Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Matematyczny
specjalność: Analiza danych

Bartosz Chmiela

Locally-informed proposals in Metropolis-Hastings algorithm with applications

Praca magisterska
napisana pod kierunkiem
dr. hab. Pawła Lorka

Wrocław 2022

Abstract

The Markov Chain Monte Carlo methods (abbrv. MCMC) are a family of algorithms used for sampling from a given probability distribution. They prove very effective when the state space is large. This fact can be used to solve many hard deterministic problems – one of them being *traveling salesmen problem*. It will be used in this thesis to test a new approach of *locally-informed propolsals* as a modification of well known *Metropolis-Hastings* algorithm. In this thesis we will present the implementation of modified algorithm, experiments based on it, results and a comparison of to previous MCMC methods.

Metody próbkowania Monte Carlo łańcuchami Markowa są rodziną algorytmow używanych do próbkowania z danego rozkładu prawdopodobieństwa. Okazują się efektywne zwłaszcza gdy przestrzeń stańw jest wielka. Ten fakt może być wykorzystany przy rozwiązywaniu wielu deterministycznych problemów – jednym z nich jest *problem komiwojażera*. Zostanie on użyty w tej pracy do przetestowania nowego podejścia *lokalnie poinformowawnego?*, jako modyfikacji dobrze znanego algorytmu *Metropolisa-Hastinigsa*. W tej pracy zaprezentujemy implementacje zmodyfikowanego algorytmu, eksperymentów bezujących na nim, wyników oraz porównania z poprzednimi metodami próbkowania Monte Carlo.

Contents

List of Tables

List of Figures

1 Introduction

The Markov Chain Monte Carlo methods (abbrv. MCMC) are a family of algorithms used for approximate sampling from a given probability distribution. At first they do not seem useful for solving practical deterministic problems, but with some tweaks they can become a powerful tool. It happens especially when space of possible solutions is enormous and computing becomes infeasible for machines. These offer a shortcut for obtaining "close enough" answers.

At their core, MCMC methods generate a Markov Chain (abbrv. MC) with a defined distribution and sample using it. The convergence of the chain is assured by ergodic theorems. One of the most known of them is *Metropolis-Hastings* algorithm, which constructs a MC using another set of distributions, maybe simpler ones.

In this thesis we work on *locally-informed proposals*, which involve determining *local* distribution – which comes down to finding transition probabilities of the state. They are a bit more complex and computationally heavy, but offer better results with less iterations.

To test this method we will need a deterministic problem which quickly becomes infeasible for machines to compute – one of them is a well-known traveling salesman problem. The testing is carried out using its benchmark training set *tsplib95* and implementation is provided in *Python3*.

2 Markov chains

Markov chains are the very basic building blocks of the theory used within this thesis. They are a natural extension of independent stochastic processes, that assume a weak dependence between the presence and the past.

In this thesis we will focus only on stochastic processes with discrete time steps and finite state space, which satisfies the Markov property. These are the ones that, we are able to simulate in computers.

2.1 Basic terminology and assumptions

We assume that the reader has a basic probabilistic background, so that we can freely use terminology from probability theory, like random or independent variables, stochastic processes, measure or σ-algebra.

A Markov chain needs to be defined with discrete state space and index set.

Definition 2.1. *A state space of a Markov chain is a countable set S.*

A state space defines values over which a Markov chain is iterating. In our case it is finite so we can associate it with a subset of natural numbers like $\{1, 2, \ldots, N\}$ (for some N depending on a number of states) instead of states.

Definition 2.2. *An index set of a Markov chain is a countable set T.*

An index set represents time in which Markov chain moves. For a chain we assume discrete time steps and again in our case it will be a finite set, so we can associate it with a subset of natural numbers like $\{1, 2, \ldots, T\}$ for some T depending on a length of time interval.

To work with any probabilistic construct such as Markov chain, we need a probability space in which it resides and can be measured.

Definition 2.3. *A probability space is a triplet* $(\Omega, \mathcal{F}, P)$*, where* Ω *is some abstract sample space,* $\mathcal{F}$ *is a* σ*-algebra (event space) and* P *is a probability measure.*

In our case the sample space is a common space for random variables $X_k,\ k = 0, 1, \ldots$ and is a space of all possible states. The event space is a space of all possible events given our states. Probability measure is not explicitly given, because it is often not easy to find a probability of a random variable.

Most of the time a Markov chain will be associated with a stochastic transition matrix $\mathbf{P}$, that represents probabilities of transitions between states.

