Institute of Formal Methods in Computer Science

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 334

Experimental Analysis of Randomized Calculations of Average Rankings

Tim Zeiß

Course of Study:	Informatik
Examiner:	Prof. Dr. Stefan Funke
Supervisor:	Dipl.-Inf. Martin Seybold
Commenced:	May 12, 2016
Completed:	November 11, 2016
CR-Classification:	G.1.2, G.2.1, G.3, G.4

Abstract

Listing a set of points, such that a point gets a higher rank, if none of its coordinates is smaller, creates a partial order. It is possible to get a ranking without randomly favoring certain points, by averaging all valid rankings. However, this brute force algorithm is too slow for more than ten points. To handle more points, we will give a randomized, approximative approach to solve this problem and analyze the convergence rates of different strategies.

Kurzfassung

Beim Auflisten einer Menge von Punkten, sodass ein Punkt höher gerankt wird, falls keine seiner Koordinaten kleiner ist, entsteht eine partielle Ordnung. Es ist möglich, ein Ranking zu bekommen, ohne willkürlich Punkte zu bevorzugen, indem man über alle gültigen Rankings mittelt. Jedoch ist dieser Brute Force Algorithmus zu langsam für mehr als zehn Punkte. Um mehr Punkte bearbeiten zu können stellen wir einen randomisierten, approximativen Ansatz vor, um dieses Problem zu lösen und analysieren die Konvergenzraten verschiedener Strategien.

Contents

1 Introduction 11

2 Related Work 13

3 Approach 15
3.1 Partial Order 15
3.2 Uniform Sampling 16
3.3 Starting Ranking 16
3.4 Averaging 16
3.5 Other Sampling Strategies 17
3.6 Brute Force 18

4 Appraisal 19

5 Conclusion 29

6 Future Work 31

A R-Package Documentation 33

Bibliography 45

List of Figures

4.1 Total Order 20
4.2 Small Variance PO 21
4.3 Medium Variance PO 22
4.4 Fix Random PO 24
4.5 Average of 100 Random POs 25
4.6 Exact Deviation 26

List of Abbreviations

MCMC Markov chain Monte Carlo. 13

PO partial order. 11

1 Introduction

Listing n points in $\mathbb{R}^d$, where a point gets a higher rank, if none of its coordinates is smaller than those of another one, creates a partial order (PO). This way, many rankings can be found.
Averaging all possible rankings gives a ranking that doesn't randomly favor certain points. An exact average ranking can be computed by checking for all possible permutations, if they are conform with the given PO. For a PO with n elements, there are $n!$ permutations, so this brute force algorithm runs in time $\Theta(n!)$. This leads to a very fast increase in run time and make the algorithm unsuitable for POs with more than 10-15 elements. To handle more elements, we take a randomized, approximative approach.
Existing methods give a uniform sample ranking, by swapping two randomly chosen neighbors $\Theta(n^3 \log n)$ times. Averaging many of these rankings will result in a ranking, that has a high probability to be close to the exact average ranking.
We will give an implementation of this method as a POSIX conform C99 library, that is accessible by an R-package. Using this, we will analyze the convergence rate of this method. Additionally, we will try to find faster sampling strategies, compare them to our original method and adapt our implementation accordingly.

2 Related Work

Karzanov and Khachiyan [KK91] developed a Markov chain Monte Carlo (MCMC) algorithm that shuffles a linear extension of a PO to generate an approximately uniform sample of all its linear extensions. This method uses a Markov chain, known as the Karzanov–Khachiyan Markov chain, that swaps a randomly chosen element and its successor with probability $\frac{1}{2}$, if the resulting permutation is still a valid linear extension of the PO. They showed that after $8n^5 \log \frac{|\Omega|}{\epsilon}) \leq O(n^6 \log n)$ transitions, where $|\Omega|$ is the number of linear extensions, the Karzanov–Khachiyan Markov chain converges to its equilibrium distribution and the resulting linear extension is approximately uniform.

Bubley and Dyer [BD99] gave a tighter bound for a variation of the Karzanov–Khachiyan Markov chain: Instead of a uniform distribution, they chose a parabolic distribution to select the element that is to be swapped. They used path coupling to give an upper bound of $O(n^3 \log n)$ for their variation and $O(n^4 \log^2 n)$ for the original Karzanov–Khachiyan Markov chain.

Wilson [Wil04] generalized Bubley and Dyer's path coupling method by adding weights and showed that the Karzanov–Khachiyan Markov chain also mixes in $O(n^3 \log n)$. Though Wilson's upper bound of $(\frac{4}{\pi^2} + o(1))n^3 \log n$ for the Karzanov–Khachiyan Markov chain is about 22% higher than Bubley and Dyer's bound of $(\frac{1}{3} + o(1))n^3 \log n$ for their variation, the Karzanov–Khachiyan Markov chain is still a better choice for practical use, since uniform sampling can be done much more efficiently than sampling from a parabolic distribution.

3 Approach

In this chapter we will take a closer look on the methods and algorithms that we'll use to compute average rankings.
We'll take advantage on previous research about uniform sampling of linear extensions to compute a certain number of random rankings, depending on the desired accuracy and the size and variance of the PO. Then we'll use these rankings to compute their average and give the option to plot the result in a Hasse Diagram.
For the appraisal of this method we will also give two other strategies to compute random rankings. While the first one is a non-uniform variant of the original algorithm, the second one picks a random leaf from the topological graph of the PO.

3.1 Partial Order

The first thing we need to do is to extract the corresponding PO from a set S of n points P_i

$$S = \{P_1, P_2, ..., P_n\}$$

with $P_i \in \mathbb{R}^d$

$$P_i = (x_{i,1}, x_{i,2}, ..., x_{i,d})$$

A point P_a shall be ranked higher than point P_b if none of its coordinates is smaller than the corresponding coordinate of P_b.

$$P_a > P_b \iff \forall x_{a,i} : x_{a,i} \geq x_{b,i}$$

Now we can create a $n \times n$ matrix M that represents the PO with

$$M_{i,j} = \left\{ \begin{array}{ll} 1, & \text{if } P_i > P_j \\ 0, & \text{else} \end{array} \right\}$$

This matrix can be used for a fast access to the PO.

