✸ Discrepancies for generalized Halton points Comparison of three heuristics for generating points set

Thomas Espitau[❄] and Olivier Marty[❄]

[❄] ENS Cachan

Abstract. Geometric discrepancies are standard measures to quantify the irregularity of distributions. They are an important notion in numerical integration. One of the most important discrepancy notions is the so-called star discrepancy. Roughly speaking, a point set of low star discrepancy value allows for a small approximation error in quasi-Monte Carlo integration. In this work we present a tool realizing the implementation of three basics heuristics for the construction of low discrepancy points sets in the generalized Halton model: fully random search, local search with simulated annealing and genetic $(5 + 5)$ search with a ad-hoc crossover function.

1 General architecture of the tool

The testing tool is aimed to be modular: it is made of independents blocks that are interfaced trough a scheduler. More precisely a master wrapper is written in Python that calls a first layer which performs the chosen heuristic. This layer is written in C++ for performances. The given discrepancy algorithm — written in C — is called when evaluations of a state is needed. The wrapper dispatch the computations on the multi-core architecture of modern computers[1]. This basic architecture is described in figure 1. More precisely the class diagram for the unitary test layer is presented in figure 2. Experiments were conducted on two machines:

- 2.4 GHz Intel Dual Core i5 hyper-threaded to 2.8GHz, 8 Go 1600 MHz DDR3.
- 2.7 GHz Intel Quad Core i7 4800MQ hyper-threaded to 3.7GHz, 16 Go 1600 MHz DDR3.

On these machines, some basic profiling has made clear that the main bottleneck of the computations is hiding in the *computation of the discrepancy*. The chosen algorithm and implantation of this cost function is the DEM-algorithm [DEM96] of *Magnus Wahlstrøm* [Wah].

All the experiments has been conducted on dimension 2,3, and 4 — with a fixed Halton prime basis 7, 13, 29, and 3. Some minor tests have been made in

[1] for us, between 2 and 4 physical cores and 4 or 8 virtual cores

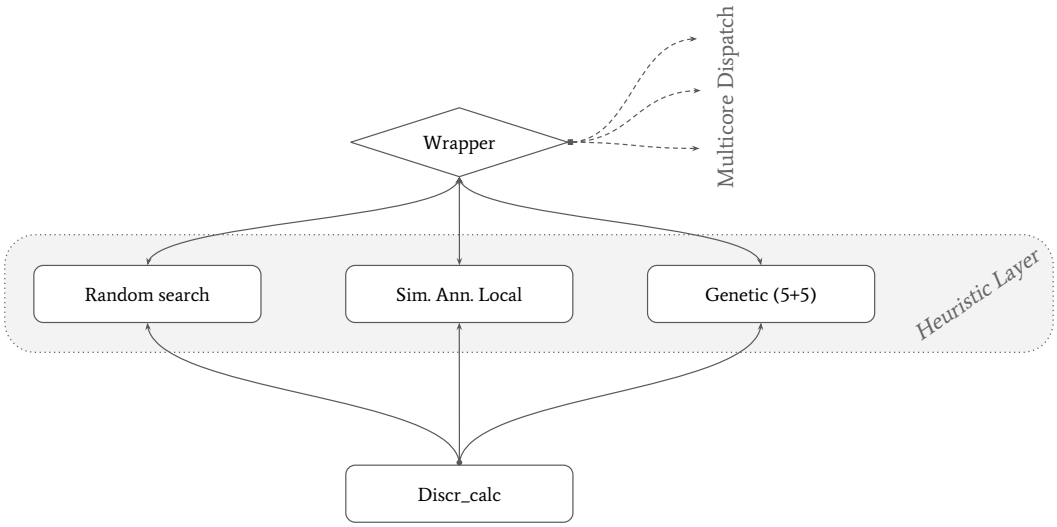

Fig. 1: Tool overview

order to discuss the dependency of the discrepancy and efficiency of the heuristics with regards to the values chosen for the prime base. The average results remains roughly identical when taking changing these primes and taking them in the range [2, 100]. For such a reason we decided to pursue the full computations with a fixed basis.

1.1 Algorithmic insights

To perform an experiment we made up a loop above the main algorithm which calls the chosen heuristic multiple times in order to smooth up the results and obtain more exploitable datas. Then an arithmetic mean of the result is performed on the values. In addition extremal values are also given in order to construct error bands graphs.

Graph are presented not with the usual box plots to show the error bounds, but in a more graphical way with error bands. The graph of the mean result is included inside a band of the same color which represents the incertitude with regards to the values obtained.

A flowchart of the conduct of one experiment is described in the flowchart 3. The number of iteration of the heuristic is I and the number of full restart is N. The function Heuristic() corresponds to a single step of the chosen heuristic.

We now present an in-depth view of the implemented heuristics.