Definition 2.4. *A stochastic matrix* $\mathbf{P}$ *is a matrix with non-negative entries, which rows sum to* 1*.*

The number of entries of a transition matrix grows quadratically with number of states, so it quickly becomes a large object, not possible to fit into the memory of a computer. Now with the notation and the background we are able to define a Markov chain.

2.2 Definition and basic properties

In this subsection we will formally define a Markov chain and list some of its basic properties.

Definition 2.5. *A Markov chain (abbrv. MC) is a sequence of random variables* $(X_n)_{n\in T}$ *defined on a common probability space* $(\Omega, \mathcal{F}, P)$*, that take values in* S*, such that it satisfies Markov property:*

$$P(X_{n+m} = j | X_n = i, X_{l_{k-1}} = i_{l_{k-1}}, X_{l_{k-2}} = i_{l_{k-2}}, \ldots, X_{l_1} = i_1) = P(X_{n+m} = j | X_n = i).$$

For all indices $l_1 < \ldots < l_{k-1} < n < n+m,\ 1 \leq k < n$ *and all states* $j, i, i_{n-1}, i_{n-2}, \ldots, i_0 \in S$*.*

This definition indicates independence of the past of a MC. The probabilities depend on current state and the number of steps.

Definition 2.6. *A Markov chain is homogeneous if additionally:*

$$P(X_{n+m} = j | X_m = i) = P(X_n = j | X_0 = i).$$

In this case we define:

$$p_{i,j}(n) \stackrel{df}{=} P(X_n = j | X_0 = i).$$

Homogeneous Markov chains (abbrv. HMC) are more natural for us and easier to study. These eliminate the dependence on the number of steps that a Markov chain went through. From now on, whenever we use a term *markov chain* we will mean a *homogeneous markov chain.*

The probabilities $p_{i,j}(n)$ give us a probability of transition between states i and j in n steps. We can use them to form a special matrix that will be linked with a MC.

Definition 2.7. *A transition matrix in n steps $\mathbf{P}(n)$ for a MC is a stochastic matrix constructed using transition probabilities:*

$$\mathbf{P}_{i,j}(n) = p_{i,j}(n),\ \mathbf{P}_{i,j}(0) = \mathbb{I},\ \mathbf{P} \stackrel{df}{=} \mathbf{P}_{i,j}(1).$$

Definition 2.8. *A initial distribution of a Markov chain is a vector $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_N)^T \in \mathbb{R}^N$ such that $\sum_{i=1}^{N} \mu_i = 1$. This is a distribution of a random variable X_0, which is an initial state of a Markov chain.*

A transition matrix together with initial distribution define a MC, so these are the only objects that need to be analyzed if one wants to study those chains.

Theorem 2.1. *Let $\boldsymbol{\mu}^{(n)} \in \mathbb{R}^N$ be a distribution of a MC at n-th step and $\boldsymbol{\mu} \in \mathbb{R}^N$ the initial distribution, then for all n:*

$$\boldsymbol{\mu}^{(n)} = \boldsymbol{\mu}\mathbf{P}^n.$$

Proof. content...? □

Proof of this theorem involves unfolding vectorized equation and using basic induction so it will be left. It also shows that given some knowledge of matrix $\mathbf{P}$ one can easily work with a MC.

2.2.1 Irreducibility

Irreducibility guarantees that all states of a MC have the same properties, so that we do not need to analyze every state separately. Moreover an irreducible MC cannot be split into more chains.

Definition 2.9 (Irreducibility)**.** *A Markov chain with transition matrix $\mathbf{P}$ is called irreducible if and only if for every pair of states i and j there exists a positive probability of transition between them i.e.,*

$$\exists\, n\, \mathbf{P}_{i,j}(n) > 0.$$

2.2.2 Periodicity

Periodicity tells us something about the structure of the transition matrix. It especially indicates when there is a possibility of a chain staying in one state.

Definition 2.10 (Periodicity)**.** *Let d_i be a greatest common divisor of those n such that $\mathbf{P}_{i,i}(n) > 0$ i.e.,*

$$d_i = gcd\{n \geq: \mathbf{P}_{i,i}(n) > 0\}$$

If $d_i > 1$ then state i is periodic. If $d_i = 1$ then state i is aperiodic.

