Sequential Monte Carlo for solving puzzles

Adrien Vacher
ENSAE Paristech
adrien.vacher@ensae.fr

Abstract

In this project we applied a tempering SMC algorithm to solve puzzles of different size. We compared the efficiency of a local-based loss against a global-based loss. The global-based loss significantly outperforms the local-based loss, in terms of computational speed and of convergence. It also outperforms the combination of the two.

1 Statement and complexity of the problem

First, let us state the problem without taking in account its inherent geomtric structure. In this case, solving a puzzle of size d simply amounts in finding a given permutation $\sigma^* \in \mathbb{S}_d$, with the ability to know *pointwise* if a candidate permutation is the solution or not. This more general problem is combinatorial.

The complexity of the problem can greatly change depending on the level of feedback. For instance, suppose that for each candidate σ we know for which i $\sigma(i) == \sigma^*(i)$. One possible algorithm would be to apply transpositions to an initial random permutation. Once a point matches, we withdraw it and perform the same process on the remaining $d-1$ points. Since there are $\mathcal{O}(n^2)$ transpositions in $\mathbb{S}_n$, this algorithm is in $\mathcal{O}(\sum^d k^2) = \mathcal{O}(d^3)$ which is polynomial. However we will restrict ourselves to a case where only an *aggregated* feedback is available. This is where the grid-like structure of our problem can appear.

2	7	6
9	5	1
4	3	8

Figure 1: Exemple of a 2d grid puzzle

In our setting, $d = n^2$ and the permutation σ^* is now a labelling of the grid $n \times n$ (see Figure 1). First one could define a global-based loss by accounting for the number of mislabled *pixels* in the grid. This naïve loss encourages a blind search and in particular, won't let the problem be solved locally. For instance, if you managed to solve your puzzle in each seperate quarter but in the wrong order, even if you are actually very close to the solution, this loss will give the maximum possible penalty to this setting. This is why one could also define a local-based loss as the number of mismatching *edges* induced by the 2d-grid labelling of σ. Unlike the previous one, this loss encourges a local-to-global solving of the problem.

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.

In this project, we first focused on an efficient implementation of a tempering SMC algorithm with this local-based loss. Even though the loss decreased quickly, the algorithm entually stagnated before solving the puzzle. We then added to this loss a global-based loss and got the algorithm to converge. However, when we compared this combination to the SMC with global-based loss only, it turned out that the latter was in fact much faster for comparable convergence perfomances.

2 Generic algorithm and implementation details

2.1 Tempering SMC

This algorithm requires 6 inputs: a number of iterations T, a loss $\mathcal{L}$ known pointwise, an increasing sequence of $\lambda_{\mathbf{k}}$, a number of particles N, a motion distribution on permutations $q(.|x)$ and the number of steps n of the Metropolis-Hasting kernel. The full sketch is given in 1

Algorithm 1 Temperaing SMC

$loss \leftarrow \infty$
$particules \leftarrow \mathcal{U}(\mathbb{S}_d)$
for t $\in T$ **do**
  $weights \leftarrow \mathcal{L}(particles)$
  $a_i \sim \mathcal{M}(weights)$
  $particles \leftarrow particles(a_i)$
  $particles \leftarrow MH(particles)$
  $loss \leftarrow loss$

The algorithm is made of two core steps: one resampling step and one Metropolis-Hasting step. The resampling step aims to perform a filtering operation: given N particules, it resamples new particles according to $p_i(x) = \frac{1}{Z_i} \exp\left(-\lambda_i \mathcal{L}(x)\right)$ in order to keep the ones with the lowest loss. The MH step aims to explore for new solutions performing a series of local changes on the filtered particles. The resampling step can as well be understood as a way to make a more efficient exploration, discrading particles who are "far away" from the solution. The global anneanling scheme can also be understood in terms of exploration: at the begining of the algorithm, we sample from wide-spread distributions to explore many solutions and as the algorithm goes by, we exploit our existing solutions and sample from skewed distributions.

2.2 Implementation details

2.2.1 Hashing the grid

A central aspect of our project was the computational efficency of the algorithm while coded on Python (and not in C++). To achieve this purpose, we had to avoid making loops and in particular, we had to use NumPy as much as possible. In this perspective, computing the local-based loss was challenging. A naïve method would have been to browse the entire grid formed by the candidate, compute the edges and compare them to the one of the solution. However this approach was way too expensive so we had to introduce *hashing* functions.