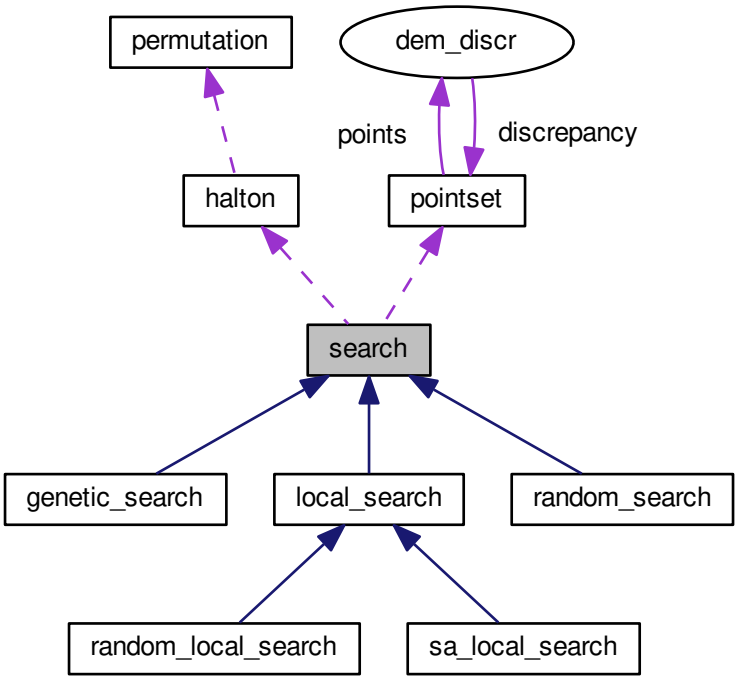

Fig. 2: Class dependencies

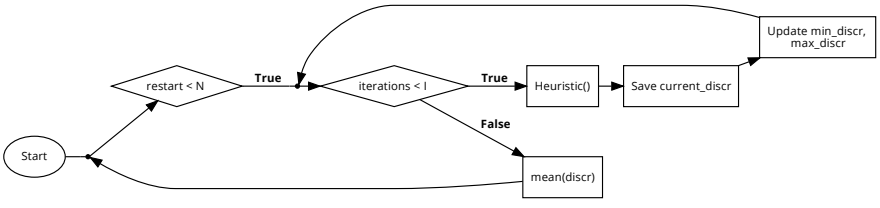

Fig. 3: Flowchart of a single experiment

2 Heuristics developed

2.1 Fully random search (Test case)

The first heuristic implemented is the random search. We generate random permutations, compute the corresponding sets of Halton points and select the best set with regards to its discrepancy. The process is wrapped up in the flowchart 4. In order to generate at each step random permutations, we transform them directly from the previous ones. More precisely the permutation is a singleton object which have a method random, built on the Knuth Fisher Yates shuffle. This algorithm allows us to generate an uniformly chosen permutation at each step. We recall this fact and detail the algorithm in the following section.

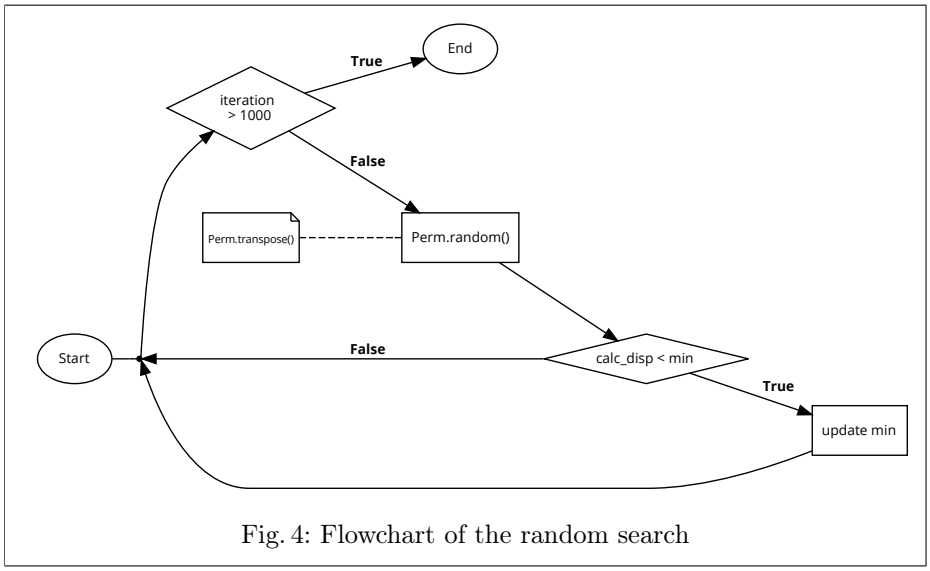

Fig. 4: Flowchart of the random search

The Knuth-Fisher-Yates shuffle The Fisher–Yates shuffle is an algorithm for generating a random permutation of a finite sets. The Fisher–Yates shuffle is unbiased, so that every permutation is equally likely. We present here the Durstenfeld variant of the algorithm, presented by Knuth in *The Art of Computer*

programming vol. 2 [Knu97]. The algorithm's time complexity is here $O(n)$, compared to $O(n^2)$ of the naive implementation.

Algorithm 1: KFY algorithm

```
Data: A table T[1..n]
Result: Same table T, shuffled
for i ← 1 to n − 1 do
    j ← Rand([1, n − i]);
    Swap(T[i], T[i + j]);
end
```

Lemma 1. *The resulting permutation of KFY is unbiased.*

Proof. Let consider the set $[1, \dots n]$ as the vertices of a random graph constructed as the trace of the execution of the algorithm: an edge (i, j) exists in the graph if and only if the swap of $T[i]$ and $T[j]$ had been executed. This graph encodes the permutation represented by T. To be able to encode any permutation the considered graph must be connected — in order to allow any pairs of points to be swapped. Since by construction every points is reached by an edge, and that there exists exactly $n - 1$ edges, we can conclude directly that any permutation can be reached by the algorithm. Since the probability of getting a fixed graph of $n - 1$ edges with every edges of degree at least one is $n!^{-1}$, the algorithm is thus unbiased.

Results and stability We first want to analyze the dependence of the results on the number of iterations of the heuristic, in order to discuss its stability. The results are compiled in the figures 5 and 6, restricted to a number of points between 80 and 180. We emphasize on the fact lot of datas appears on graphs, and error bands representation make them a bit messy. These graphs were made for extensive internal experiments and parameters researches. The final wrap up graphs are much more lighter and only present the best results obtained. As expected from a fully random search, error bands are very large for low number of iterations (15% of the value for 400 iterations) and tend to shrink with a bigger number of iterations (around 5% for 1500 iterations). This shrinkage is a direct consequence of well known concentrations bounds (Chernoff and Asuma-Hoeffding). The average results are quite stable, they decrease progressively with the growing number of iterations, but seem to get to a limit after 1000 iterations. This value acts as a threshold for the interesting number of iterations. As such interesting results can be conducted with *only* 1000 iterations, without altering too much the quality of the set with regards to its discrepancy and this heuristic.

