Metaheuristics for Smart Manufacturing

3. Random Sampling

Thomas Weise · 汤卫思

tweise@hfuu.edu.cn · http://iao.hfuu.edu.cn

Hefei University, South Campus 2	合肥学院 南艳湖校区/南2区
Faculty of Computer Science and Technology	计算机科学与技术系
Institute of Applied Optimization	应用优化研究所
230601 Shushan District, Hefei, Anhui, China	中国 安徽省 合肥市 蜀山区 230601
Econ. & Tech. Devel. Zone, Jinxiu Dadao 99	经济技术开发区 锦绣大道99号

Outline

1 Introduction

2 Algorithm Concept

3 Experiment and Analysis

4 Summary

The slides and source code examples are available at
`http://iao.hfuu.edu.cn/teaching/lectures/metaheuristics-for-smart-manufacturing`

Introduction

- OK, so we can now represent a Gantt chart for m machines and n jobs as an integer string of length $m \times n$.

Introduction

- OK, so we can now 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?

- OK, so we can now 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!

Create a Single Random Point g in the Search Space $\mathbb{G}$

Listing: Creation of a single random $g \in \mathbb{G}$.

```
public class Representation {
  public int[] createRandom(final Random random) {
    final int jobs = this.m_jobTime.length;
    final int machines = this.m_machineTime.length;
    final int[] res = new int[jobs * machines];
    int index = -1;

    // first create a feasible string which contains each job index m times
    for (int i = 0; i < jobs; i++) {
      for (int j = 0; j < machines; j++) {
        res[++index] = i;
      }
    }
    // then we randomly shuffle the string: the values don't change, just
    // their positions,
    // i.e. we get a representation for a random schedule
    Tools.shuffle(random, res, 0, res.length);
    return (res);
  }
}
```

Single Random Sample Algorithm

Listing: An algorithm which only creates one random sample.

```java
public final class SingleSample {

  /**
   * Generate a single random schedule.
   *
   * @param problem
   *          the problem instance, i.e., the jobs and machines
   * @param representation
   *          the methods provided by our representation
   * @param log
   *          the log: simple class to collect/print results
   * @return the Gantt chart data found
   */
  public static final int[][] run(final Problem problem,
      final Representation representation, final Log log) {

    final Random random = new Random();
    final int[][] solution = problem.newEmptySolution();

    // create a random point in the search space
    int[] g = representation.createRandom(random);

    // map the point to a solution
    representation.map(g, solution);

    // compute quality: actually calls problem.evaluate and remembers best result
    log.evaluate(solution);

    return log.getBestSolution(); // return best logged solution
  }
}
```

So what do we get?

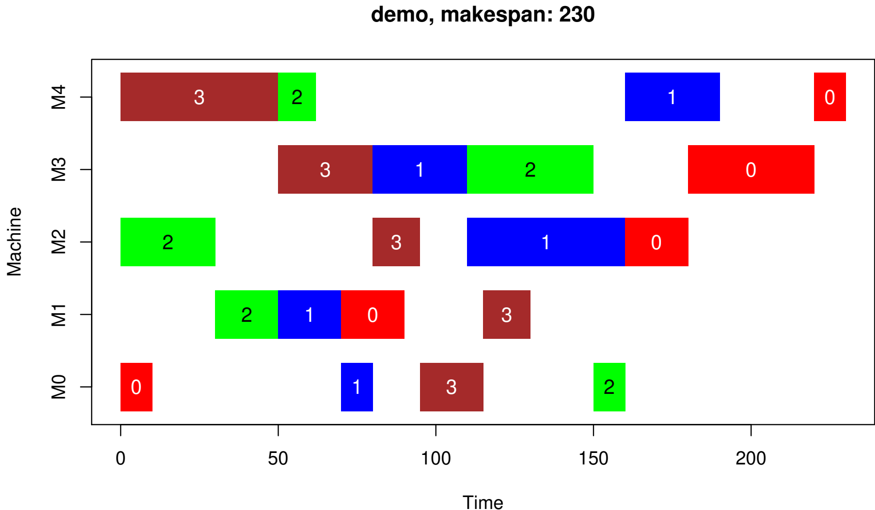
- I execute the program 100 times for each of the datasets `demo`, `la24`, `swv15`, and `yn4`

So what do we get?