3.2 Uniform Sampling

To get a random ranking we will use an algorithm based on a method developed by Karzanov and Khachiyan [KK91].
Their approach was to give a Markov chain with an equilibrium distribution equal to the desired distribution. Running this Markov chain for several times on a starting ranking creates a MCMC algorithm, that returns a random ranking by shuffling the starting ranking. In each step of this Markov chain, known as the Karzanov–Khachiyan Markov chain, a random element of the ranking is selected and and swapped with its successor with probability $\frac{1}{2}$, if the resulting ranking is still conform with the PO.
Though the result can never be completely independent from the starting ranking, Wilson [Wil04] showed that $\Theta(n^3 \log n)$ steps are enough to get an approximately uniform sample of all valid linear extensions.

3.3 Starting Ranking

For the computation of a random ranking, our algorithm needs a starting ranking as input. We'll take a deterministic approach to this, since the impact of the starting ranking is negligible, if we take enough steps of the Markov chain. To reduce this effect even more, we will use this deterministic ranking only for the first random ranking, each following computation will take the result of the previous one as input.
For the first starting ranking, we will use the topological graph of the PO and pick in each step the leaf with the smallest index and remove it from the graph.

3.4 Averaging

Once we have the required number of random rankings, the averaging process is rather simple: For each element we add the ranks given by the random rankings and divide the result by the number of rankings. The difficult part is here to decide how many random rankings we need to get close enough to the exact average ranking. As default values for the accuracy we will choose for each element the standard deviation $s = 0.5$ and the probability $p = 90\%$ to keep the error $er < s$, because this will give us a good trade-off between run time and accuracy. Nevertheless these parameters can be customizes if the situation requires a higher focus on accuracy or run time, at the cost of the other. Since we don't know the actual distribution for each element of the PO, we can only

approximate the required number of rankings. An estimation for this can be found with Chebyshev's inequality:

$$Pr[|X - \mu| < k] \geq 1 - \frac{\sigma^2}{k^2}$$

Which leads for our case to

$$Pr[er < s] \geq 1 - \frac{\sigma_n^2}{s^2}$$

$$p \geq 1 - \frac{\sigma_n^2}{s^2}$$

$$\sigma_n^2 \geq (1-p)s^2$$

where σ_n^2 is the variance of the rank after taking the average of n rankings. For the computation of σ_n^2 we need the basic variance σ^2

$$\sigma_n^2 = \frac{\sigma^2}{n}$$

Without the distribution we also don't know the exact variance, but we can easily get the range of possible ranks for each elements from the PO. By assuming uniform distribution within this range we can get an approximation of the exact variance:

$$\sigma^2 = \frac{(b-a+2)(b-a)}{12}$$

with minimum rank a and maximum rank b. It can easily be seen that this will give us an upper bound of the variance: If element e is incomparable to any other element that may appear within its range, the distribution will be uniform. For each element in range that is greater or less than e, it will be less likely for e to get close to the maximum or minimum, reducing its variance.

$$n = \frac{(b-a+2)(b-a)}{12(1-p)s^2}$$

3.5 Other Sampling Strategies

3.5.1 Non-Uniform Variation

To Reduce the computation time of each random ranking, we simply reduce the steps of our Markov chain. This will make each individual ranking non-uniform, because it depends on a higher rate on the starting ranking. But since we have a high amount of rankings and take the each resulting ranking as starting ranking for the next one, we expect this to converge eventually, tough we might need more rankings to get a decent average.

3.5.2 Picking form the Topological Sorting

Another strategy we try is to pick random leafs from the topological sorting of the PO, similar to our deterministic algorithm for the starting ranking. For this, we will create a topological graph to represent the PO and determine for each element if they have an edge to any other element. If not, they are a possible candidate to be picked as next element in the ranking. We will use two different methods to pick the next element: For the first one, it will be selected uniformly from all leafs. The second method will use a weighted approach, where each possible candidate has a basic weight of 1, plus 1 additional weight for each incoming edge.

3.6 Brute Force

For small POs with up to 10 elements we will use a brute force algorithm to compute an exact average ranking. In spite of its high computation time of $\Theta(n!)$ it is still fast enough to solve small POs almost instantly.
Our brute force algorithm checks for each possible permutation, if they are a valid linear extension of the PO. For each element, we add their positions in all valid linear extensions and compute its average.

4 Appraisal

In this chapter we will evaluate the accuracy of each method and compare their convergence rate. We will take a look at different test cases and give a plot of the convergence rate regarding the computation time, the number of random rankings and the amount of random numbers.
The exact ranking will usually be unknown, so we can't use the error of the result for the convergence rate. Therefore, we will use the average difference between the rank of the same element in two rankings as a measure for the convergence rate. For better comparison of each method, we won't take the difference after each additional ranking, but compare the current average rankings after a certain time step. The reason we do this is that we have a huge difference of computation time for random rankings between different methods. Since the impact of a single ranking is highly influenced by the number of already created rankings, methods with a low computation time of random rankings would appear to converge much faster.
All test cases have been calculated on an Intel Core i5-4690 processor and may vary, depending on the CPU.

We will compare the following methods:

- The original Karzanov–Khachiyan Markov chain, that gives uniform rankings by swapping $\Theta(n^3 \log n)$ elements
  - [nˆ3log(n)]
- Three non-uniform variations of the Karzanov–Khachiyan Markov chain with $\log n$, n and n^2 swaps
  - [log(n)]
  - [n]
  - [nˆ2]
- Randomly picking leafs from the topological graph, with and without weights
  - [Top] (non-weighted)
  - [TopW] (weighted)