2.2 Local search with simulated annealing

The second heuristic implemented is a randomized local search with simulated annealing. This heuristic is inspired by the physical process of annealing in metallurgy. Simulated annealing interprets the physical slow cooling as a slow

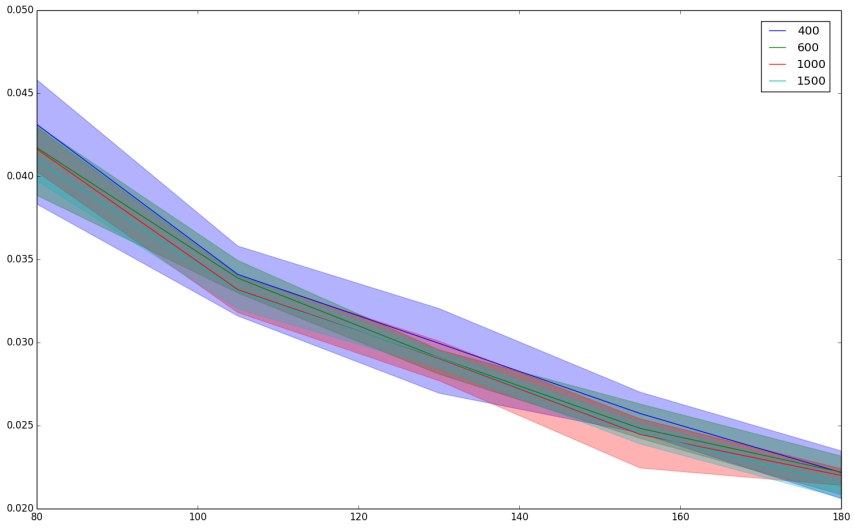

Fig. 5: Dependence on iterations, dimension 2

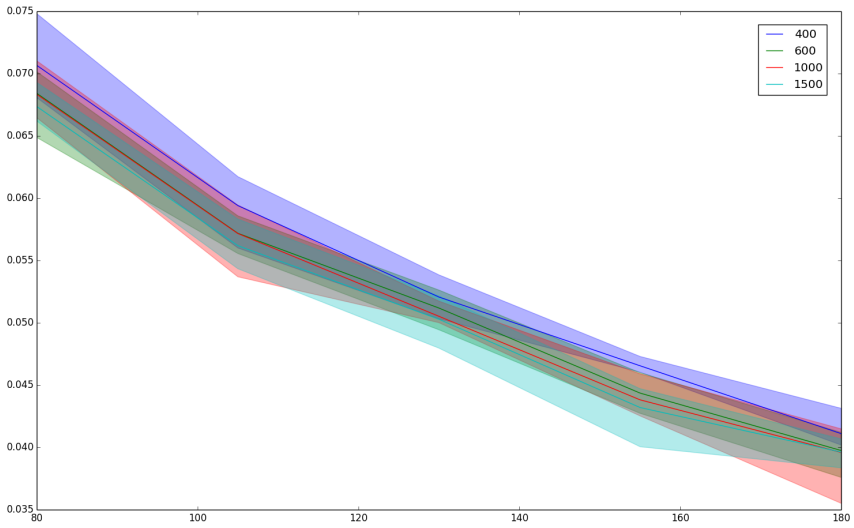

Fig. 6: Dependence on iterations, dimension 3

decrease in the probability of accepting worse solutions as it explores the solution space. More precisely a state is a d-tuple of permutations, one for each dimension, and the neighborhood is the set of d-tuple of permutations which can be obtained by application of exactly one transposition of one of the permutations of the current state. The selection phase is dependant on the current temperature: after selecting randomly a state in the neighborhood, either the discrepancy of the corresponding Halton set is decreased and the evolution is kept, either it does not but is still kept with a probability $e^{\frac{\delta}{T}}$ where δ is the difference between the old and new discrepancy, and T the current temperature. If the discrepancy has decreased, the temperature T is multiplied by a factor λ (fixed to 0.992 in all our simulations), hence is decreased. The whole algorithm is described in the flowchart 7.

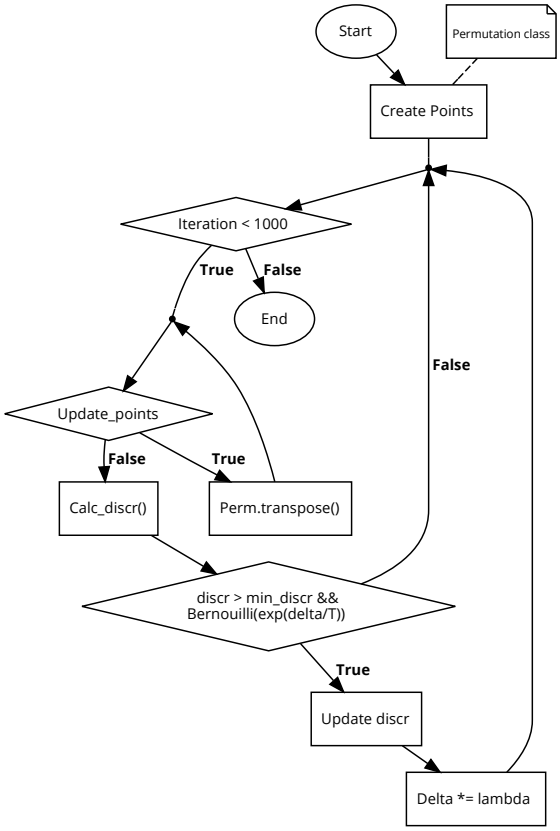

Fig. 7: Flowchart of the local search with simulated annealing heuristic

Dependence on the temperature First experiments were made to select the best initial temperature. Results are compiled in graphs 8, 9, and 10. Graphs 8 and 9 represent the results obtained respectively in dimension 2 and 3 between 10 and 500 points. The curve obtained is characteristic of the average evolution of the discrepancy optimization algorithms for Halton points sets: a very fast decrease for low number of points — roughly up to 80 points — and then a very slow one after [DDR13]. The most interesting part of these results are concentrated between 80 and 160 points were the different curves splits. The graph 10 is a zoom of 9 in this window. We remark on that graph that the lower the temperature is, the best the results are, with a threshold at 10^{-3}.

