., if different runs of $\mathcal{A}$ produce different results.

Summary: Restarts

- We started with an algorithm that created a single random solution. Let's call this algorithm $\mathcal{A}$.
- We then wrapped a loop around $\mathcal{A}$, we restarted $\mathcal{A}$ again and again until the time was up (and of course, remembered the best solution).
- This is actually basic strategy of "algorithm $\mathcal{B} =$ a restarted algorithm $\mathcal{A}$", a tool that we have available from now on!
- It can be applied in many scenarios, but has the following limitations:
  1. It only works if there is a reasonably-large variance, i.e., if different runs of $\mathcal{A}$ produce different results.
  2. It only works if $\mathcal{A}$ produces good-enough results early-enough, so that we have enough time in our budget to restart $\mathcal{A}$.

谢谢

Thank you

References I

1. Thomas Weise. *An Introduction to Optimization Algorithms*. Institute of Applied Optimization (IAO) [应用优化研究所] of the School of Artificial Intelligence and Big Data [人工智能与大数据学院] of Hefei University [合肥学院], Hefei [合肥市], Anhui [安徽省], China [中国], 2018–2020. URL `http://thomasweise.github.io/aitoa/`.
2. Thomas Weise. *Global Optimization Algorithms – Theory and Application*. it-weise.de (self-published), Germany, 2009. URL `http://www.it-weise.de/projects/book.pdf`.
3. Sir Ronald Aylmer Fisher and Frank Yates. *Statistical Tables for Biological, Agricultural and Medical Research*. Oliver & Boyd, London, UK, 3 edition, 1948.
4. Donald Ervin Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison–Wesley, Reading, MA, USA, 1969.
5. James C. Spall. *Introduction to Stochastic Search and Optimization*, volume 6 of *Estimation, Simulation, and Control – Wiley-Interscience Series in Discrete Mathematics and Optimization*. Wiley Interscience, Chichester, West Sussex, UK, April 2003. ISBN 0-471-33052-3. URL `http://www.jhuapl.edu/ISSO/`.
6. Paul Anthony Samuelson and William Dawbney Nordhaus. *Microeconomics*. McGraw-Hill Education (ISE Editions), Boston, MA, USA, 17 edition, 2001. ISBN 0071-180664.