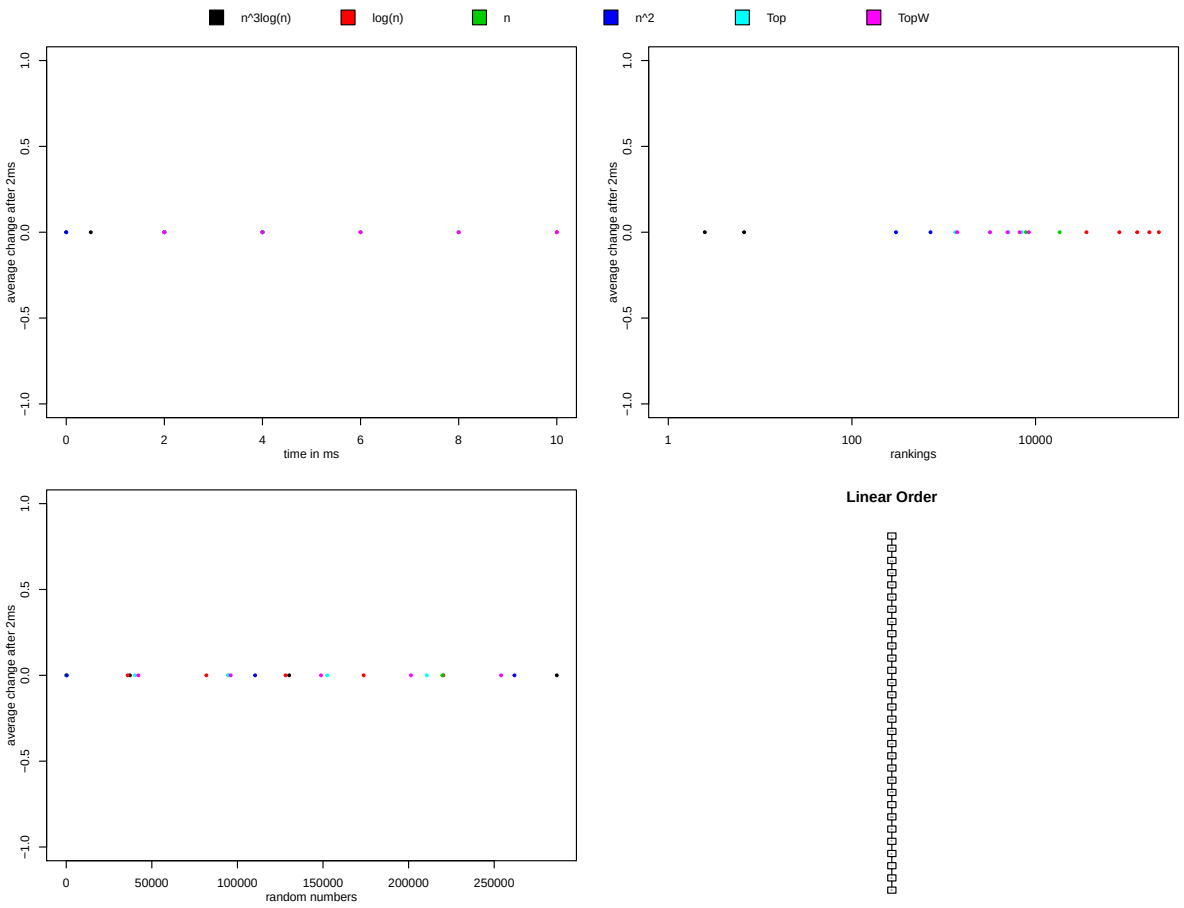

Figure 4.1: Total Order

The first case we will look at is a total order. As expected, all methods need only one step to give the exact average, since there is only one possible random ranking that each of them can produce. For the plotting of the convergence rate, the numbers of iterations have been artificially raised to 10, because we don't have any convergence for a single ranking. There it can easily be seen that each line is stuck to zero, because there is absolutely no change within their rankings.

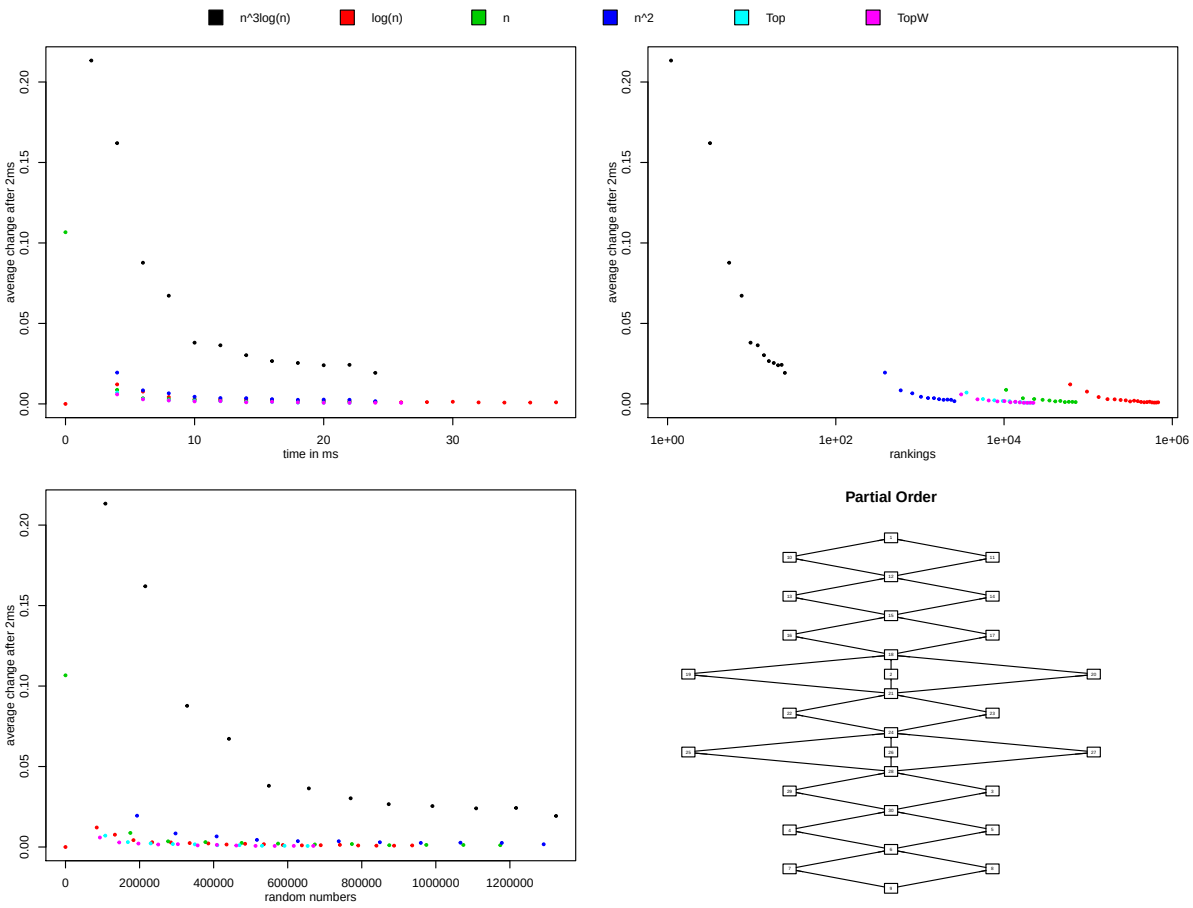

Figure 4.2: Small Variance PO

Our second test case is a PO with a rather small variance. Each method converges quite fast to zero. While our original Karzanov–Khachiyan Markov chain needs some time to converge, the other methods have already produced enough random rankings after the first measure point to get very close to zero. This suggests that our original algorithm is overall slower than the other approaches.