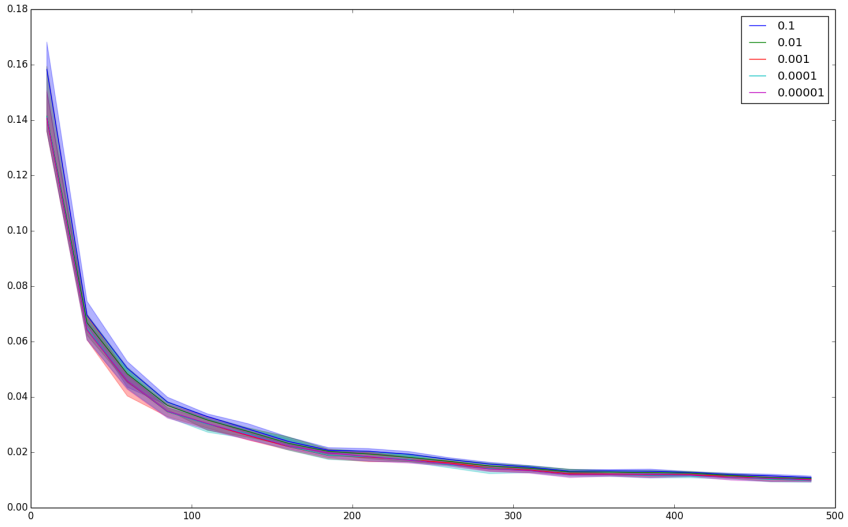

Fig. 8: Dependence on initial temperature: D=2

Stability with regards to the number of iterations As for the fully random search heuristic we investigated the stability of the algorithm with regards to the number of iterations. We present the result in dimension 3 in the graph 11. Once again we restricted the window between 80 and 180 points were curves are split. An interesting phenomena can be observed: the error rates are somehow invariant w.r.t. the number of iterations and once again the 1000 iterations threshold seems to appear — point 145 is a light split between iteration 1600 and the others, but excepted for that point, getting more than 1000 iterations tends be be a waste of

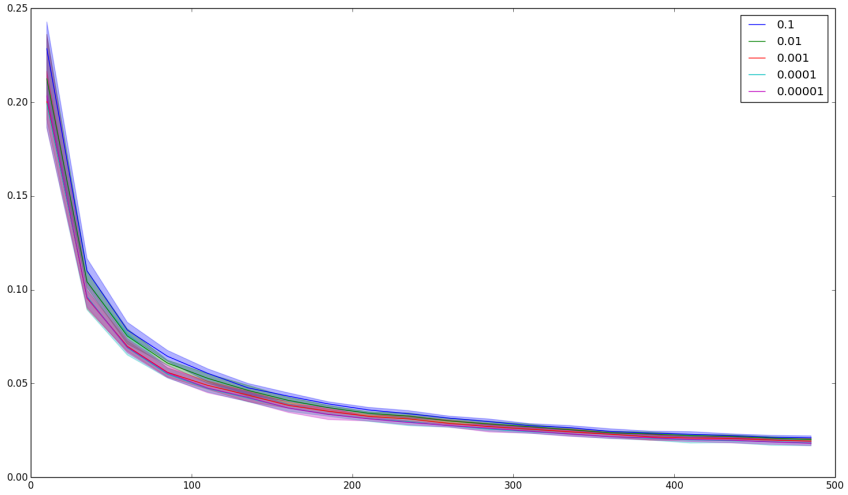

Fig. 9: Dependence on initial temperature: D=3

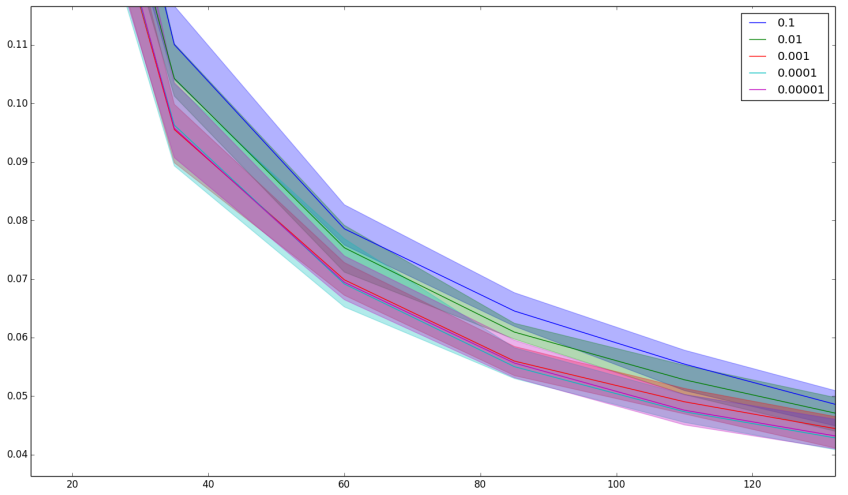

Fig. 10: Dependence on initial temperature (zoom): D=3

time. The error rate is for 80 points the biggest and is about 15% of the value, which is similar to the error rates for fully random search with 400 iterations.