Definition 2.11. *A Markov chain with transition matrix $\mathbf{P}$ is called periodic with a period d when all states are periodic with a period d. In particular, when MC is irreducible and there is a state with a period d, then all the states are with period d and chain is periodic.*

2.3 Stationarity and ergodicity

In this subsection we will cover asymptotics for the long-term behavior of a MC.

Definition 2.12 (Stationarity). *A probability distribution* $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_N)$ *is called stationary if it satisfies*

$$\pi_j = \sum_{i \in S} \pi_i p_{ij},$$

or equivalently in vector form:

$$\boldsymbol{\pi} = \boldsymbol{\pi}\mathbf{P}.$$

This equation is often described as the balance equation.

From this definition it is easy to see that if a MC gets to stationary distribution it will not leave it.

Definition 2.13 (Ergodicity). *A Markov chain is ergodic when it is irreducible and aperiodic.*

Theorem 2.2. *For any ergodic Markov chain, there exist at least one stationary distribution.*

The proof of the theorem 2.2 can be found in [3].

Theorem 2.3. *Let* (X_n) *be a ergodic Markov chain, then:*

$$\lim_{n \to \infty} p_{ij}(n) = \pi_j.$$

Proof. content... □

Those theorems tell us for some MC we are able to predict their behaviors after enough time passes. It also gives us a tool to sample from a given distribution $\boldsymbol{\pi}$ using Markov chains.

2.4 Reversibility

A reversible MC has a property of having the same distribution in the past and in the future.

Definition 2.14 (Reversibility). *A Markov chain* (X_n) *is reversible if random vectors*

$$(X_{i_1}, X_{i_2}, \ldots, X_{i_k}) \text{ and } (X_{m-i_1}, X_{m-i_2}, \ldots, X_{m-i_k}),$$

have the same distribution for all m *and* $i_j, m - i_j \in T$.

This condition is just a mathematical representation of the previous statement and it is not practical, so we need another way of describing it.

Definition 2.15 (Detailed balance). *A probability distribution* $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_N)$ *satisfies detailed balance for a Markov chain with transition matrix* $\mathbf{P}$ *if*

$$\forall\, i, j \in S \; \pi_i p_{i,j} = \pi_j p_{j,i}$$

Definition 2.16. *A Markov chain is said to be reversible if there exists a reversible distribution for it.*

This definition is much more practical, because the condition can be calculated for each proposed distribution.

Theorem 2.4. *If a probability distribution $\boldsymbol{\pi}$ satisfies detailed balance for a Markov chain with transition matrix $\mathbf{P}$, then $\boldsymbol{\pi}$ is a stationary distribution for this chain.*

Proof. Summing over all $i \in S$ we get

$$\sum_{i \in S} \pi_i p_{i,j} = \pi_j \sum_{i \in S} p_{j,i} = \pi_j,$$

which is a balance equation. □

The detailed balance will be often used in as it is somewhat easier to check than balance equation.

3 Markov chain Monte Carlo methods

In this section we will show how to use aforementioned properties of Markov chains for solving problems. The idea is to use a Markov chain to simulate complicated models and estimate relevant parameters.

The classical example is measuring the area under a curve or a figure, with counting how many randomly generated points are inside or outside the figure and estimating area as a mean. Such a sequence of points is a sequence of random independent variables, which is also a MC, but without any dependence.

If we could generate a MC with a stationary distribution proportional to some function of our interest we could find its maximum, by counting frequencies of states. That is the core idea of Metropolis algorithm.

3.1 Metropolis-Hastings algorithm

We seek an algorithm constructing a MC, which has a stationary distribution of our given probability distribution $\boldsymbol{\pi}$ ($\pi_i > 0$). One could of course find such a transition matrix $\mathbf{P}$ that has stationary distribution $\boldsymbol{\pi}$ but this does not avoid the problem of enormous state space – the matrix $\mathbf{P}$ would also be enormous. So it would be a feasible idea, when the state space is small and also deterministic algorithms are able to find solutions.