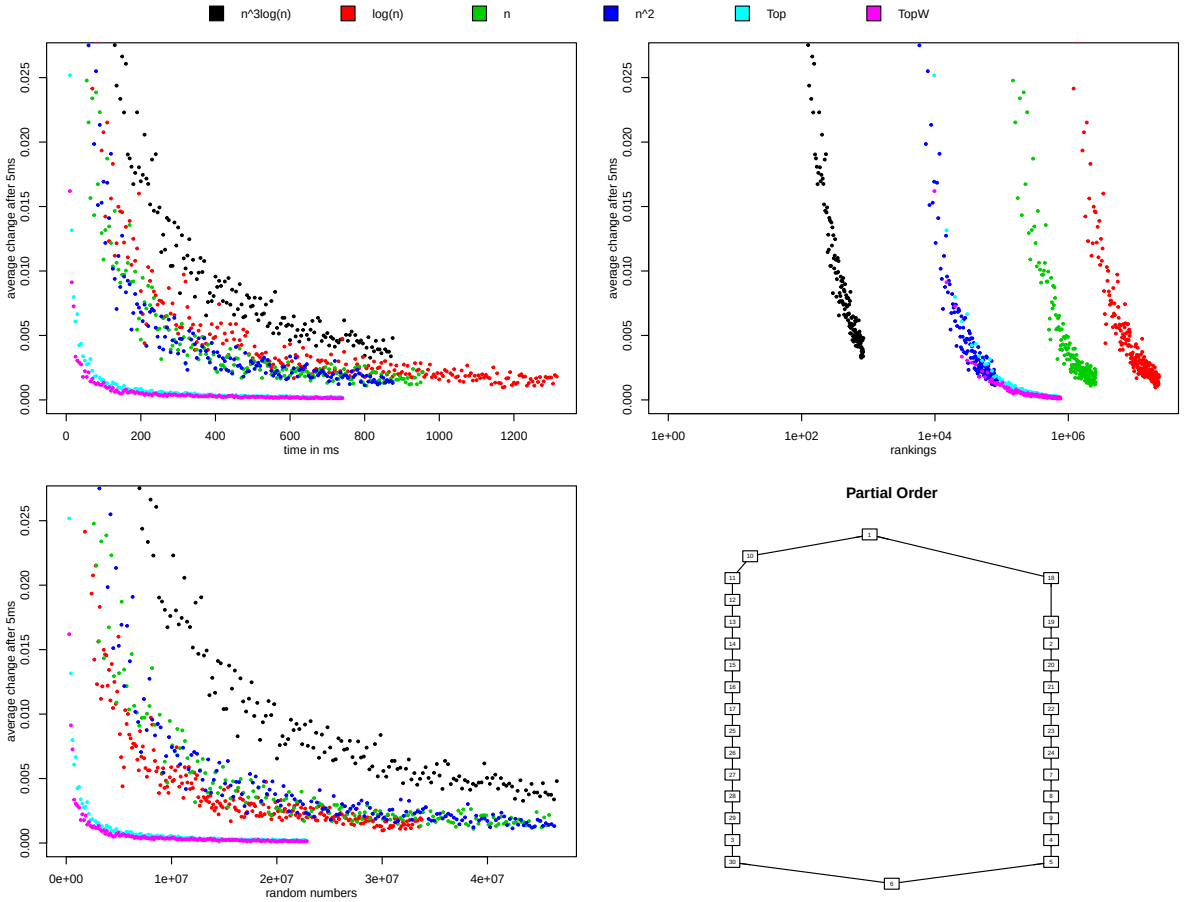

Figure 4.3: Medium Variance PO

Our next case seems like a rather simple PO, but nevertheless, it has a much higher variance and needs much more random rankings to get an accurate average than the previous cases. Therefore, this will give us a more distinctive view on the different methods. As before, the uniform Karzanov–Khachiyan Markov chain needs the least number of random rankings to give a good approximation, but because each uniform sample needs a lot of computation time and random numbers, this method is overall slower than the others and requires more random numbers.
The non-uniform Karzanov–Khachiyan Markov chains are all a bit faster and require less random numbers. While there isn't much variation between the different non-uniform Markov chains, the method with a logarithmic number of swaps needs fewer random than the others, while the one with a square number is a bit faster. Reducing the number of swaps decreases the quality of each single random ranking and requires to compute more of them, but each ranking needs less computation time and random numbers.
The topological approaches appear to require about the same number of random rankings as the Karzanov–Khachiyan Markov chain with a square number of swaps, but are able to

compute each ranking much faster and show the best convergence rates. The weighted variant converges slightly faster than the non-weighted approach.