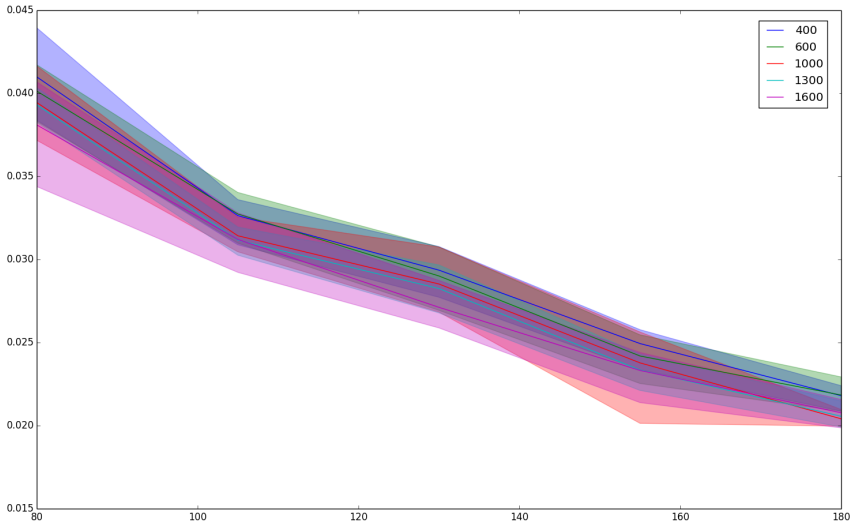

Fig. 11: Dependence on iterations number for simulated annealing : D=3

2.3 Genetic $(\mu + \lambda)$ search

The third heuristic implemented is the $(\mu + \lambda)$ genetic search. This heuristic is inspired from the evolution of species: a family of μ genes is known (they are generated randomly at the beginning), from which λ new genes are derived. A gene is the set of parameters we are optimizing, i.e. the permutations. Each one is derived either from one gene applying a mutation (here a transposition of one of the permutations), or from two genes applying a crossover: a blending of both genes (the algorithm is described in details further). The probability of making a crossover rather than a mutation is c, the third parameter of the algorithm, among μ and λ. After that, only the μ best genes are kept, according to their fitness, and the evolutionary process can start again.

Because making variations over μ or λ does not change fundamentally the algorithm, we have chosen to fix $\mu = \lambda = 5$ once and for all, which seemed to be a good trade-off between the running time of each iteration and the size of the family.

Crossover algorithm We designed an ad-hoc crossover for permutations. The idea is simple: given two permutations A and B of $\{1..n\}$, it constructs a new permutation C value after value, in a random order (we use our class permutation for this). For each index i, we take either A_i or B_i. If exactly one of those values is available (understand it was not already chosen) we choose it. If both are available, we choose randomly and we remember the second. If both are unavailable, we choose a remembered value. The flowchart of this algorithm is described in figure 12.

The benefits of this method are that it keeps common values of A and B, the values C_i are often among $\{A_i, B_i\}$ (hence C is close to A and B), and it does not favor either the beginning or the ending of permutations.

Algorithm 2: Permutations crossover

Data: Two permutations A[1..n], B[1..n]
Result: A permutation C[1..n]
$pi \leftarrow$ `a random permutation of` $\{1, \ldots, n\}$;
$available \leftarrow \{\}$;
$got \leftarrow \{\}$;
for $i \leftarrow 1$ **to** n **do**
  $j \leftarrow pi_i$;
  $a \leftarrow A_j$;
  $b \leftarrow B_j$;
  if $a \in got \wedge b \in got$ **then**
    $v \leftarrow$ `a random value in` $available$;
  end
  else if $a \in got$ **then**
    $v \leftarrow b$;
  end
  else if $b \in got$ **then**
    $v \leftarrow a$;
  end
  else
    `Swap`(A, B) with probability $1/2$;
    $v \leftarrow A_j$;
    $available \leftarrow available \cup \{B_j\}$;
  end
  $C_j \leftarrow v$;
  $got \leftarrow got \cup \{v\}$;
  $available \leftarrow available \setminus \{v\}$;
end

Dependence on the parameter c First experiments were made to select the value for the crossover parameter c. Results are compiled in graphs 13, 14,15 and 16. Graph 13, represents the results obtained in dimension 2 between 10 and 500 points. The curve obtained is, with no surprise again, the characteristic curve