Regardless of this fact let us start with constructing such a matrix. Assume that we have another stochastic matrix $\mathbf{Q}$ which is irreducible and aperiodic. Let us consider a matrix defined us:

$$\mathbf{P}_{i,j} = \begin{cases} \mathbf{Q}_{i,j} \min\left(1, \frac{\pi_j \mathbf{Q}_{j,i}}{\pi_i \mathbf{Q}_{i,j}}\right) & \text{if } i \neq j, \\ 1 - \sum_{j \in S \setminus \{i\}} \mathbf{P}_{i,j} & \text{if } i = j. \end{cases} \tag{3.1.1}$$

Theorem 3.1. *A matrix defined in 3.1.1 is stochastic, irreducible, aperiodic and has a stationary distribution $\boldsymbol{\pi}$.*

Proof. The matrix $\mathbf{P}$ is stochastic from a definition – one entry in a row is just a sum of every other and subtracted from 1. For $\mathbf{Q}_{i,j} > 0$ we have $\mathbf{P}_{i,j} > 0$ and irreducibility and aperiodicity are inherited from $\mathbf{Q}$. Let us look at the detailed balance:

$$\pi_j \mathbf{P}_{j,i} = \pi_j \mathbf{Q}_{j,i} \min\left(1, \frac{\pi_i \mathbf{Q}_{i,j}}{\pi_j \mathbf{Q}_{j,i}}\right) = \min\left(\pi_i \mathbf{Q}_{i,j}, \pi_j \mathbf{Q}_{j,i}\right) = \pi_i \mathbf{Q}_{i,j} \min\left(1, \frac{\pi_j \mathbf{Q}_{j,i}}{\pi_i \mathbf{Q}_{i,j}}\right) = \pi_i \mathbf{P}_{i,j}.$$

We see that this matrix satisfies detailed balance, co the distribution $\boldsymbol{\pi}$ is stationary. □

The matrix $\mathbf{Q}$ is called *candidate matrix* as it will be later proposing candidates for a MC. It is kind of a parameter for our later algorithms – a well chosen matrix will give better or worse results. We proved more general case, but if matrix $\mathbf{Q}$ is symmetric some terms cancel and the proof becomes easier.

The *Metropolis-Hastings algorithm* (abbrv. M-H) utilizes this construction to generate irreducible, aperiodic MC with a stationary distribution $\boldsymbol{\pi}$. When candidate matrix $\mathbf{Q}$ is symmetric we call this a *Metropolis algorithm*.

Algorithm 1 Metropolis-Hastings algorithm

Choose a state $i \in S$.
$X_0 \leftarrow i$
for $n = 1, 2, \ldots$ **do**
    Sample $j \sim \mathbf{Q}_i = (\mathbf{Q}_{i,1}, \mathbf{Q}_{i,2}, \ldots, \mathbf{Q}_{i,N})$.
    Sample $U \sim Unif(0, 1)$.
    if $U \leq \min\left(1, \frac{\pi_j \mathbf{Q}_{j,i}}{\pi_i \mathbf{Q}_{i,j}}\right)$ **then**
        $X_{n+1} \leftarrow j$
    else
        $X_{n+1} \leftarrow X_n$
    end if
end for

In reality one does not need to create whole candidate matrix or a transition matrix. We just need to know how to sample at one step, so how to choose a candidate given a current state. If the procedure is symmetric it simplifies drastically, we need only to know the quotient of distribution $\boldsymbol{\pi}$ at this step. The next constructions use this as an advantage to reduce number of computations.

4 Traveling salesman problem

Traveling salesman problem (abbrv. TSP) is a well known for being hard to solve and this is why many researchers, including us, use it as a benchmark problem for testing new methods. It is an old problem, with no solution, only with methods that try to achieve the best answer. It is proved to be a NP-hard problem, so deterministic algorithms cannot be reasonably used. This is why probabilistic methods like MCMC become interesting as they eliminate the need of computing all the steps or states of the problem.

TSP asks to find a shortest (the least costly) path between the vertices of a given graph, that covers all of them and is a cycle. This question becomes harder to answer with more vertices added to a graph. It started as a problem of salesman visiting all of the cities and coming back to his place.

4.1 Statement of the problem

To state this problem we need to define weighted graphs and paths as they can represent the problem.

Definition 4.1. *A undirected graph G is a pair (V, E), where V is a set of vertices and $E \subseteq V \times V$ is a set of edges.*

Definition 4.2. *A weighted undirected graph $G = (V, E)$ is a graph such that each edge e has assigned a weight $w_e \in \mathbb{R} \cup \{\infty\}$.*

Vertices could be any set, so we can think of them as a set of all cities. An edge is a pair of vertices, so it can be a connection between cities. A weight is just a function on edges, so it could be a distance between cities.

Again, because we have finite number of cities, we can work on set of indices instead. We will associate weight $w_{i,j}$ with the weight of an edge between vertices i and j.

