Mini-project : Deploying a 5G Network in a country

Delio VICINI, Ahmad AJALLOEIAN and Heloise DUPONT DE DINECHIN (Team name: DHA)

Abstract

In this project, we use the Metropolis algorithm to optimize a complex geometric objective function. The objective function is motivated by the problem of optimally placing 5G antennas on a given map of cities. We would like to cover cities according to a per city value, while at the same time minimizing the spatial spread of the network. We show how to efficiently optimize the given objective function, making use of the geometric properties of the problem to implement a method which scales favorably in the number of considered cities.

1. Introduction

Given a map of cities, we would like to establish a network of 5G antennas. Due to constraints on the availability of a maintenance team, the network of antennas initially should be established in a localized region. The goal is to cover a significant portion of the cities, while not increasing the distance traveled by maintenance workers too much. Formally, the problem has been specified as a maximization of the objective function:

$$f(\lambda, \mathcal{S}) = \sum_{i \in \mathcal{S}} v_i - \lambda \cdot n \cdot \max_{(i,j) \in \mathcal{S}} \pi(d(x_i, x_j)/2)^2 \quad (1)$$

In practice, we will work on the following non-convex minimization objective:

$$f(\lambda, \mathcal{S}) = -\sum_{i \in \mathcal{S}} v_i + \lambda \cdot n \cdot \max_{(i,j) \in \mathcal{S}} \pi(d(x_i, x_j)/2)^2 \quad (2)$$

where v_i are the (normalized) population values associated to each city, x_i their 2D position, λ a scalar and n the total number of cities. The distance $d(x_i, x_j)$ is the euclidean distance between the cities i, j, and the set $\mathcal{S}$ is a selected subset of all cities.

A naive bruteforce strategy to optimize this objective function would be to try out all 2^n possible selections $\mathcal{S}$ of all cities. This quickly becomes computationally intractable.

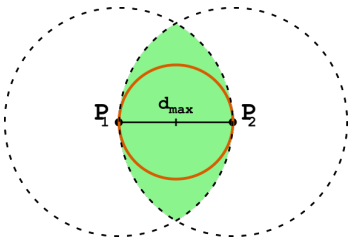

Fig. 1

Another bruteforce approach would be to iterate over all pairs of cities (in $O(n^2)$). For each pair of cities, we then assume that the maximum pairwise distance of the selected cities $\mathcal{S}$ is attained by this particular pair. In other words, we assume that given the cities with the maximum distance, there has to be a single optimal selection $\mathcal{S}$. We could then try to add further cities, as long as they do not change the maximum distance value. As seen in Fig 1, if p_1 and p_2 are the two points attaining the maximum distance, all further selected points need to be inside the green zone, while having a maximum distance between them smaller than the distance between p_1 and p_2. A reasonable way to do this would be to create a circle with the radius set to half the maximum distance between the originally selected pair of cities, centered at their midpoint, the red circle in Fig 1. All cities within that circle can then be added to the selection without increasing the maximum distance. This however is *not* optimal, as potentially some cities outside the circle could be added without increasing the maximum distance. In short, such a strategy would not reliably find the optimal solution, and it's complexity would still be on the order of $O(n^3)$.

Therefore, we use the Metropolis algorithm [1] to optimize the given objective function. Using Metropolis, we can solve this problem for very large numbers of cities (up to 100,000 without any issues). In the following, we first describe how a naive implementation of Metropolis would solve the problem. We then improve over this baseline by using the geometric properties of the objective function. We further make use of geometrical algorithms to reduce the computational complexity of our Metropolis-Hastings implementation.

2. Method

2.1. Problem statement

Given the non-convex, discrete nature of the optimization problem in Equation 2, Metropolis seems to be a good choice to optimize f. Our goal is to find the selection $\mathcal{S}^*$ of cities such that

$$\mathcal{S}^* = \arg\min_{\mathcal{S}} f(\lambda, \mathcal{S}) \quad (3)$$

The parameter λ is assumed to be fixed and is part of the problem specification. The Metropolis algorithm now works as follows: we start from an initial selection $\mathcal{S}$. In each iteration, we then randomly sample a new state $\hat{\mathcal{S}}$. The new state is then accepted with probablity:

$$a = \min\left[1, \exp(-\beta(f(\lambda, \hat{\mathcal{S}}) - f(\lambda, \mathcal{S})))\right] \quad (4)$$