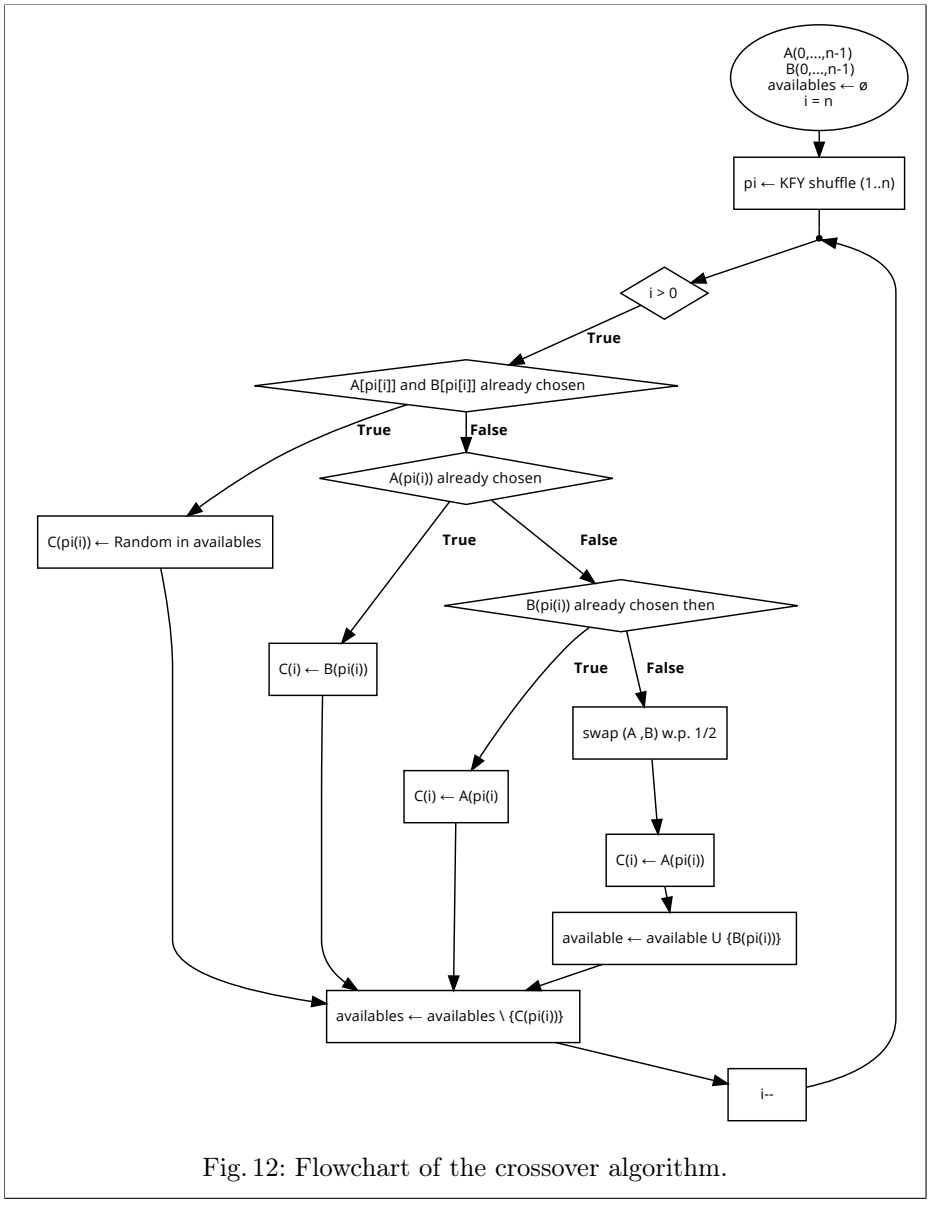

Fig. 12: Flowchart of the crossover algorithm.

of the average evolution of the discrepancy we already saw with the previous experiments. The most interesting part of these results are concentrated — once again — between 80 and 160 points were the different curves splits. The graph 14 is a zoom of 13 in this window, and graphs 15 and 16 are focused directly into it too. We remark that in dimension 2, the results are better for c close to 0.5 whereas for dimension 3 and 4 the best results are obtained for c closer to 0.1, that is a low probability of making a crossover.

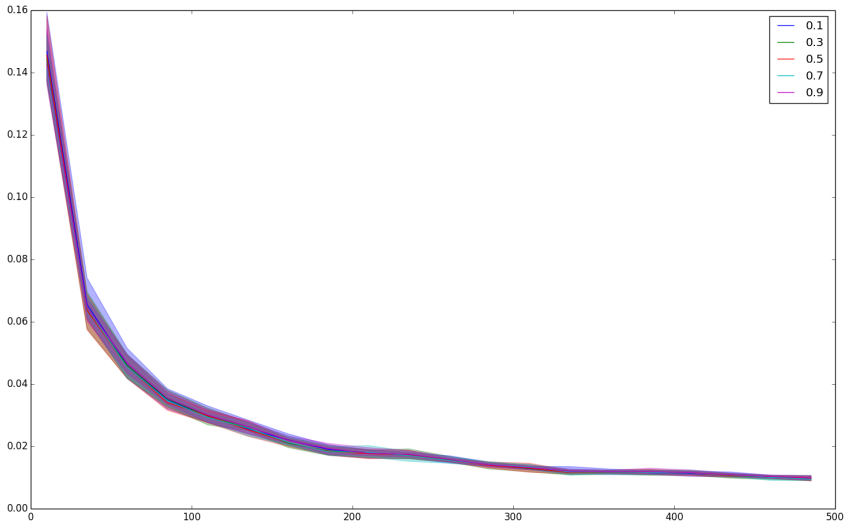

Fig. 13: Dependence on parameter c: D=2

Stability Once again we investigated the stability of the algorithm with regards to the number of iterations. Once again we restricted the window between 80 and 180 points were curves are split. The results are compiled in graph 17. An interesting phenomena can be observed: the error rates are getting really big for 1400 iterations at very low points (up to 120), even if the average results are stables after the threshold 1000 iterations, like we get before.

3 Results and conclusions

Eventually we made extensive experiments to compare the three previously presented heuristics. The parameters chosen for the heuristics have been guessed using the experiments conducted in the previous sections. Results are compiled