Definition 4.3. *A path is a sequence of edges, in which all edges (and vertices joining them) are distinct.*

Definition 4.4. *A cycle is a path $c = (e_1, e_2, \ldots, e_n)$ such that only first and last edge are equal ($e_1 = e_n$).*

Definition 4.5. *A Hamiltonian cycle is a cycle that visits each vertex exactly once.*

Now we can express salesman tour as a Hamiltonian cycle that visits a city once. Such a cycle can be thought as a permutation of vertices.

Definition 4.6. *A tour is a Hamilitonian cycle and we identify it with a permutation of vertices.*

Definition 4.7 (Traveling salesman problem)**.** *Given an undirected weighted graph $G = (V, E)$, $|V| = n$ find a permutation $\sigma_{\min}$ of vertices such that*

$$\sigma_{\min} = \arg\min_{\sigma \in S_n} \sum_{i=1}^{n-1} w_{\sigma(i),\sigma(j)} + w_{\sigma(n),\sigma(1)}.$$

Where S_n is a set of all permutations of vertices.

It definition exactly states the Traveling salesman problem: visit all cities once, with the least distance covered.

4.2 Complexity

At first glance, one might not think of this as a hard problem, but to understand complexity of that, it is enough to think of all the permutation of vertices. The set of n vertices S_n has $n!$ elements, a number which grows rapidly. It means that if we want to check all possible salesman tours, we need to compute distances at most $n!$ times, which becomes infeasible with only 20 cities.

Depending on a method of calculating the distance we obtain complexity $O(n! \cdot d(n))$ where $d(n)$ is a number of steps needed to calculate distance of one path. If one just adds all weights then the complexity will be of $O(n! \cdot n)$, which is a lot more than a polynomial complexity and quickly becomes impossible to compute.

There are other methods like dynamic programming – Held-Karp algorithm, but the complexity $O(n^2 \cdot 2^n)$ is still a lot. The problem has been shown to be NP-hard even with removing some of the contradicts or using easier metrics.

4.3 Dataset

To test our methods we have obtained data from *TSPLIB*([2]) a site, which is a library of sample instances for TSP (not only that) from various sources and types. All the files there are of the extension *.tsp* (or alternatively *.xml*) and of following structure: *nameN.tsp*. *name* defines where does the data come from and N defines how many vertices there are. For handling this extension we use *tsplib95* package in *Python3*

All of the datasets there have an optimal solution, so we are able to compere our solutions. We have chosen only some of them:

- *berlin52* 52 locations in Berlin, with an optimal solution: 7542,
- *kroA150* 150-city problem A, with an optimal solution: 26524,
- *att532* 532 AT&T switch locations in the USA, with an optimal solution: 27686,
- *dsj1000* clustered random problem, with an optimal solution: 18659688.

5 Simulations

In the precious section we have seen that a deterministic approach of computing salesman tours is infeasible, so we can turn now to probabilistic methods of MCMC. Instead of checking all possible tours, now we want sample from this space, so the question is, how do we define a distribution there? It is usually done with a *softmax*, a transformation of our target function, which is in this case a distance of a tour.

Definition 5.1. *For a given vector* $\boldsymbol{x} = (x_1, x_2, \ldots, x_d)^T \in \mathbb{R}^d$ *a softmax function* $s : \mathbb{R}^d \to [0,1]^d$ *is defined as*

$$s(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{d} e^{x_j}},$$
$$s(\boldsymbol{x}) = (s(\boldsymbol{x})_1, s(\boldsymbol{x})_2, \ldots, s(\boldsymbol{x})_d)\,.$$

Such defined function is a probability distribution and it does not change the order of a given vector. This is important for us, because we can apply a softmax to the distances (weights) of the tours, so that we change them into distribution, but we will not change the relation between tours – ones with longer distances will have greater probability. As we are interested in shortest distances, we can add a minus sign to give them the greatest probability.

Let us get back to the notation of weights and use a vector $\boldsymbol{w} = (w_1, w_2, \ldots, w_N)^T$ representing distances of all possible tours. As we noticed earlier, this space is enormous, so calculating softmax of this vector $s(\boldsymbol{w})$ is again not feasible, because of enormous sum in the denominator $\sum_{j=1}^{N} e^{w_j}$. This is where Metropolis-Hastings algorithm comes in handy – we do not need actually probabilities, we just need their quotients, and this is where the sums will erase each other. What we actually need is something proportional to the probability:

$$\pi_i \propto e^{-w_i}.$$

Using softmax is even more beneficial, because of the properties of exponential function. In each step of M-H algorithm we need to compute:

$$\frac{\pi_j \mathbf{Q}_{j,i}}{\pi_i \mathbf{Q}_{i,j}} = e^{-(w_j - w_i)} \cdot \frac{\mathbf{Q}_{j,i}}{\mathbf{Q}_{i,j}}.$$

If candidate matrix $\mathbf{Q}$ is symmetric this equation simplifies further. In practice we will be using logarithms of quotients in M-H algorithm, because multiplying numbers from $(0, 1)$ get small quickly, so that they get out of the range of computer abilities. After applying logarithm we have:

$$\log\left(\frac{\pi_j \mathbf{Q}_{j,i}}{\pi_i \mathbf{Q}_{i,j}}\right) = -(w_j - w_i) + \log(\mathbf{Q}_{j,i}) - \log(\mathbf{Q}_{i,j}).$$

Again, if candidate matrix $\mathbf{Q}$ is symmetric we are left only with evaluating $w_j - w_i$ which is simple and it does not involve computing any other weights.

Now that we have a target distribution we can try sampling using a MC produced by M-H algorithm and after some point we should reach the tour with highest probability, *i.e.,* smallest distance covered. To do that we need to define candidate matrix $\mathbf{Q}$. The better the candidates, the faster MC can get to the minimum.

5.1 Candidates

Let us start with considering what the candidate is. We know that the space of all possible tours is enormous, so we cannot take every tour into consideration. Instead we focus on, what we will call from now on *neighbours*.

Definition 5.2. *A neighbour σ' of a permutation σ is a permutation, that for some k, l it satisfies $\sigma'(k) = \sigma(l)$, $\sigma'(l) = \sigma(k)$ and $\sigma'(i) = \sigma(i)$ for the rest of indices.*

These neighbours are the original tour with two swaped indices. This let's us consider a smaller space – there are $\binom{n}{2} = \frac{n(n-1)}{2} \approx n^2$ neighbours if the number of vertices is n.

5.2 Random neighbours

One approach is to sample neighbours uniformly, it has a benefit of being simple to understand and implement. It was already successfully presented in [4]. Sampling them uniformly is equivalent to choosing a random indices to swap, so it can be done efficiently. This way we do not need to create a candidate matrix, we just use a simple procedure. Most of the entries in this matrix would be 0, because we choose only some subset of all possible permutations to be our neighbours. It is a symmetrical procedure (every neighbour has the same probability), so the step in M-H algorithm simplifies further.

5.2.1 Computational considerations

For this algorithm to be truly efficient, we need to find some other way to calculate distance (weight) of a tour, as it can also contain many edges. Without any optimizations for each tour σ_i we need to calculate a sum:

$$w_i = \sum_{i=1}^{n-1} w_{\sigma(i),\sigma(j)} + w_{\sigma(n),\sigma(1)}.$$

What was observed in [4] is that the weight does not change drastically when changing a tour to its neighbour. It is because most of the edges stay the same. When given a tour

σ and its neighbour σ' they differ only on those edges where swap is happening, let's say k, l. So for this situation we have tours (permutations of vertices):

$$\sigma = (1, 2, 3, \ldots, k-1, k, k+1, \ldots, l-1, l, l+1, \ldots, N)$$
$$\sigma' = (1, 2, 3, \ldots, k-1, l, k+1, \ldots, l-1, k, l+1, \ldots, N)$$

Assuming that we know the sum of weights for tour σ to obtain the new one for neighbour σ' we need to remove from it weights $w_{k-1,k}$, $w_{k,k+1}$, $w_{l-1,l}$, $w_{l,l+1}$ and add $w_{k-1,l}, w_{l,k+1}, w_{l-1,k}, w_{k,l+1}$. This is only 8 operations per neighbour, a constant complexity cost (if we can get instantly weights).

5.2.2 Implementation

To sum up all the information we present the algorithm for this method (2). As mentioned before, we will be working on logarithms, because they are better suited for computer computations.

Algorithm 2 Random neighbours algorithm

Choose a tour $\sigma \in S_n$.
$X_0 \leftarrow \sigma$
Compute weight w_σ.
for $i = 1, 2, \ldots$ **do**
    Sample $k, l \sim Unif\{1, 2, \ldots, n\}$ without replacement.
    Sample $U \sim Unif(0, 1)$.
    $w_{\sigma'} \leftarrow w_\sigma - (w_{k-1,k} + w_{k,k+1} + w_{l-1,l} + w_{l,l+1}) + (w_{k-1,l} + w_{l,k+1} + w_{l-1,k} + w_{k,l+1})$
    if $\log(U) \leq \min(0, -(w_{\sigma'} - w_\sigma))$ **then**
        $X_{i+1}(k), X_{i+1}(l) \leftarrow X_{i+1}(l), X_{i+1}(k)$
        $w_\sigma \leftarrow w_{\sigma'}$
    else
        $X_{i+1} \leftarrow X_i$
    end if
end for

5.3 Locally-informed proposals

Locally-informed proposals are more complicated family of methods, they include more computational labor. The approach with uniform distribution of candidates is less complex, but forces us to make a lot of iterations. It is because choosing neighbours randomly conveys no information, so it is required for us to check a lot of neighbours until we find a better one.

This time, we want to compute a *local* distribution of neighbours and sample them from it. It has to be done efficiently too, because number of neighbours grows quadratically with the number of vertices (so for *dsj1000* its around 1 million neighbours).

5.3.1 Computational considerations

The observation here is similar to the previous one – when we have weights for all neighbours of a starting tour, we can update them by a common factor, only some of

them change in a different way. Then we can calculate softmax on neighbour weights. One might notice now, that this procedure is not symmetrical, because softmax can be different and the sums dividing exponent of weights will be different each time. Most notably, when a neighbour will have a high transition probability, getting back will have lower probability, as it means that there is more distance. This means that we will be using M-H algorithm (not only Metropolis).

Let us show this on an example: assume that we have chosen tour σ and its neighbour σ' that is connected with swapping indices k and l. So for this situation we have tours:

$$\sigma = (1, 2, 3, \ldots, k-1, k, k+1, \ldots, l-1, l, l+1, \ldots, N)$$
$$\sigma' = (1, 2, 3, \ldots, k-1, l, k+1, \ldots, l-1, k, l+1, \ldots, N)$$

Now we need to think of a neighbour connected to some other swap for both of those tours, let's say m and n. When m and n are far away from k and l we have almost the same tours, but with a swap on k and l. So the neighbours σ_{mn}, σ'_{mn} of σ and σ' respectively look like:

$$\begin{aligned}\sigma_{mn} =&(\ldots, (m-1), n, (m+1), \ldots, (n-1), m, (n+1), \ldots,\\ &\ldots, (k-1), k, (k+1), \ldots, (l-1), l, (l+1), \ldots)\\ \sigma'_{mn} =&(\ldots, (m-1), n, (m+1), \ldots, (n-1), m, (n+1), \ldots,\\ &\ldots (k-1), l, (k+1), \ldots, (l-1), k, (l+1), \ldots)\end{aligned}$$

Again like before, assuming we know the weight of σ_{mn} we can remove weights $w_{k-1,k}$, $w_{k,k+1}$, $w_{l-1,l}$, $w_{l,l+1}$ and add $w_{k-1,l}, w_{l,k+1}, w_{l-1,k}, w_{k,l+1}$. This case covers most of the neighbours.

Let us think now about the case, when swap of m and n indices happen somewhere close to k and l, for example $k = m - 1$:

$$\begin{aligned}\sigma_{mn} =&(\ldots, (k-1), k, n, (m+1), \ldots,\\ &\ldots, (n-1), m, (n+1), \ldots, (l-1), l, (l+1), \ldots)\\ \sigma'_{mn} =&(\ldots, (k-1), l, n, (m+1), \ldots,\\ &\ldots, (n-1), m, (n+1), \ldots, (l-1), k, (l+1), \ldots)\end{aligned}$$

This time we have more edges to remove and add. We need to remove: $w_{k-1,k}$, $w_{k,n}$, $w_{l-1,l}$, $w_{l,l+1}$ and add: $w_{k-1,l}, w_{l,n}, w_{l-1,k}, w_{k,l+1}$. One might notice that these are not the same weights as in a previous paragraph. It means, that some neighbours (swaps) have to be considered separately. These are the scenarios when m or n are elements of $\{(k-1), k, (k+1), (l-1), l, (l+1)\}$.

5.3.2 Implementation

To sum up all the information we present the algorithm for this method (3). As mentioned before, we will be working on logarithms, because they are better suited for computer computations.

The core of this algorithm is still the same, but we need to add more complicated part with computing neighbour weights and updating them after each step of a main loop. To not blur the algorithm we have used a term *difference in weights* which would be a sub-algorithm, that would compute all the weights to remove and add.