where β a positive constant also called the *inverse temperature*. This acceptance probablity formulation implies that whenever the new loss value is lower than the previous one, the new state is accepted. If the new loss value is larger than the old one, the state might be accepted, but that depends on β and the amount the loss increases. A small value of β will make it more likely that non-improving states are accepted. Accepting states which do not decrease the loss function is important to escape local minima, but can also lead to a poor final solution if β is too small. If β is 0, the solution space is explored randomly

without considering the optimization objective at all. If β is infinite, we will only ever accept solutions which decrease the loss function. In the following we now describe different ways in which the Metropolis algorithm can be applied to our problem.

2.2. Strategies

2.2.1. Baseline. The simplest application of the Metropolis algorithm just randomly selects and deselects one city in each iteration. We start with no cities selected, i.e. $\mathcal{S} = \{\}$. In each optimization iteration, we uniformly sample an index $k \in \{1, ..., n\}$. If the city k is in the current selection $\mathcal{S}$, we remove it. If it's not in the current selection, we add it $\mathcal{S} = \mathcal{S} \cup \{k\}$. In each case we then accept this change with the acceptance probability. This is similar to the flipping strategy usually used in the Ising model spin assignment [2]. This simple strategy indeed works and finds some local optimum to f. However, we found that this strategy also has some limitations. By randomly adding and removing individual cities, the algorithm struggles to efficiently explore the whole space of possible solutions. Regardless of how β is set, it seems that the final result of this method highly depends on the random seed: indeed, once a lot of cities from the same area have been added, it will struggle to jump to a solution where the cities selected are in a completely different place. As n becomes larger (e.g. $n = 1000$), another issue occurs: if λ is small enough that the optimal solution should select a significant fraction of all cities, it takes a lot of steps to add them and this first strategy does not perform well.

2.2.2. Convex hull. The objective function that is used implies some very geometric constraints. Indeed, the maximum distance term requires the computation of the *diameter of a point set*. A naive computation of this diameter would be in $O(n^2)$, but it turns out that it can be easily computed using the convex hull, see [3]. The maximum pairwise distance of a set of points is equal to the maximum pairwise distance of vertices on their convex hull. This means in particular that at each step we can safely add all the cities that lie inside the convex hull of the already selected cities. This idea can be used to improve the complexity of our baseline algorithm, but it also leads to another strategy. In this one, at each step we chose to compute the objective function as if all the points contained in the convex hull of the current selection had been added, without truly adding them. This allows to more efficiently cover larger regions of the map, especially if the number of cities is large. Note that for all our methods, we add all the points in the convex hull at the end of the optimization, as this can only improve the result.

2.2.3. Clustering. As described, the drawback of the baseline method is that it is very inefficient when there are many cities. One solution is to approach the problem hierarchically. Clustering can be used: the optimization can be done over 10 mega-cities, then 100, ..., to N cities, using $\log_{10} N$ clustering steps. We implemented a very simple version of such an algorithm. This method first solves the optimization on the 10 most populated cities, then uses that result to initialize the solver for the 100 most populated cities, etc. This works reasonably well, but also has failure cases, as we will discuss later.

2.2.4. Continuous Markov Chain. We found that solution strategies based on directly modifying a discrete set of selected cities inherently struggle to use the geometrical properties of the problem. Once they selected a cluster of cities in one region, they tend to mostly stick to it and not be able to drastically change it. We noticed that the distance term in the loss function strongly favors solutions which select all cities in a mostly circular region. While there is no guarantee that the best solution lies in a circle, it intuitively makes sense to try different circular selections of cities. Therefore, we decided to build a solution which explores the solution space by running the Metropolis algorithm on the parameters of a circle. We then consider all the cities inside this circle as our current selection. Formally, the parameters of our Markov chain are now a circle center $c \in \mathbb{R}^2$ and radius $r > 0$. At each iteration, we either modify the radius or the circle center, with a 50% chance each. For the center, we compute the new center as $\hat{c} = c + u$, with $u \sim \mathcal{N}_2(0, 0.04)$. To improve global exploration, we occasionally do a larger mutation with $u \sim \mathcal{N}_2(0, 0.2)$. The probability for a large mutation is 0.05. If we decide to modify the radius, we compute the new radius as $\hat{r} = \max(0.01, r + v)$ with $v \sim \mathcal{N}(0, r/10)$. The change in the circle radius is proportional to the current radius. We found these parameters to work well, but we did not tune them systematically, as the method did not seem very sensitive to these. To compute the acceptance probablity, we then find all the cities inside the given circle and compute the original loss function using this set of current cities. Note that the proportional change of the radius makes the underlying Markov Chain non-symmetrical, but we did not find this to be an issue in practice. This continuous formulation reliably outperforms all our other strategies.

3. Implementation

We implemented our method and baselines using Python, relying heavily on Numpy [4] and Scipy [5] for efficiency. The objective function evaluation is completely vectorized, as our initial unvectorized implementation using Python for-loops was very slow (on the order of minutes of computation for $n = 100$ and 1000 iterations). Our final implementation runs on the order of seconds for small problem sizes ($n \leq 10,000$) and in around 30 seconds to 1 minute on $n = 100,000$ (measured on an AMD Ryzen 5 3600 CPU). The timings depend on the setting for λ. The lower λ, the more cities have to be selected and the more costly the iterations become. We run our method for 5000 to 10000 iterations, which seems to be sufficient for convergence. In the following we briefly discuss some of the implementation details.

3.1. Loss function

The run time of the algorithm is largely determined by the performance of evaluating the loss function f for the current state of the Markov chain. In each iteration of the algorithm, we need to evaluate the loss function to compute the acceptance probability. Computing the mutation itself is typically very cheap.

To lower the cost of evaluating the cost function, we only evaluate it once per iteration and keep track of the loss value of the previous state. Further, the main cost of evaluation the function f comes from the distance term. The sum of the values of the cities is trivial to compute in $O(n)$, and potentially could even be computed in $O(1)$ if the previous sum of v_i terms is stored after each iteration.

The costly term in the loss function is the maximum of the pairwise squared euclidean distances. A naive implementation of this computation has cost $O(n^2)$. When adding a single city to an existing selection, we could recompute the maximum distance in $O(n)$ by evaluating the distance of the newly added cities to all previous cities, if we assume that we know the previous maximum distance. However, we will not be able to do the same simplification if we decide to remove a city. Then we will again have to recompute all pairwise distances. Fortunately, the distance term can be computed as the maximum pairwise distance of the vertices of the convex hull of the current selection. The complexity of the convex hull computation is in $O(n \log n)$ or $O(n \log h)$, where h is the number of vertices in the convex hull. We use Scipy's implementation of the convex hull algorithm. Using this optimization gave a significant performance improvement compared to the naive maximum distance computation.

3.2. Cities inside a circle

When using our continuous formulation, we do not only need to evaluate the loss function, but we also need to determine which cities are inside the current circle. We do this using the KD-tree implementation in Scipy. The KD-tree allows for an efficient query of all points in a circle. We found this to improve performance for larger number of cities ($n > 10000$). For a small number of cities, a naive implementation is not much slower.

3.3. Beta Parameter

We found the process of tuning the parameter β to reach the best (fixed) parameter not effective. In practice there was not any significant difference over different choices of β and there was a large standard deviation in the final loss value when running for different initializations of a fixed β regime. We found that in practice increasing the β from small to large values during the optimization process is more effective, a process which is called *simulated annealing*. By starting with a small β, the algorithm visits all the states almost uniformly in the beginning. After a sufficient number of iterations, we can increase the β and run the algorithm again from the state found from the previous step. We can redo this process multiple times and gradually increase β until we are almost sure that we have reached the global minimum with high probability. Algorithm 1 shows the simulated annealing process we used in our experiments.

Algorithm 1 Simulated annealing

Input: total number of iterations n, current iteation i
if $i < \frac{n}{5}$ **then**
  $\beta = 1$
else if $i < \frac{2n}{5}$ **then**
  $\beta = 5$
else if $i < \frac{3n}{5}$ **then**
  $\beta = 10$
else if $i < \frac{4n}{5}$ **then**
  $\beta = 20$
else
  $\beta = 50$
end if
Return: β

3.4. Parallelization

Lastly, we found that even using our continuous implementation, different random seeds do not always converge to the same optimal result. It is therefore still advantageous to run the optimization several times with different random seeds. Among all these runs, we can then pick the result with the lowest error. In practice, we can easily parallelize the algorithm over different random initializations. This means that running different random seeds is not very costly, as most modern systems can run at least 8 threads simultaneously. We implemented this parallel execution using Python's *multiprocessing* module and built-in thread pool.