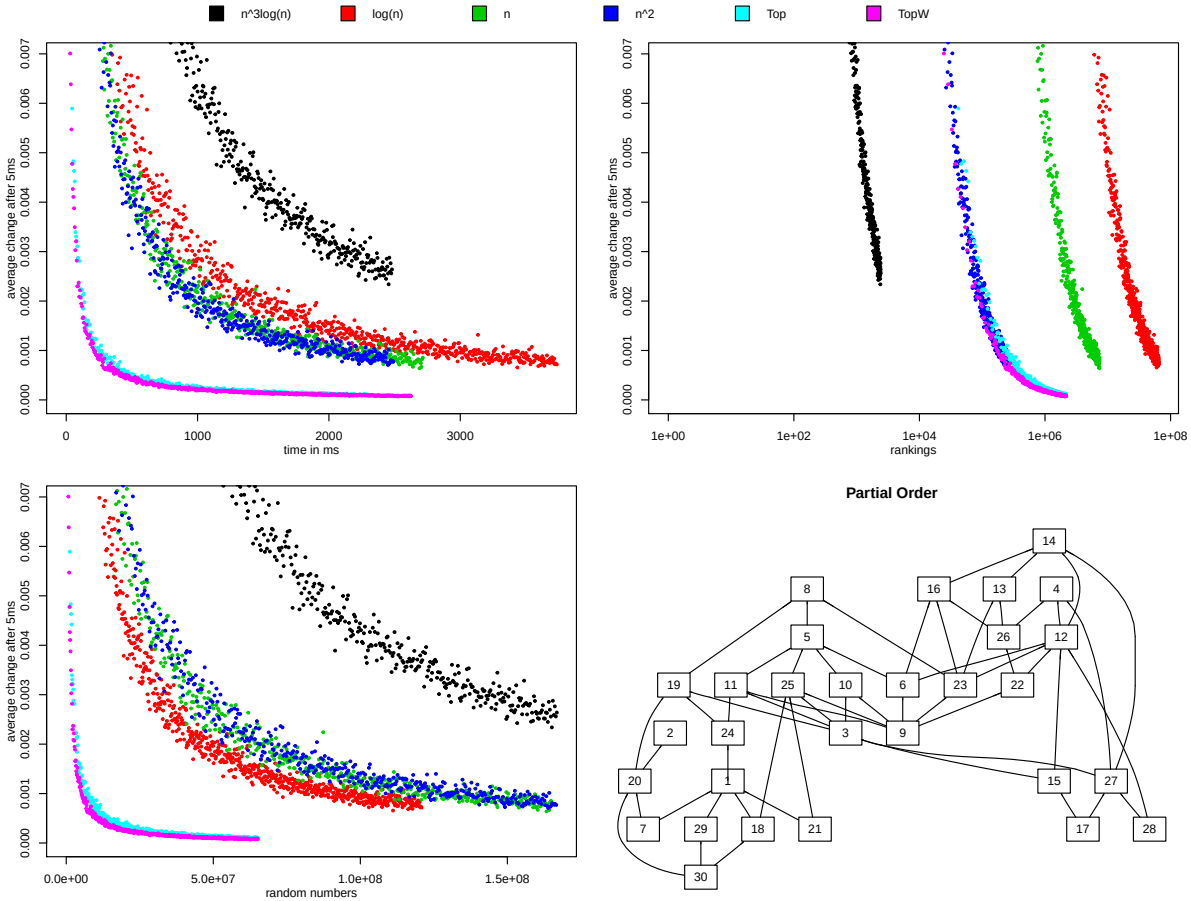

Figure 4.4: Fix Random PO

The previous cases were all artificially constructed POs. Now we will see if our algorithms will still converge, if we take a random PO instead.
The results look quite similar to the previous one. Each method needs more time and random numbers to converge and the differences between them got a bit wider, but the characteristic features of each methods are the same as before. This suggests that these will converge even for POs with high variance and we didn't just hit a convenient PO. But since one random PO isn't enough to give representative feedback, we will take a look at a higher number of different random POs in the next case.

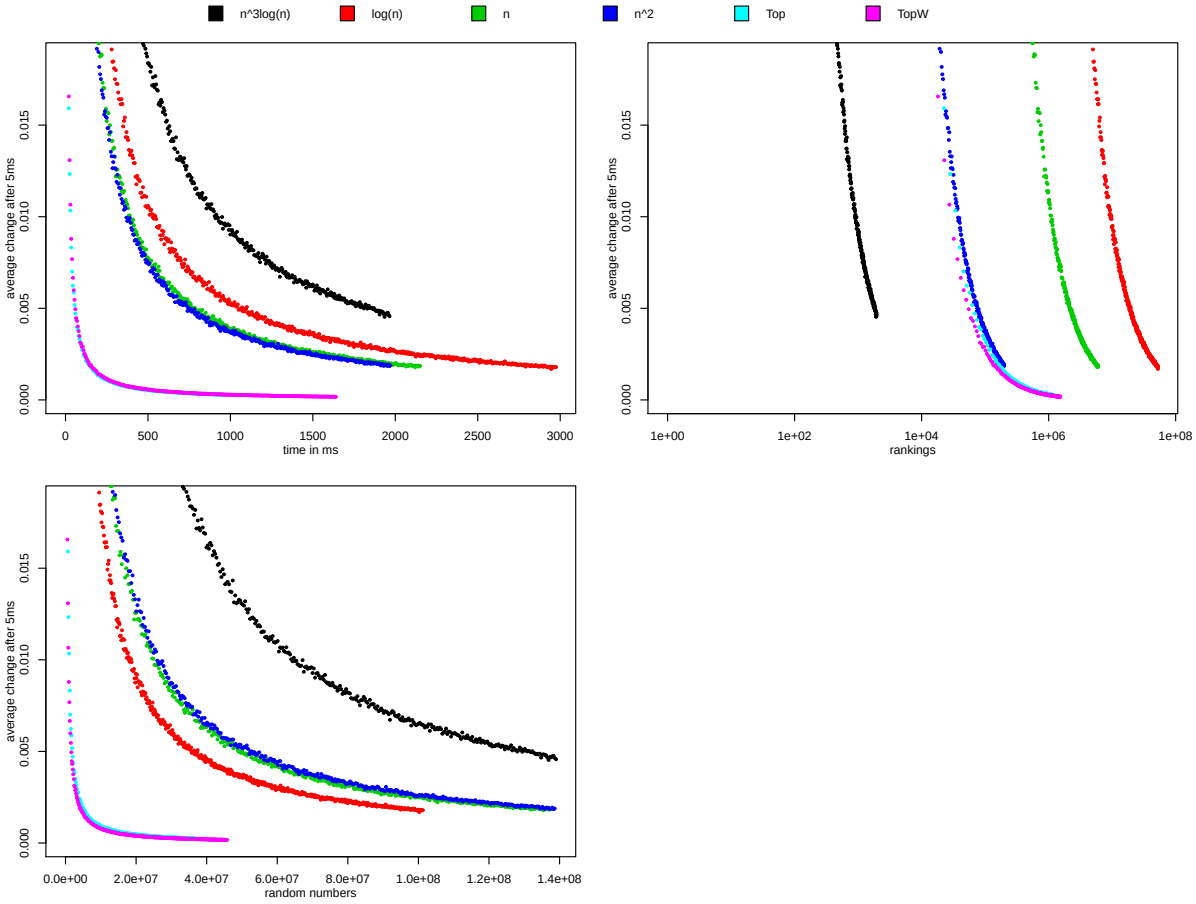

Figure 4.5: Average of 100 Random POs

We see here the average convergence rate of 100 different random POs, created from sets of 30 points with dimensions between 2 and 5 and average dimension of 3.46. On average, an element of the POs had a variance of 46.84. Again, we get a plot similar to the previous cases, which supports our assumptions. The topological approach is by far the fastest, the uniform Markov chain converges rather slow and the non-uniform variants lie in between.