- I execute the program 100 times for each of the datasets `demo`, `la24`, `swv15`, and `yn4`

	makespan		
instance	**mean**	**best**	**stddev**
`demo`	249.04	180	36.067
`la24`	1853.02	1498	175.15
`swv15`	6516.36	5593	295.67
`yn4`	2053.39	1762	134.09

So what do we get?



So what do we get?

la24, makespan: 1742

Machine: M0–M9

Time: 0, 500, 1000, 1500

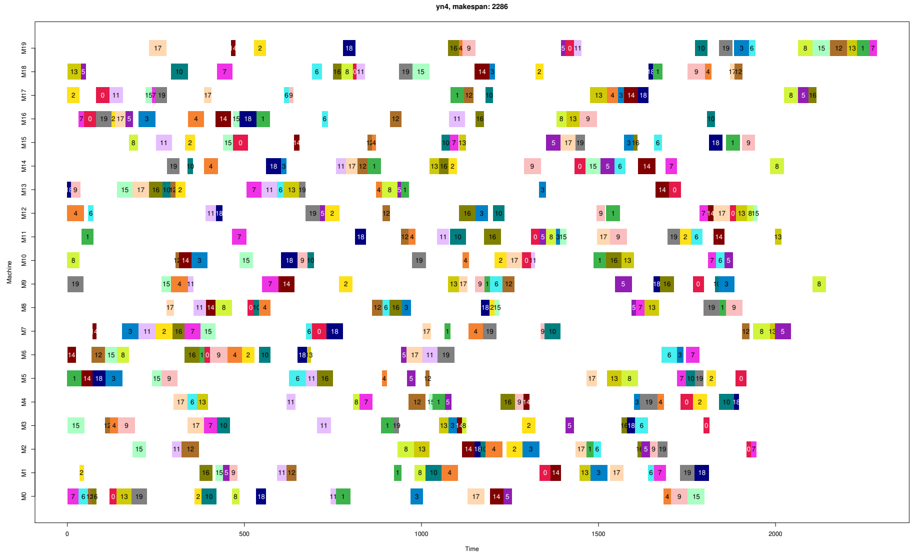
So what do we get?

swv15, makespan: 6581

Machine: M0, M1, M2, M3, M4, M5, M6, M7, M8, M9

Time: 0, 1000, 2000, 3000, 4000, 5000, 6000

So what do we get?



So what do we get?

- I execute the program 100 times for each of the datasets `demo`, `la24`, `swv15`, and `yn4`
- So the results are not good, there is lots of white space $\equiv$ wasted time.

	makespan		
instance	**mean**	**best**	**stddev**
`demo`	249.04	180	36.067
`la24`	1853.02	1498	175.15
`swv15`	6516.36	5593	295.67
`yn4`	2053.39	1762	134.09

- I execute the program 100 times for each of the datasets `demo`, `la24`, `swv15`, and `yn4`
- So the results are not good, there is lots of white space $\equiv$ wasted time. That was expected.

	makespan		
instance	**mean**	**best**	**stddev**
`demo`	249.04	180	36.067
`la24`	1853.02	1498	175.15
`swv15`	6516.36	5593	295.67
`yn4`	2053.39	1762	134.09

So what do we get?

- I execute the program 100 times for each of the datasets `demo`, `la24`, `swv15`, and `yn4`
- So the results are not good, there is lots of white space $\equiv$ wasted time. That was expected.
- Notice 1. We can create and test the schedules very very fast.

	makespan		
instance	**mean**	**best**	**stddev**
`demo`	249.04	180	36.067
`la24`	1853.02	1498	175.15
`swv15`	6516.36	5593	295.67
`yn4`	2053.39	1762	134.09

So what do we get?

- I execute the program 100 times for each of the datasets `demo`, `la24`, `swv15`, and `yn4`
- So the results are not good, there is lots of white space $\equiv$ wasted time. That was expected.
- Notice 1. We can create and test the schedules very very fast.
- Notice 2. There is a standard deviation in the results due to randomness.

	makespan		
instance	**mean**	**best**	**stddev**
`demo`	249.04	180	36.067
`la24`	1853.02	1498	175.15
`swv15`	6516.36	5593	295.67
`yn4`	2053.39	1762	134.09

Exploit Randomness: Random Sampling

- If we can generate solutions fast and sometimes are lucky, sometimes not. . .

Exploit Randomness: Random Sampling

- If we can generate solutions fast and sometimes are lucky, sometimes not...
- then why don't we generate 1'000'000 schedules and keep the best one?

Random Sampling Algorithm

Listing: An algorithm which creates many random samples.

```java
public class RandomSampling {

  /**
   * Solve a problem by generating many multiple random samples
   *
   * @param problem
   *          the problem
   * @param representation
   *          the representation
   * @param log
   *          the log
   * @return the result
   */
  public static int[][] run(final Problem problem,
      final Representation representation, final Log log) {

    final Random random = new Random();
    final int[][] xbest = problem.newEmptySolution();

    while (!(log.shouldTerminate())) { // this becomes true after 1'000'000 steps
      // create a random point in the search space
      int[] g = representation.createRandom(random);

      // map the point to a solution
      representation.map(g, xbest);

      // compute quality (and remember best in an internal variable)
      log.evaluate(xbest);
    }

    return log.getBestSolution(); // return best logged solution
  }
}
```

So what do we get?

- I execute the program 100 times for each of the datasets `demo`, `la24`, `swv15`, and `yn4`

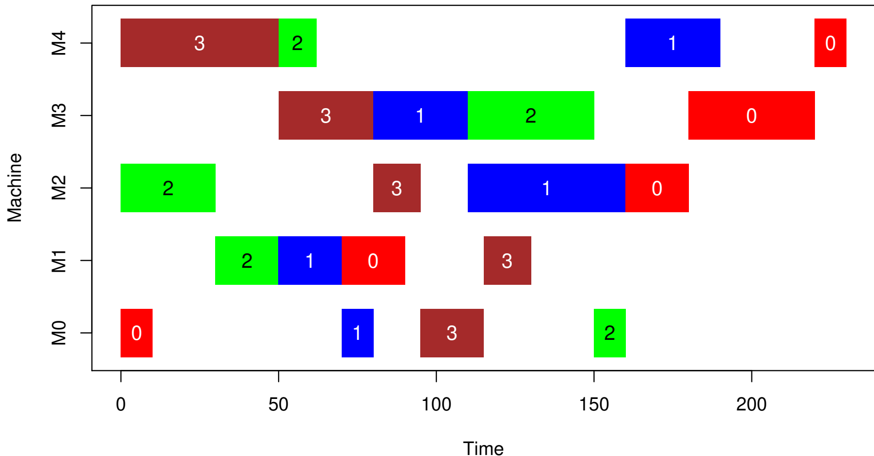
So what do we get?

- I execute the program 100 times for each of the datasets `demo`, `la24`, `swv15`, and `yn4`

		makespan		
algo	**inst**	**mean**	**best**	**stddev**
Random Sampling	demo	180	180	0
Single Sample	demo	249.04	180	36.07
Random Sampling	la24	1254.03	1205	18.16
Single Sample	la24	1853.02	1498	175.15
Random Sampling	swv15	5255.16	5038	52.06
Single Sample	swv15	6516.36	5593	295.67
Random Sampling	yn4	1532.79	1469	17.56
Single Sample	yn4	2053.39	1762	134.09

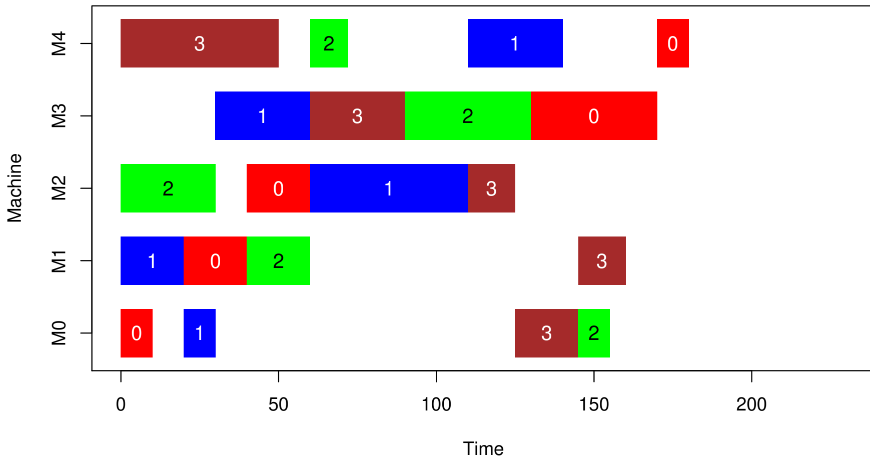
So what do we get?

single random sample



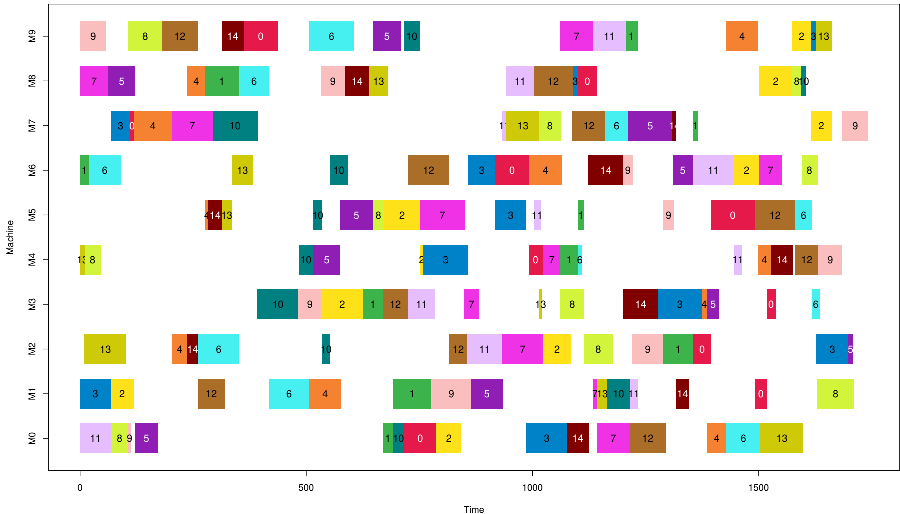
So what do we get?

1'000'000 random samples



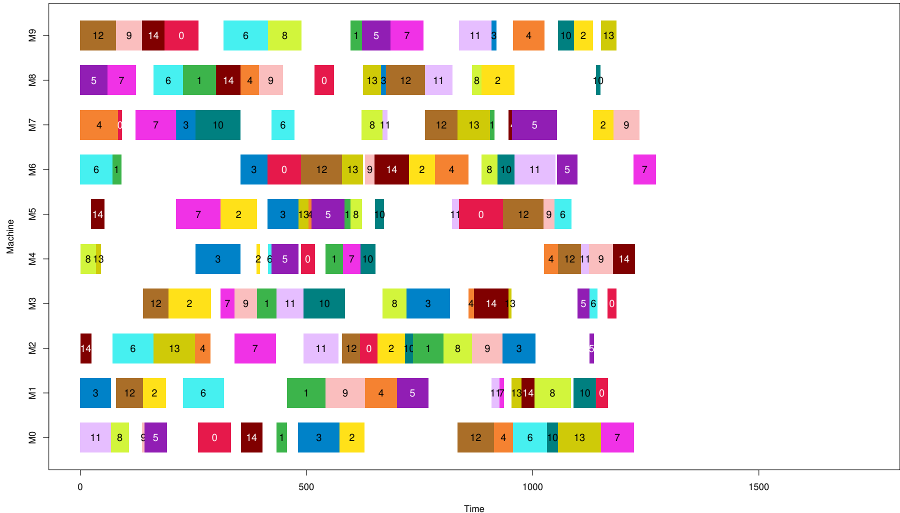
So what do we get?

single random sample



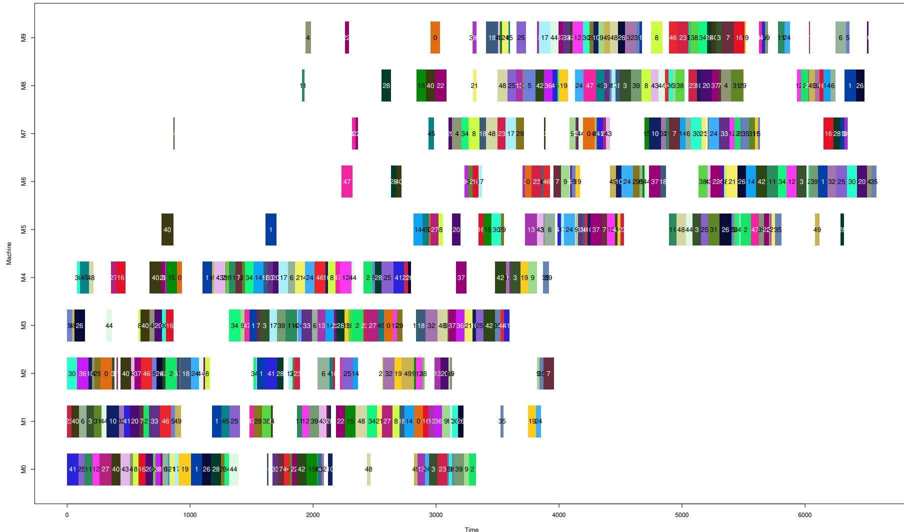
So what do we get?

1'000'000 random samples



So what do we get?

single random sample



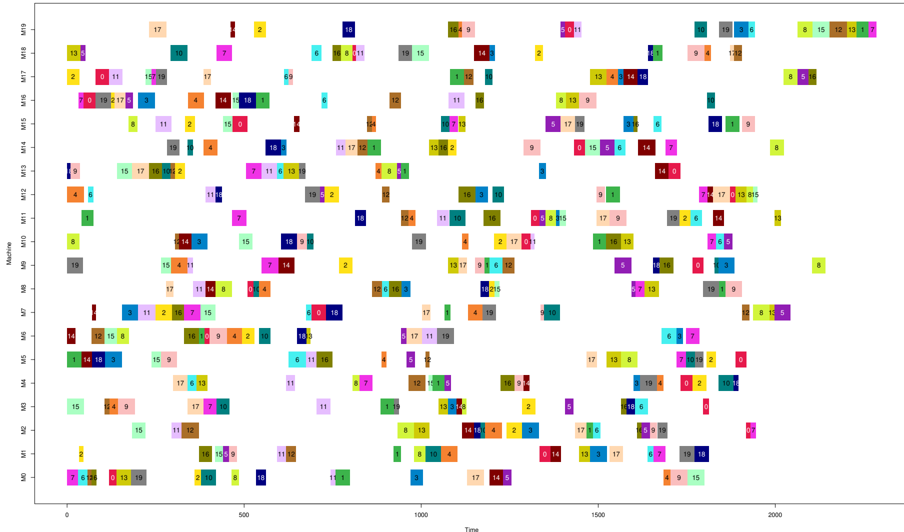
1’000’000 random samples

swv15, makespan: 5217

Machine: M0, M1, M2, M3, M4, M5, M6, M7, M8, M9

Time: 0, 1000, 2000, 3000, 4000, 5000, 6000

So what do we get?



So what do we get?