4. Results

4.1. Comparison between strategies

The baseline is quite slow and inefficient for large set of cities: in this case, the loss of the obtained solution is often greater than zero, meaning that it is worse than selecting a single city. This is improved by using en empty set as the initialization state and a high beta, but it is not enough. The *convex hull* method performs well in certain cases, in particular when the number of cities is large ($n \geq 1000$). The clustering methods can give pretty good solutions when a large number of cities are the solution (small λ), but at a cost of performance: our simple implementation does not really scale beyond 1000 cities. Overall, continuous Markov chain strategy provides the best results and scales up to 100,000 cities without any issues.

We ran experiments on two different distribution of the cities' populations and for different values of λ. We mainly

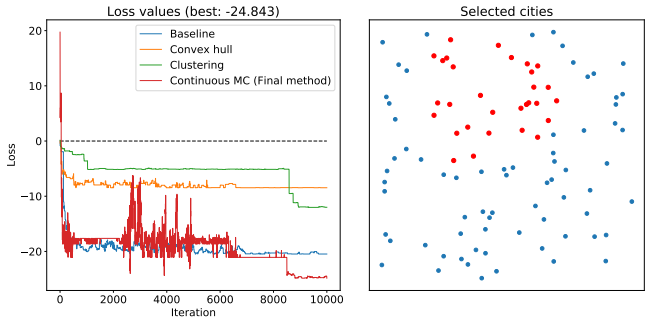

Fig. 2: Plots of optimization losses over iterations for all our techniques and visualization of the map of cities and the final solution found by our algorithm (selected cities in red)

used the uniform distribution $\mathcal{G}_1$:

$$\mathcal{G}_1^n : \{(v_i)_{1\leq i\leq n} \sim \mathcal{U}([0,1]), (x_i)_{1\leq i\leq n} \sim \mathcal{U}([0,1]^2)\}$$

and the log-normal distribution $\mathcal{G}_2$:

$$\mathcal{G}_2^n : \{(\ln v_i)_{1\leq i\leq n} \sim \mathcal{N}(-0.85, 1.3), (x_i)_{1\leq i\leq n} \sim \mathcal{U}([0,1]^2)\}$$

We ran the Metropolis algorithm with the strategies introduced in Section 2.2 to optimize the equation (2) for both of the distributions $\mathcal{G}_1$ and $\mathcal{G}_2$. Figure 2 shows the convergence plots for the different algorithms run on an example data set with distribution $\mathcal{G}_2$ and $n = 100$. Figures 3 and 4 show a comparison between our different strategies over different choices of λ for the objective function. We can observe that the continuous Markov Chain strategy clearly outperforms the other strategies and can reach lower values for the objective function.

4.2. Stability

Another favorable feature of the continuous Markov Chain strategy is that it is less susceptible to the random initialization compared to the other strategies. Figure 5 shows that the baseline strategy can perform drastically different depending on the random seed, whereas the continuous Markov Chain strategy is more stable and shows less deviation in the minimum loss value it can reach. Note that occasionally the baseline strategy will produce good results, but just not reliably.

5. Conclusion

We explored the Metropolis algorithm to optimize a complex geometric objective function which is highly non-convex. We found that the transition strategy between the states has a significant influence on the performance of the Metropolis algorithm on optimizing this objective function. Mainly we observed that for a large number of cities, the strategies which only add or remove one city in each iteration are slow and usually struggle to find an optimal set of cities which is far from the initial cities selected in the first iterations. In light of this observation, we proposed to use a continuous Markov

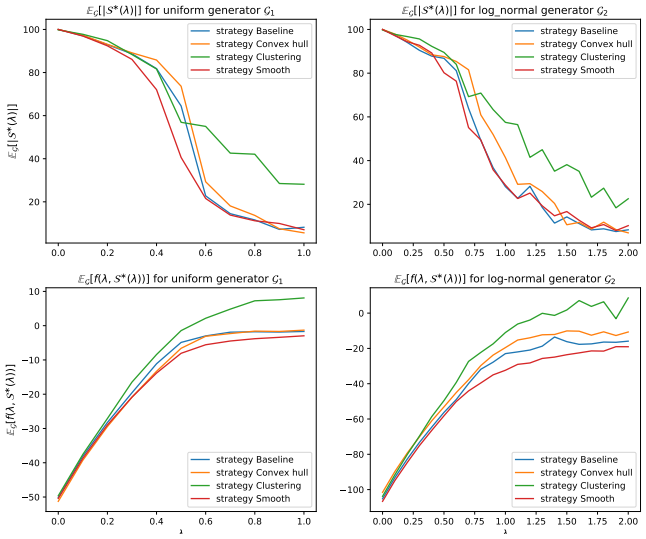

Fig. 3: For all our techniques, we plot the expected number of cities being selected for both data sets $\mathcal{G}_1$ and $\mathcal{G}_2$ (top) over different values of λ. We also plot the expected loss value as we change λ (bottom). All results have been averaged over 20 different random seeds. the total number of cities was 100 in this experiment. Our continuous Markov chain (here called "Smooth") consistently outperforms all our other implementations.

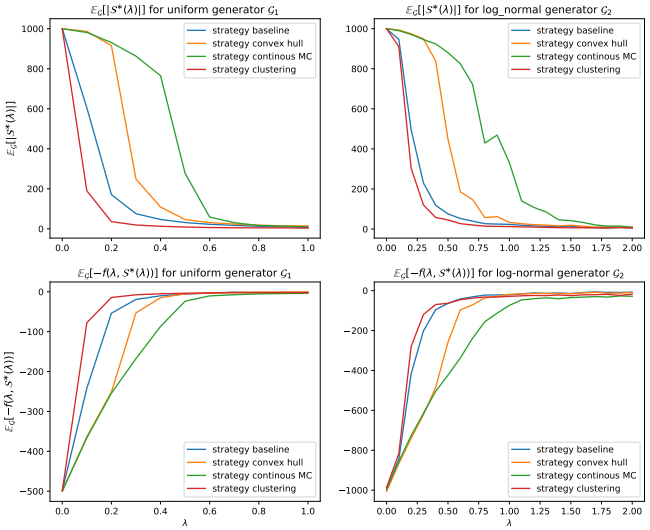

Fig. 4: We plot the number of selected cities and expected loss values also for $n = 1000$. Again, our continuous Markov chain outperforms the other techniques (the names and ordering of techniques are different from Figure 3 since we did not have time to regenerate this result.

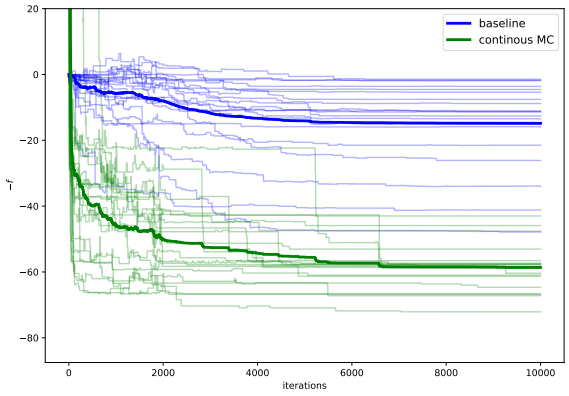

Fig. 5: Stability of the continuous Markov Chain strategy. The thin light-colored lines show each of the 20 runs of the algorithm with different initializations. The bold dark-colored lines show the average over all runs.

Chain to move between the states as our best strategy. We claim that this strategy helps to explore the whole map of cities more efficiently to find the optimal set of cities. The solution set derived by this method are always circular, but in practice they can be used as an initialization to the baseline method to find a more optimal solution. Overall, we believe that our method strikes a good balance between quality of results and computation time.

References

[1] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *The Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087–1092, 1953. [Online]. Available: https://doi.org/10.1063/1.1699114

[2] M. Newman and G. Barkema, *Monte carlo methods in statistical physics chapter 1-4*. Oxford University Press: New York, USA, 1999, vol. 24.

[3] M. Suderman. (2000) Diameter algorithm based on convex hull. [Online]. Available: http://cgm.cs.mcgill.ca/~athens/cs507/Projects/2000/MS/diameter/node2.html

[4] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del R'ıo, M. Wiebe, P. Peterson, P. G'erard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[5] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–. [Online]. Available: http://www.scipy.org/