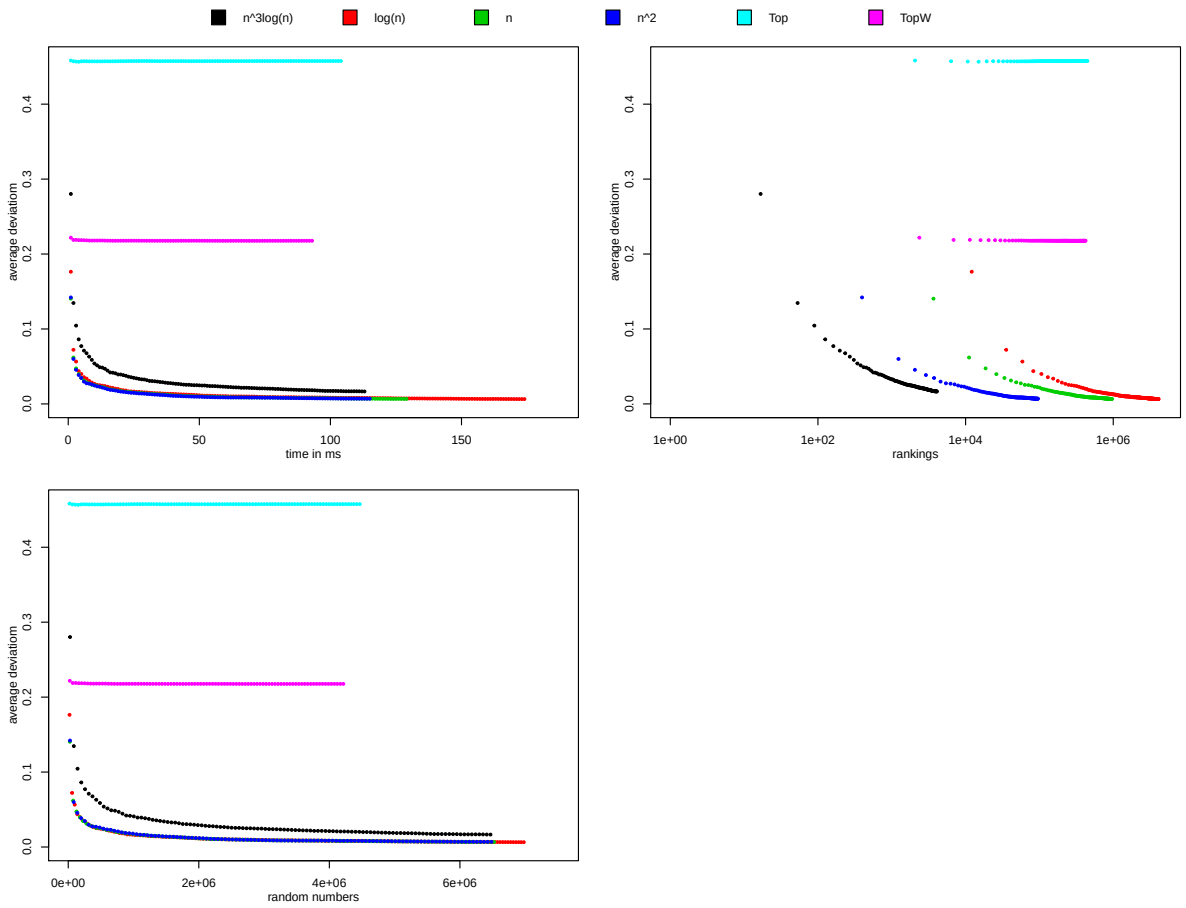

Figure 4.6: Exact Deviation

So far, we could see that each method converges, but we don't know if they actually converge to correct ranks that an exact method would give. Since Wilson [Wil04] proofed that the original Karzanov–Khachiyan Markov chain gives uniform samples after $\Theta(n^3 \log n)$ steps, it's obvious that averaging many of these rankings will converge to the correct average, but we don't know about the other strategies, yet. So this time we will limit the size of our POs to ten elements. These will be small enough for our brute force algorithm to compute an exact average. Now we can compare the average ranking in each step with the exact ranking and use its average deviation for the convergence rate. This allows us to check if each method converges to the exact average ranking.
The given plot shows the average convergence rate of 100 different POs of size 10, dimensions 2 and an average variance of 0.98. It can easily be seen that the methods based on the Karzanov–Khachiyan Markov chain each converges to zero, so they will eventually give an average ranking that is close to the exact one. The methods based on the topological graph, however, converge to a value different from zero. While the weighted approach give considerably better results than the non-weighted one, but

both of them are rather unreliable. The improved accuracy of the weighted approach suggests, that there may be weightings that give an even better accuracy, or that even hits the exact average for all possible POs, but finding such a weighting would exceed the bounds of this work.

Results

We have seen in our test cases, that the original Karzanov–Khachiyan Markov chain is slower and needs more time to converge than the other methods. Picking from the topological graph showed very fast convergence rates, but didn't converge to the exact average, making it unreliable. The non-uniform variants of the Karzanov–Khachiyan Markov chain each converged faster than the uniform one. While each of them showed similar convergence rates, the variant with a square number of swaps was a bit faster. We can use these information to optimize our implementation. We switched to a non-uniform sampling strategy by reducing the number of swaps to $\Theta(n^2)$ and increased the number of random rankings, so we don't lose accuracy.

5 Conclusion

We successfully implemented our method to approximate the average ranking of a partial order. We used an approach by Karzanov and Khachiyan [KK91] for uniform sampling, that shuffles a ranking, by swapping two random neighbors $\Theta(n^3 \log n)$ times, until it is approximately uniform. Averaging many of these samples gives a ranking, that is close to the exact average. Reducing the number of swaps also reduces the computation time for each sample, but increases the number of required samples to get a decent result, because the samples are no longer uniform. An even faster way to sample rankings is to pick random leafs from the topological graph of the PO. Comparing the convergence of these methods showed, that the topological method is faster than the others, but gives unreliable result, since it converges to a ranking that differs from the exact average. Adding weight improved its accuracy, but still didn't give reliable results. The non-uniform variants of the Karzanov–Khachiyan Markov chain, however, converged correctly and faster than the uniform approach. Adapting our implementation accordingly gave it a considerable improvement of run time, enabling it to handle even highly variant POs with about 100 element. Each of these methods is highly parallelizable, since each sample ranking can be computed in a separate thread.