1’000’000 random samples

yn4, makespan: 1534

Machine: M0–M19

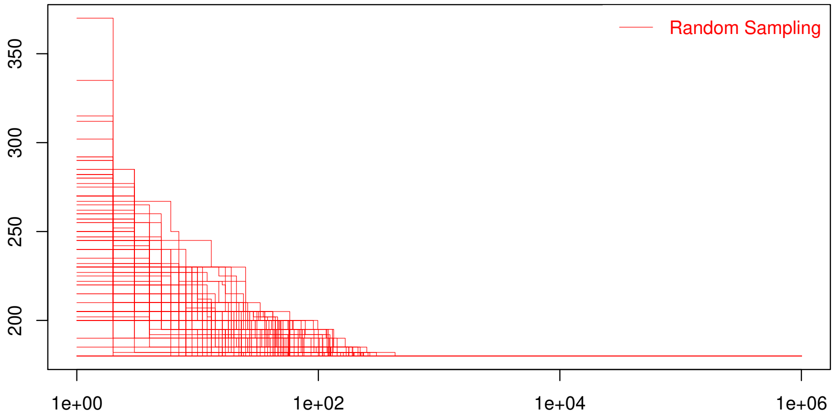
Time: 0, 500, 1000, 1500, 2000

Progress over Time

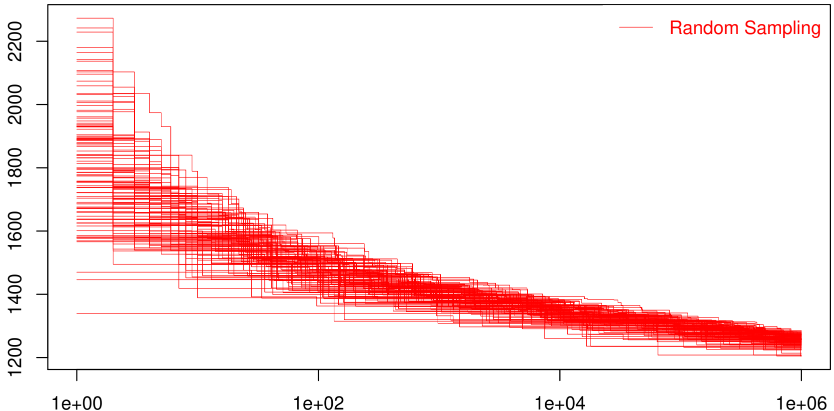
What progress does the algorithm make over time (measured in objective function evaluations, FEs)?

Progress over Time

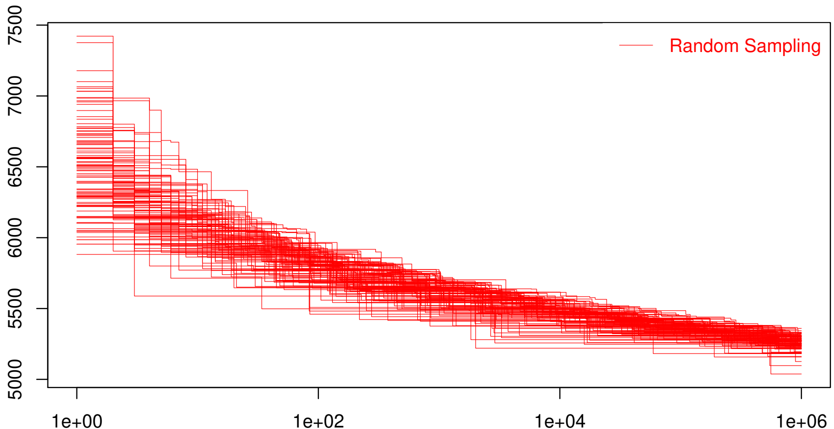
What progress does the algorithm make over time (measured in objective function evaluations, FEs)?



What progress does the algorithm make over time (measured in objective function evaluations, FEs)?

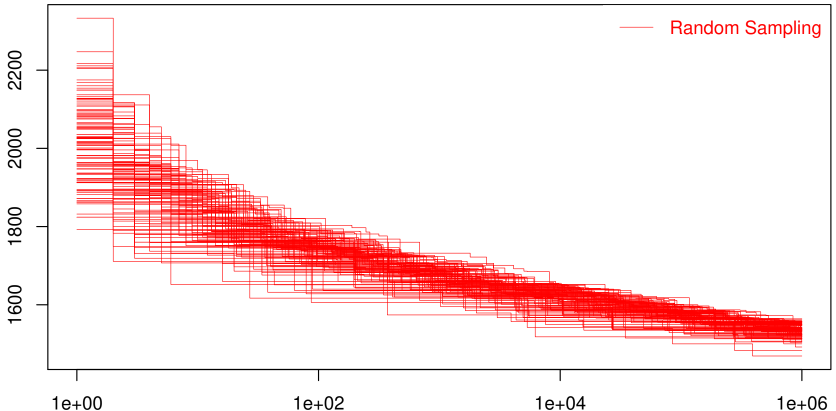


What progress does the algorithm make over time (measured in objective function evaluations, FEs)?



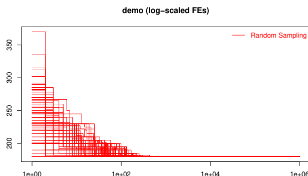
Progress over Time

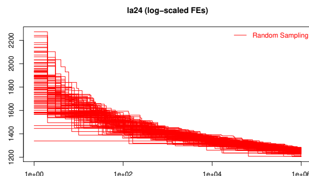
What progress does the algorithm make over time (measured in objective function evaluations, FEs)?

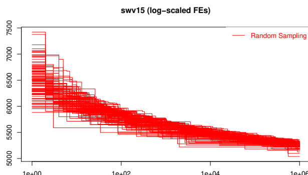


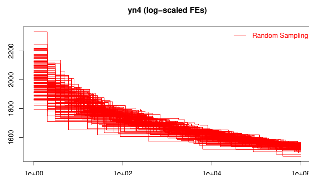
Progress over Time

- Law of Diminishing Returns [1]: 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.



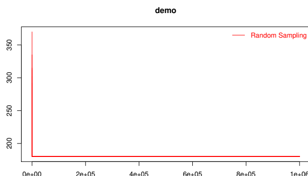


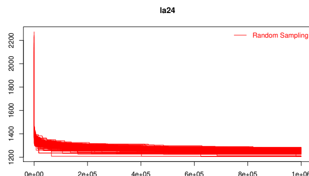


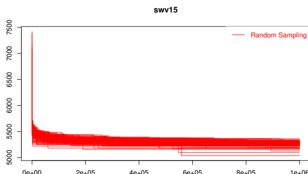


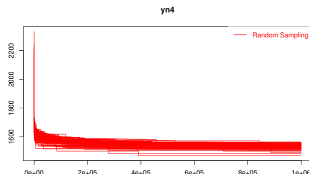
Progress over Time

- Law of Diminishing Returns [1]: 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.



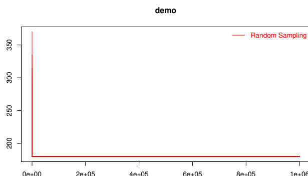


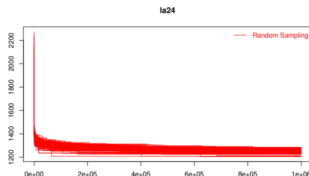


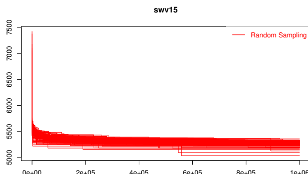


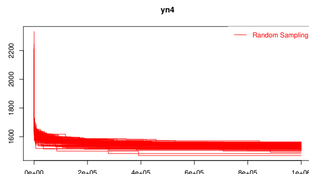
Progress over Time

- Law of Diminishing Returns [1]: 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

- We have now a basic algorithm that provide some solutions.

Summary

- We have now a basic algorithm that provide some solutions.
- But it is stupid and does not make any use of the information it has seen during the search.

谢谢

Thank you

Thomas Weise [汤卫思]
tweise@hfuu.edu.cn
http://iao.hfuu.edu.cn

Hefei University, South Campus 2
Institute of Applied Optimization
Shushan District, Hefei, Anhui,
China

Caspar David Friedrich, “Der Wanderer über dem Nebelmeer”, 1818
http://en.wikipedia.org/wiki/Wanderer_above_the_Sea_of_Fog

Bibliography

Bibliography I

1. Paul Anthony Samuelson and William Dawbney Nordhaus. *Microeconomics*. Boston, MA, USA: McGraw-Hill Irwin, 17th edition, 2001. ISBN 0071-180664.