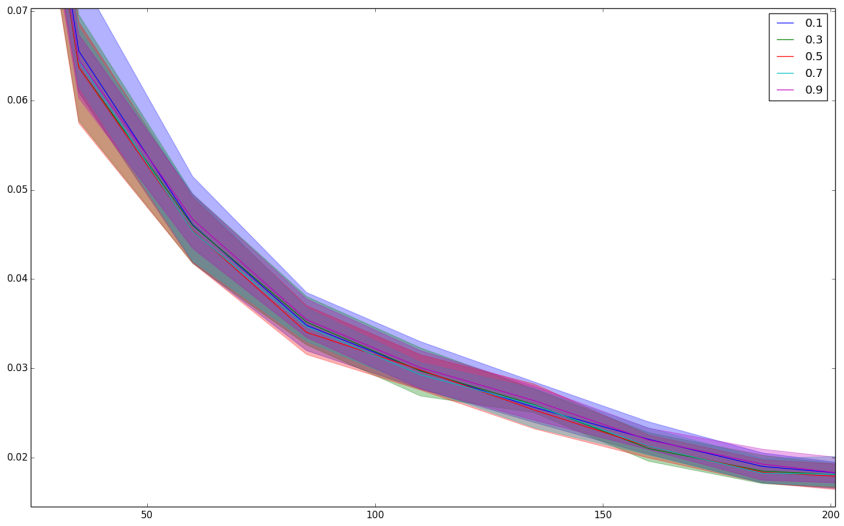

Fig. 14: Dependence on parameter c (zoom): D=2

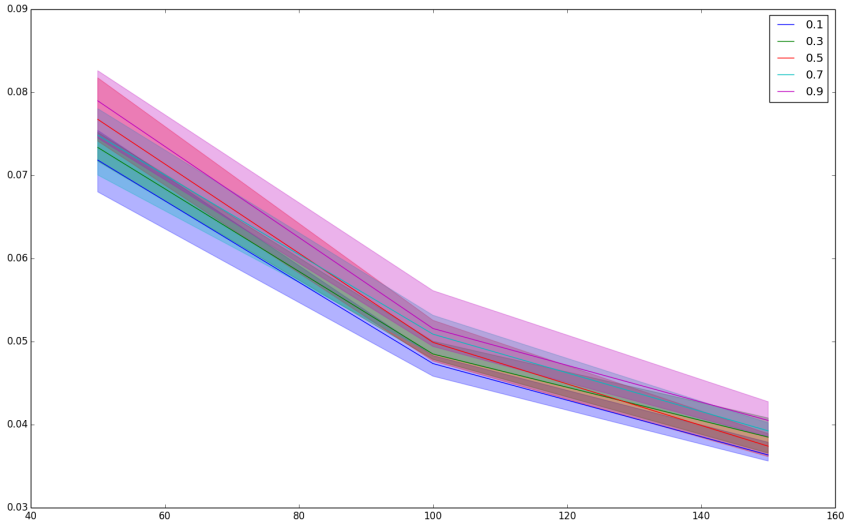

Fig. 15: Dependence on parameter c: D=3

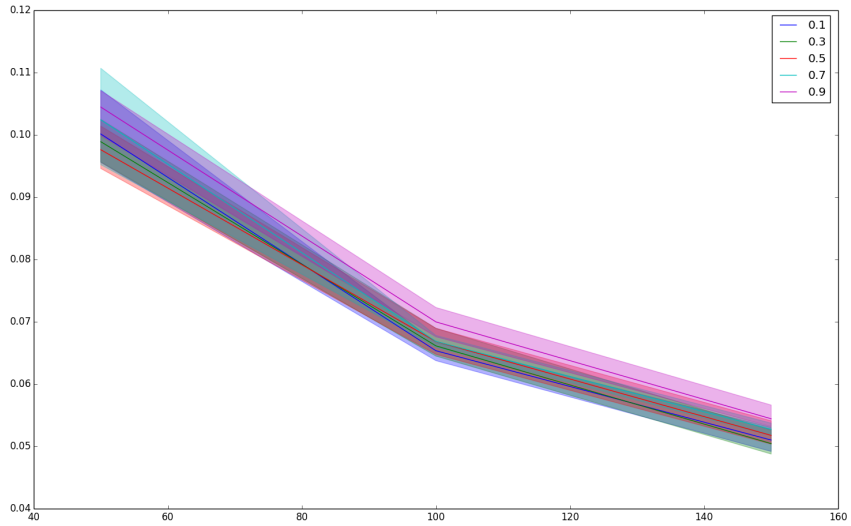

Fig. 16: Dependence on parameter c: D=4

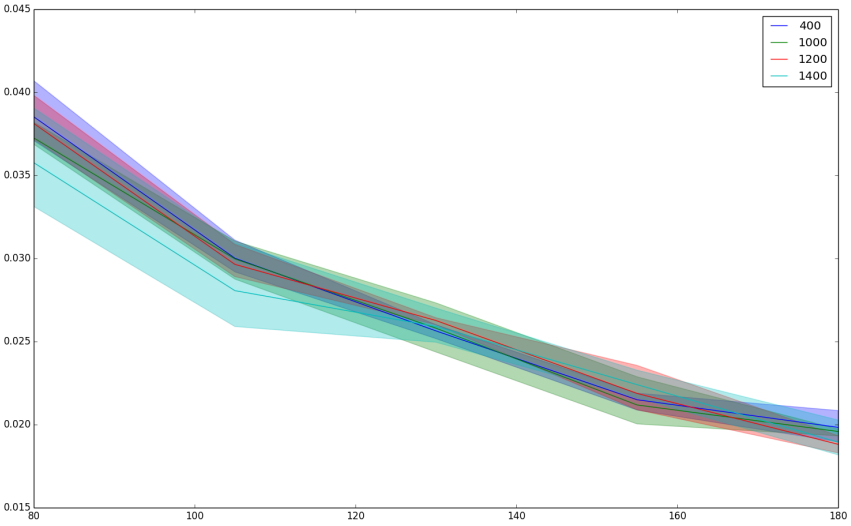

Fig. 17: Stability w.r.t. number of iterations: D=2

in the last figures 18, 19, 20, and 21. The recognizable curve of decrease of the discrepancy is still clearly recognizable in the graph 18, made for points ranged between 10 and 600. We then present the result in the — now classic — window 80 points - 180 points. For all dimensions, the superiority of non-trivial algorithms — simulated annealing and genetic search — is clear over fully random search. Both curves for these heuristics are way below the error band of random search. As a result *worse average results of non trivial heuristics are better than best average results when sampling points at random.* In dimension 2 19, the best results are given by the simulated annealing, whereas in dimension 3 and 4 20, 21, best results are given by genetic search. It is also noticeable that in that range of points the error rates are roughly the same for all heuristics: *for 1000 iteration, the stability of the results is globally the same for each heuristic.*

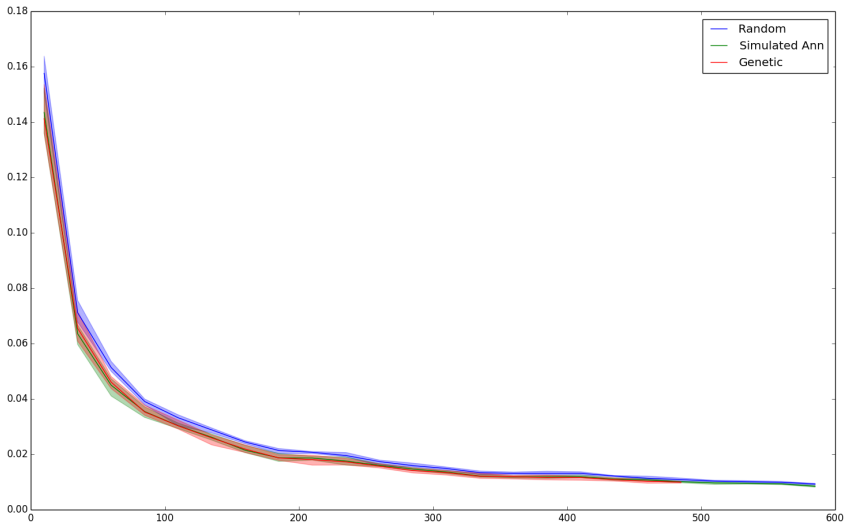

Fig. 18: Comparison of all heuristics: D=2

Acknowledgments

We would like to thank Magnus Wahlstrom from the Max Planck Institute for Informatics for providing an implementation of the DEM algorithm. We would also like to thank Christoph Dürr and Carola Doerr for several very helpful talks on the topic of this work. Both Thomas Espitau and Olivier Marty are supported by the French Ministry for Research and Higher Education, trough the École Normale Supérieure.