6 Future Work

So far, our algorithm can handle partial orders with up to 100 elements. This could be improved by further optimizing the algorithms, that generate the random rankings, or by finding tighter bounds for the variances, reducing the number of required rankings. However, the potential improvement here is limited and its unlikely that the algorithm will be able to handle much more elements this way, but it can decrease the required time.

Picking random elements uniformly from the topological sorting proved as an unsuitable method for the generation of random rankings. Our weighted approach gave better results, but was still not suited for this task. It could be an interesting challenge to try and find a weighting that represents the actual distribution and leads to a convergence to the correct average ranks, but it is unclear if such a weighting even exists, that fits all possible partial orders. Finding an appropriate weighting would greatly improve the speed to compute average rankings, since this approach showed a promising run time and convergence rate.

A R-Package Documentation

This appendix shows the documentation of our implementation. This R-package accesses the functions of the C-library to compute average rankings. The actual computations are executed by the C-library, because they require a lot of time and this can be done very efficiently by C. R is generally much slower, but it can access the functions much more conveniently. It also offers good plotting functionalities for the appraisal of our algorithms.
The C-library can also be accessed outside of R, but it is recommended to use this package for convenient access.

Package 'AverageRankings'

November 10, 2016

Type Package

Title Compute an Approximated Average Ranking of a Partial Order

Version 1.0.0

Author Tim Zeiss

Maintainer Tim Zeiss <timzeiss@hotmail.de>

Description
Computing exact average ranking usually reaches its limit at about 10 elements. With this package we give an approximative approach to this problem and raise the bound to about 100 elements.

License LGPL

LazyData TRUE

Imports relations

Suggests Rgraphviz

RoxygenNote 5.0.1

R topics documented:

bruteForce 2
getAverageRankingOptimized 2
getAverageRankingUniform 3
getConvMarkov 4
getConvPlotNew 4
getConvPlotRand 5
getConvPlotRandEx 6
getConvTop 6
getConvTopWeight 7
getIterations 7
getTopSort 8
getVariances 8
plotPO 9
pointsToMatrix 9

Index **10**

bruteForce *Brute force calculation of an average ranking*

Description

Compute the exact average ranking of a partial order by a brute force algorithm

Usage

```
bruteForce(matrix)
```

Arguments

matrix	The patrial order as a matrix

Details

Gives an exact average ranking by using a very slow brute force algorithm. Therefor, it should only be used for small partial orders with up to about 10 elements.

Value

the average ranking of the patrial order

Examples

```
bruteForce(matrix(c(0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1,1,0,0,0,0,0,1,1,0,0,0,1,1,1,1,1,0), nrow=6))
```

getAverageRankingOptimized *Compute an average ranking*

Description

Compute an average ranking

Usage

```
getAverageRankingOptimized(points, dim, matrix, s, p, i, threads, startSeed,
  plot = FALSE)
```

Arguments

points	A set of points of dimension dim as a single vector
dim	The dimenstion of the points
matrix	A partial order given as a matrix
s	Desired standard deviation
p	Desired probability to keep the error within the standard deviation
i	Desired number of random rankings to compute the average ranking

threads	The desired number of cuncurrent threads
startSeed	Seed to be used by the RNG for the first thread. The following threads will choose their seed according to the startSeed
plot	Gives the option to plot the Hasse Diagram of the partial order

Details

This is the main function of the library. It calls all the necessary subfunctions to compute an average ranking. It takes as input either a partial order as a matrix or a set of points together with their dimension. The number of rankings to be averaged can either be chosen by the user or will be calculated according to the standard deviation and its probability. By default, the standard deviation is set to 0.5 and the probability to keep within its bound to 90 If needed, the Hasse Diagramm of the partial order can be plottet, showing the resulting average ranks for each element.

`getAverageRankingUniform`

Compute an average ranking

Description

Compute an average ranking

Usage

```
getAverageRankingUniform(points, dim, matrix, s, p, i, seed)
```

Arguments

points	A set of points of dimension dim as a single vector
dim	The dimenstion of the points
matrix	A partial order given as a matrix
s	Desired standard deviation
p	Desired probability to keep the error within the standard deviation
i	Desired number of random rankings to compute the average ranking
seed	Seed to be used by the RNG

Details

LEGACY FUNCTION: It is recommended to use "gerAverageRankingOptimized" instead. This function computes an average ranking by averaging uniform sample rankings. Siince uniform sampling is rather slow, an optimized version of this function has been created.

`getConvMarkov` *Gives the convergence rate for a given partial order (DEBUGGING FUNCTION)*

Description

Gives the convergence rate for a given partial order (DEBUGGING FUNCTION)

Usage

```
getConvMarkov(points, dim, matrix, s, p, i, seed, timeStep, o)
```

Arguments

`points`	A set of points of dimension dim as a single vector
`dim`	The dimenstion of the points
`matrix`	A partial order given as a matrix
`s`	Desired standard deviation
`p`	Desired probability to keep the error within the standard deviation
`i`	Desired number of random rankings to compute the average ranking
`seed`	Seed to be used by the RNG for the first thread. The following threads will choose their seed according to the startSeed
`timeStep`	Time in ms after which a sample for the convergence rate is taken
`o`	The number of swaps that the Markov chain takes. Should be given in relation to the size of the PO

`getConvPlotNew` *Plots the average convergence rate of a given random partial orders after several runs (DEBUGGING FUNCTION)*

Description

Plots the average convergence rate of a given random partial orders after several runs (DEBUGGING FUNCTION)

Usage

```
getConvPlotNew(points, dim, matrix, s, p, i, seed, timeStep, plotStart, nRuns)
```

Arguments

points	A set of points of dimension dim as a single vector
dim	The dimenstion of the points
matrix	A partial order given as a matrix
s	Desired standard deviation
p	Desired probability to keep the error within the standard deviation
i	Desired number of random rankings to compute the average ranking
seed	Seed to be used by the RNG for the first thread. The following threads will choose their seed according to the startSeed
timeStep	Time in ms after which a sample for the convergence rate is taken
plotStart	Specifies the window for the plots
nRuns	The number of runs