The underlying idea of the hashing is to encode all the edges formed by the candidate on the grid by a sequence. For the purpose of efficiency, this encoding should be made only thanks to NumPy opearations on the grid. In our case, we want to find ϕ such that the set of coefficients of the column-wise (1) first order difference matrix (recall $d = n^2$)

$$\begin{pmatrix} \phi(\sigma(2)) - \phi(\sigma(1)) & \cdots & \phi(\sigma(n)) - \phi(\sigma(n-1)) \\ \vdots & \vdots & \vdots \\ \phi(\sigma(d+1-n)) - \phi(\sigma(d-n)) & \cdots & \phi(\sigma(d)) - \phi(\sigma(d-1)) \end{pmatrix} \tag{1}$$

and the row-wise first order difference matrix (2)

$$\begin{pmatrix} \phi(\sigma(n+1)) - \phi(\sigma(1)) & \cdots & \phi(\sigma(2n)) - \phi(\sigma(n-1)) \\ \vdots & \vdots & \vdots \\ \phi(\sigma(d+1-n)) - \phi(\sigma(d-2n)) & \cdots & \phi(\sigma(d)) - \phi(\sigma(d-n)) \end{pmatrix} \tag{2}$$

encode for the vertical and horizontal edges respectively. One *sufficient* way to achieve this encoding is to have ϕ s.t

$$\forall i \neq j, i^{'} \neq j^{'} \in \{1, \cdots, d\}, \phi(i) - \phi(j) = \phi(i^{'}) - \phi(j^{'}) \implies (i, j) = (i^{'}, j^{'})$$

We will admit that the exponential satisfies this condition and extend it to rationnal numbers. Therefore we are going to chose as $\phi \exp(./d)$ where d allows to rescale and avoid dealing with potentially hudge exponentials.

2.2.2 Metropolis-Hasting

The Metropolis-Hasting step requires to choose a motion distribution q that is efficient for exploring the solution space but also easy to simulate and compute. More precisely, when $y \sim q(.|x)$ one should be able to easily compute $q(x|y)$. For the reason, we chose q such that $y = \tau(x)$ where τ is a randomly drawn transposition. This simple distribution is such that $q(.|x)$ is constant and $q(x|y)$ and $q(y|x)$ cancel in the MH formula. However, as we will show in section 3, this too local transformation will be very harmfull in the case of the edge-based loss SMC.

3 Experimental results

Before presenting any results, it is important to emphasize that each of obtained curves or results were average on a number (about 10) of independent runs as we are dealing with stochastic algorithms.

3.1 Local-based loss

We tried to solve the puzzle for $d = 16$ using the local-based loss. We used $N = 500$ particles and $n = 50$ steps. The Figure 2 shows the evolution of the number of matching edges when iterating over the algorithm. Even if the plot shows a rapid decrease at the begining, the algorithm clearly

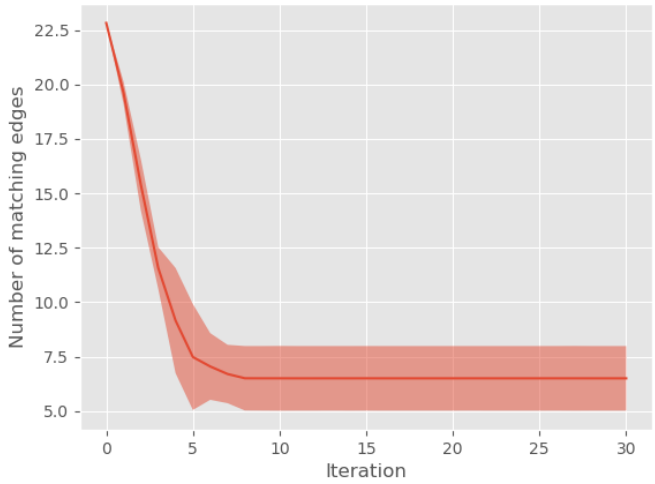

Figure 2: Number of non matching edges over iterations

stagnates. This scheme is not very surprising when we think of how the motion kernel was build. At the begining small transpositions are very benificial for solving the puzzle and most likely, the algorithm starts to sovle the puzzle fraction by fraction. However, once the sub-puzzles are solved, the existing particles need to have the transpositions at the scale of whole blocks which are never provided by the exploration kernel. Furthemore, not only the appropriate permutations don't appear in the kernel but also, because of the low global sensitivity of the loss, even if they were they would have low chances to be selected.

One natural response to those two issues is to change the kernel and add a global loss accounting for the number of mismatching pixels. In the kernel, we add a possibility to flip the permutation over a

randomly selected index i (once again $q(x|y) = q(y|x)$ in motion step). We also add an annealing scheme inside the kernel: at the begining, we favorise local transpositions and as time goes by, we favorise flips.

3.2 Improved SMC

The simultaions were done in the same conditions as above. We obtained the following results:

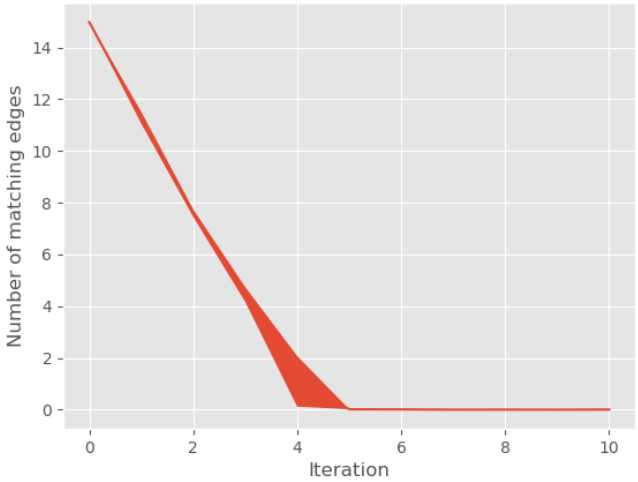

Figure 3: Composite loss over iterations

This time the algorithm converges. It takes 1.9 +- 0.2 seconds to execute (10 iterations). However, to properly assess this improvement, we need to observe the performance with the global-based loss only.

3.3 Global vs local + global

To distinguish between the two algorithms, we had to push for $d = 25$. Also, though we did not perofrm the search according to same loss, we plotted the results in term of (non) accuracy i.e number of non matching pixels. One can see on Figure 4 that the global based loss slightly outperforms in terms of accuracy, variance and number of steps to find a solution. However, the difference in perforrmances mainly come from the computational efficiency. Local is 12 secs +- 1.1 secs while global is 2.5 +- 0.3 secs. At this point, there is no possible matching.

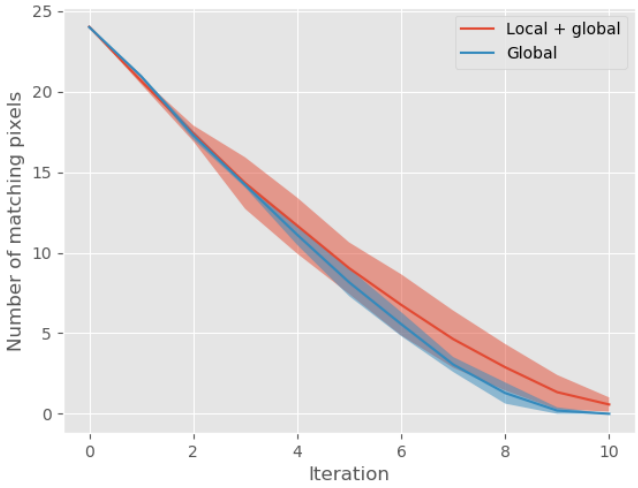

Figure 4: Comparasion of the two SMC versions

4 Conclusion and perspectives

As disappointing as it can be, the non-subtile approach clearly beat the more subtle approach. However, few improvements could be made to the latter:

- When the per-edge loss was computed, it wasn't parrallelized accros the whole batch of particles. If one could find a way to parrallelize these operations within the NumPy framweork, the computationnal time would be comparable.
- The motion kernel could be improved. For instance, it could explicitely divide the grid into an arbitrary number of sub-pieces and shuffle them. However, one must keep in mind that because of the aggregated feedback, one cannot perform a specific shuffle of already solved sub-puzzles.