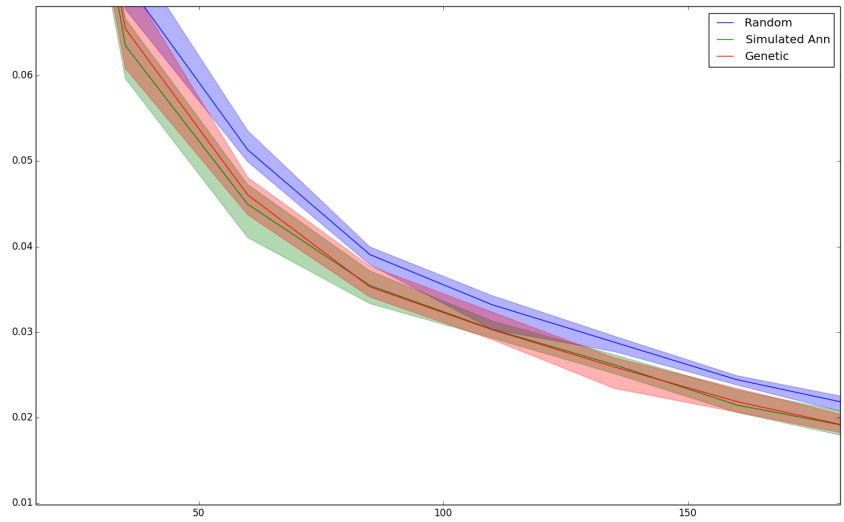

Fig. 19: Comparison of all heuristics (zoom): D=2

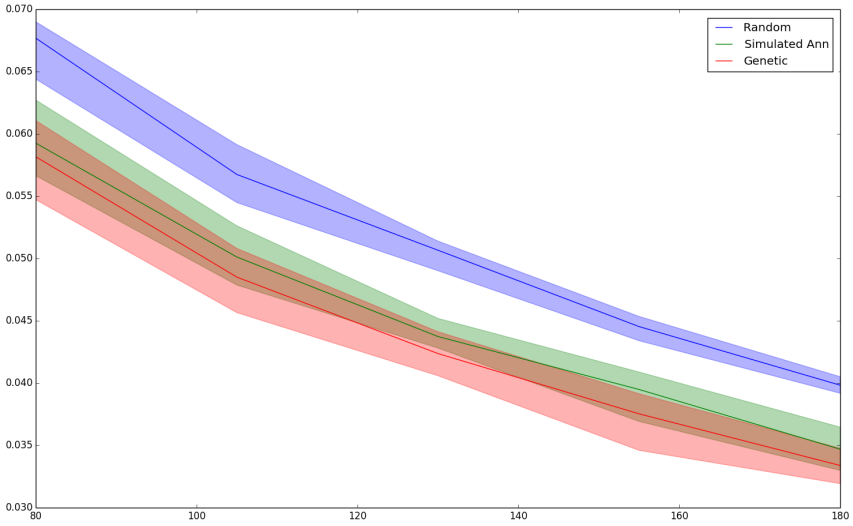

Fig. 20: Comparison of all heuristics: D=3

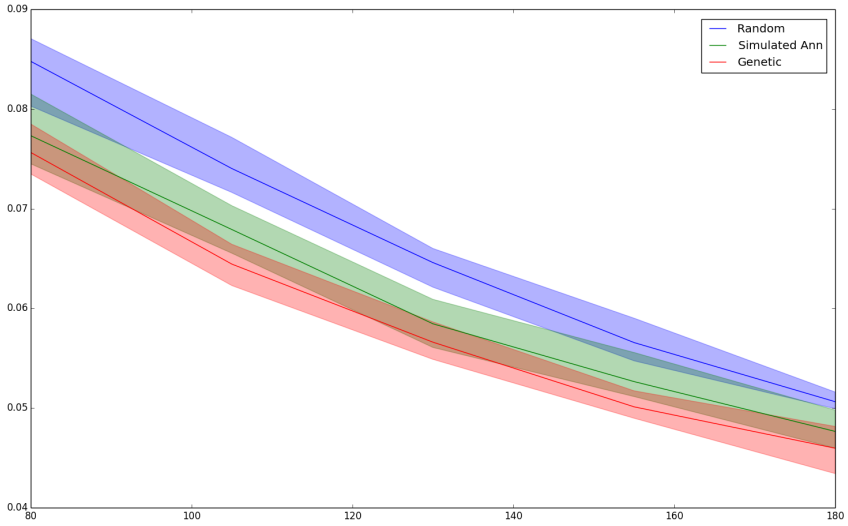

Fig. 21: Comparison of all heuristics: D=4

References

[DDR13] Carola Doerr and François-Michel De Rainville. Constructing low star discrepancy point sets with genetic algorithms. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, pages 789–796, New York, NY, USA, 2013. ACM.

[DEM96] David P. Dobkin, David Eppstein, and Don P. Mitchell. Computing the discrepancy with applications to supersampling patterns. *ACM Trans. Graph.*, 15(4):354–376, October 1996.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[Wah] Magnus Wahlström. discrcalc.tar.gz.