---

getConvPlotRand *Plots the average convergence rate for a number of random partial orders (DEBUGGING FUNCTION)*

Description

Plots the average convergence rate for a number of random partial orders (DEBUGGING FUNCTION)

Usage

```
getConvPlotRand(s, p, i, seed, timeStep, plotStart, nRuns)
```

Arguments

s	Desired standard deviation
p	Desired probability to keep the error within the standard deviation
i	Desired number of random rankings to compute the average ranking
seed	Seed to be used by the RNG for the first thread. The following threads will choose their seed according to the startSeed
timeStep	Time in ms after which a sample for the convergence rate is taken
plotStart	Specifies the window for the plots
nRuns	The number of random partial orders to be plotted

`getConvPlotRandEx` *Plots the average deviation from the exact average ranking for a number of random partial orders (DEBUGGING FUNCTION)*

Description

Plots the average deviation from the exact average ranking for a number of random partial orders (DEBUGGING FUNCTION)

Usage

```
getConvPlotRandEx(s, p, i, seed, timeStep, plotStart, nRuns)
```

Arguments

`s`	Desired standard deviation
`p`	Desired probability to keep the error within the standard deviation
`i`	Desired number of random rankings to compute the average ranking
`seed`	Seed to be used by the RNG for the first thread. The following threads will choose their seed according to the startSeed
`timeStep`	Time in ms after which a sample for the convergence rate is taken
`plotStart`	Specifies the window for the plots
`nRuns`	The number of random partial orders to be plotted

`getConvTop` *Gives the convergence rate for a given partial order (DEBUGGING FUNCTION)*

Description

Gives the convergence rate for a given partial order (DEBUGGING FUNCTION)

Usage

```
getConvTop(points, dim, matrix, s, p, i, seed, timeStep)
```

Arguments

`points`	A set of points of dimension dim as a single vector
`dim`	The dimenstion of the points
`matrix`	A partial order given as a matrix
`s`	Desired standard deviation
`p`	Desired probability to keep the error within the standard deviation
`i`	Desired number of random rankings to compute the average ranking
`seed`	Seed to be used by the RNG for the first thread. The following threads will choose their seed according to the startSeed
`timeStep`	Time in ms after which a sample for the convergence rate is taken

getConvTopWeight — *Gives the convergence rate for a given partial order (DEBUGGING FUNCTION)*

Description

Gives the convergence rate for a given partial order (DEBUGGING FUNCTION)

Usage

```
getConvTopWeight(points, dim, matrix, s, p, i, seed, timeStep)
```

Arguments

points	A set of points of dimension dim as a single vector
dim	The dimenstion of the points
matrix	A partial order given as a matrix
s	Desired standard deviation
p	Desired probability to keep the error within the standard deviation
i	Desired number of random rankings to compute the average ranking
seed	Seed to be used by the RNG for the first thread. The following threads will choose their seed according to the startSeed
timeStep	Time in ms after which a sample for the convergence rate is taken

getIterations — *Compute the required number of Iterations*

Description

Compute the required number of Iterations

Usage

```
getIterations(matrix, s, p)
```

Arguments

matrix	A partial order as matrix
s	Standard deviation
p	Probability

Value

The required Iterations to reach the specified s and p

Examples

```
getIterations(matrix(c(0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1,1,0,0,0,0,0,1,1,0,0,0,1,1,1,1,1,0), nrow=6), 0
```

getTopSort — *Compute a deterministic ranking of a partial order from its topological sorting*

Description

Compute a deterministic ranking of a partial order from its topological sorting

Usage

```
getTopSort(matrix)
```

Arguments

matrix	The partial order as a matrix

Details

This function computes a deterministic ranking of a partial order, that is needed as starting ranking to compute a random ranking

Value

A deterministic ranking of the partial order

Examples

```
getTopSort(matrix(c(0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1,1,0,0,0,0,0,1,1,0,0,0,1,1,1,1,1,0)))
```

getVariances — *Computes the approximated variance for each element of the partial order*

Description

Computes the approximated variance for each element of the partial order

Usage

```
getVariances(matrix)
```

Arguments

matrix	The partial order as a matrix

Value

The approximated variance of each element as a vector

Examples

```
getVariances(matrix(c(0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1,1,0,0,0,0,0,1,1,0,0,0,1,1,1,1,1,0), nrow=6))
```

plotPO *Plot a partial order*

Description

Plot the Hasse diagram of the partial order. If provided, it also displays the (average) rank of each element.

Usage

```
plotPO(matrix, ranks)
```

Arguments

matrix	The partial order as a matrix
ranks	The rank of each elemetn as a vector

Examples

```
plotPO(matrix(c(0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1,1,0,0,0,0,0,1,1,0,0,0,1,1,1,1,1,0), nrow=6))
```

pointsToMatrix *Convert a Set of points into its partial order as a matrix*

Description

Convert a Set of points into its partial order as a matrix

Usage

```
pointsToMatrix(points, dim)
```

Arguments

points	A set of points of dimension dim as a single vector
dim	The dimenstion of the points

Value

The partial order of the points as a matrix

Examples

```
pointsToMatrix(c(1,6,5,5,3,3,1,4,2,2,0,0), 2)
```

Index

bruteForce, 2

getAverageRankingOptimized, 2
getAverageRankingUniform, 3
getConvMarkov, 4
getConvPlotNew, 4
getConvPlotRand, 5
getConvPlotRandEx, 6
getConvTop, 6
getConvTopWeight, 7
getIterations, 7
getTopSort, 8
getVariances, 8

plotPO, 9
pointsToMatrix, 9

Bibliography

[BD99] R. Bubley, M. Dyer. “Faster random generation of linear extensions.” In: *Discrete mathematics* 201.1 (1999), pp. 81–88 (cit. on p. 13).

[KK91] A. Karzanov, L. Khachiyan. “On the conductance of order Markov chains.” In: *Order* 8.1 (1991), pp. 7–15 (cit. on pp. 13, 16, 29).

[Wil04] D. B. Wilson. “Mixing times of lozenge tiling and card shuffling Markov chains.” In: *Annals of Applied Probability* (2004), pp. 274–325 (cit. on pp. 13, 16, 26).

All links were last followed on March 17, 2008.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature