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

1. Introduction
2. Algorithm Concept
3. Experiment and Analysis
4. Improved Algorithm Concept
5. Experiment and Analysis 2
6. Summary
Introduction
Introduction

- We will now learn our very first optimization algorithm.
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?
• 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 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 $m \times n$
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 $m \times n$ (so we have a valid point $x_0 \in X$)
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 $m \times n$ (so we have a valid point $x_0 \in X$), then
  2. randomly shuffle the values like a deck of cards
• 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 $m \times n$ (so we have a valid point $x_0 \in X$), then
  2. randomly shuffle the values like a deck of cards (so we get a random valid point $x \in X$)
• 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 \(m \times n\) (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 \(\gamma\) to get a Gantt chart \(y = \gamma(x)\), \(y \in \mathbb{Y}\).
Algorithm Concept
Interface for a Function to Sample 1 Point from \( X \)

- We already have the interface that we need to implement to do such a thing.
Interface for a Function to Sample 1 Point from \( X \)

- 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 \( X \)

- We already have the interface that we need to implement to do such a thing: the \texttt{INullarySearchOperator}

```java
package aitoa.structure;

public interface INullarySearchOperator<X> {
    void apply(X dest, Random random);
}
```
Implementation: Create Random Point in X

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

    // unnecessary stuff omitted here...

}
public class JSSPNullaryOperator implements INullarySearchOperator<int[]> {
    // unnecessary stuff omitted here...
    public void apply(int[] dest, Random random) {
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
    }
}
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
    }
}
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;
        }
    }
}
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
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
    }
}
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);
        }
    }
}
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
    }
}
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
        for (int i = dest.length; i > 1;) {
            //
            //
            //
            //
            //
        }
    }
}
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
        for (int i = dest.length; i > 1;) {
            int j = random.nextInt(i--);
            int t = array[i];
Implementation: Create Random Point in 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 shuffle³⁴
        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

```java
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) {
    //
    //
    //
    //
    }
}
```
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

```java
package aitoa.algorithms;

public class SingleRandomSample<X, Y> extends Metaheuristic<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());
    }
}
```
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

<table>
<thead>
<tr>
<th>I</th>
<th>makespan</th>
<th>last improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>best</td>
<td>mean</td>
</tr>
<tr>
<td>abz7</td>
<td>1’131</td>
<td>1’334</td>
</tr>
<tr>
<td>la24</td>
<td>1’487</td>
<td>1’842</td>
</tr>
<tr>
<td>swv15</td>
<td>5’935</td>
<td>6’600</td>
</tr>
<tr>
<td>yn4</td>
<td>1’754</td>
<td>2’036</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>(\mathcal{I})</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>1'131</td>
<td>1'334</td>
<td>1'326</td>
<td>106</td>
<td>0s</td>
<td>1</td>
</tr>
<tr>
<td>la24</td>
<td>1'487</td>
<td>1'842</td>
<td>1'814</td>
<td>165</td>
<td>0s</td>
<td>1</td>
</tr>
<tr>
<td>swv15</td>
<td>5'935</td>
<td>6'600</td>
<td>6'563</td>
<td>346</td>
<td>0s</td>
<td>1</td>
</tr>
<tr>
<td>yn4</td>
<td>1'754</td>
<td>2'036</td>
<td>2'039</td>
<td>125</td>
<td>0s</td>
<td>1</td>
</tr>
</tbody>
</table>
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 ≡ wasted time. That was expected: Our solutions are random.

<table>
<thead>
<tr>
<th>(I)</th>
<th>makespan</th>
<th>last improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>best</td>
<td>mean</td>
</tr>
<tr>
<td>abz7</td>
<td>1'131</td>
<td>1'334</td>
</tr>
<tr>
<td>la24</td>
<td>1'487</td>
<td>1'842</td>
</tr>
<tr>
<td>swv15</td>
<td>5'935</td>
<td>6'600</td>
</tr>
<tr>
<td>yn4</td>
<td>1'754</td>
<td>2'036</td>
</tr>
</tbody>
</table>
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).

<table>
<thead>
<tr>
<th>(\mathcal{I})</th>
<th>makespan</th>
<th>last improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>best</td>
<td>mean</td>
</tr>
<tr>
<td>abz7</td>
<td>1'131</td>
<td>1'334</td>
</tr>
<tr>
<td>la24</td>
<td>1'487</td>
<td>1'842</td>
</tr>
<tr>
<td>swv15</td>
<td>5'935</td>
<td>6’600</td>
</tr>
<tr>
<td>yn4</td>
<td>1’754</td>
<td>2’036</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>(I)</th>
<th>makespan</th>
<th>last improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>best</td>
<td>mean</td>
</tr>
<tr>
<td>abz7</td>
<td>1'131</td>
<td>1'334</td>
</tr>
<tr>
<td>la24</td>
<td>1'487</td>
<td>1'842</td>
</tr>
<tr>
<td>swv15</td>
<td>5'935</td>
<td>6'600</td>
</tr>
<tr>
<td>yn4</td>
<td>1'754</td>
<td>2'036</td>
</tr>
</tbody>
</table>
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 \((\text{med}(t) \approx 0)\) 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 ($\text{med}(t) \approx 0$) 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 ($\text{med}(t) \approx 0$) 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 ($\text{med}(t) \approx 0$) 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

```java
package aitoa.algorithms;

public class RandomSampling<X, Y> {
    // unnecessary stuff (e.g., constructor) omitted here...
```
```
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) {
        //
        //
        //
        //
        //
        //
        //
        //
        //
        //
    }
}
```
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
text
package aitoa.algorithms;

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());
    }
}
```

```
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

<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>1rs</td>
<td>1131</td>
<td>1334</td>
</tr>
<tr>
<td></td>
<td>rs</td>
<td>895</td>
<td>947</td>
</tr>
<tr>
<td>la24</td>
<td>1rs</td>
<td>1487</td>
<td>1842</td>
</tr>
<tr>
<td></td>
<td>rs</td>
<td>1153</td>
<td>1206</td>
</tr>
<tr>
<td>swv15</td>
<td>1rs</td>
<td>5935</td>
<td>6600</td>
</tr>
<tr>
<td></td>
<td>rs</td>
<td>4988</td>
<td>5166</td>
</tr>
<tr>
<td>yn4</td>
<td>1rs</td>
<td>1754</td>
<td>2036</td>
</tr>
<tr>
<td></td>
<td>rs</td>
<td>1460</td>
<td>1498</td>
</tr>
</tbody>
</table>
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?

1rs: median result of single random sample algorithm
So what do we get?

rs: median result of 3 min of random sampling algorithm

0 500 1000 1500
0
1
2
3
4
5
6
7
8
9
11 8 5 0 1 14 2 3 4 6 12 7 13
3 2 12 4 6 1 9 5 0 10 8
4 6 13 12 11 0 7 2... 10 9 2 5 11 8 7
4 10 3 7 13 12 8 6 5 9 2
7 1 4 5 6 14 13 12 9 0 11 2 8
12 9 0 14 8 6 5 1 7 11 4 10 2 13

la24 / 1208
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?

1rs: median result of single random sample algorithm
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

time in ms

abz7
rs
Progress over Time
Progress over Time

time in ms

yn4
rs

time in ms
Progress over Time

• Law of Diminishing Returns$^6$
Progress over Time

- Law of Diminishing Returns\textsuperscript{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

normal plot

time in ms

abz7

rs
Progress over Time

• Law of Diminishing Returns: 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\textsuperscript{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**: 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.
We started with an algorithm that created a single random solution. Let's call this algorithm $A$. 
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 $A$.

• We then wrapped a loop around $A$, we restarted $A$ 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 $\mathcal{B} = \text{a restarted algorithm } \mathcal{A}$”, 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 $A$.
- We then wrapped a loop around $A$, we restarted $A$ again and again until the time was up (and of course, remembered the best solution).
- This is actually basic strategy of “algorithm $B = a$ restarted algorithm $A$”, 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 $\mathcal{A}$.
Thank you
References