Algorithm 3 Locally-informed proposals algorithm

Choose a tour $\sigma \in S_n$.
$X_0 \leftarrow \sigma$
Compute weight w_σ.
Compute all neighbour weights $\boldsymbol{w}_\sigma$.
for $i = 1, 2, \ldots$ **do**
    $s(\boldsymbol{w}_\sigma) = \text{softmax}(\boldsymbol{w}_\sigma)$
    Sample $\sigma' \sim s(\boldsymbol{w}_\sigma)$.
    Find k, l connected with swapping.
    $C \leftarrow -(w_{k-1,k} + w_{k,k+1} + w_{l-1,l} + w_{l,l+1}) + (w_{k-1,l} + w_{l,k+1} + w_{l-1,k} + w_{k,l+1})$
    $w_{\sigma'} \leftarrow w_\sigma + C$
    $\boldsymbol{w}_{\sigma'} \leftarrow \boldsymbol{w}_\sigma + C$
    for $m = 1, 2, \ldots, N$ **do**
        for $n = m + 1, \ldots, N$ **do**
            if $m \vee n$ in $\{(k-1), k, (k+1), (l-1), l, (l+1)\}$ **then**
                $\boldsymbol{w}_{\sigma'}[(m, n)] \leftarrow \boldsymbol{w}_{\sigma'}[(m, n)] - C$ ▷ neighbour with a swap of m, n.
                $\boldsymbol{w}_{\sigma'}[(m, n)] \leftarrow \boldsymbol{w}_{\sigma'}[(m, n)]+$ difference in weights
            end if
        end for
    end for
    $s(\boldsymbol{w}_{\sigma'}) = \text{softmax}(\boldsymbol{w}_{\sigma'})$
    Sample $U \sim Unif(0, 1)$.
    if $\log(U) \leq \min(0, -(w_{\sigma'} - w_\sigma) + \log(s(\boldsymbol{w}_{\sigma'})[(k.l)]) - \log(s(\boldsymbol{w}_\sigma)[(k, l)]))$ **then**
        $X_{i+1}(k), X_{i+1}(l) \leftarrow X_{i+1}(l), X_{i+1}(k)$
        $w_\sigma \leftarrow w_{\sigma'}$
        $\boldsymbol{w}_\sigma \leftarrow \boldsymbol{w}_{\sigma'}$
    else
        $X_{i+1} \leftarrow X_i$
    end if
end for

Unfortunately, there are more practical things to consider this time. In this algorithm we are actually calculating softmax function, nothing is vanishing here. Computing exponents of large numbers ends up with ∞ or $-\infty$ on a computer. Also, for datasets like *dsj1000* there are around 1 million neighbours, and we need to keep probabilities with higher precision, that is a lot of memory to use.

We can alleviate this problem (for some cases) by working on differences in weights, not weights themselves, as they can be large as well. Let's say we know weight of our current tour w_σ and its neighbours weights vector $\boldsymbol{w}_\sigma$. We can think of every weight in this vector as a difference between it and current tour weight: $\boldsymbol{w}_\sigma[i] = w_\sigma + d_i$. Computing softmax then

$$s(\boldsymbol{x})_i = \frac{e^{\boldsymbol{w}_\sigma[i]}}{\sum_{j=1}^{n(n-1)/2} e^{\boldsymbol{w}_\sigma[j]}} = \frac{e^{w_\sigma + d_i}}{\sum_{j=1}^{n(n-1)/2} e^{w_\sigma + d_j}} = \frac{e^{d_i}}{\sum_{j=1}^{n(n-1)/2} e^{d_j}}.$$

So the probabilities are the same when working on differences, which are smaller than distances and require less memory. The problem is now, that updating neighbours weights vector has to be different. That algorithm is presented in 4. For this algorithm we do not

Algorithm 4 Locally-informed proposals algorithm 2

Choose a tour $\sigma \in S_n$.
$X_0 \leftarrow \sigma$
Compute weight w_σ.
Compute all neighbour weight differences $\boldsymbol{d}_\sigma$.
for $i = 1, 2, \ldots$ **do**
    $s(\boldsymbol{d}_\sigma) = \text{softmax}(\boldsymbol{d}_\sigma)$
    Sample $\sigma' \sim s(\boldsymbol{d}_\sigma)$.
    Find k, l connected with swapping.
    Find index j of the neighbour σ'.
    $w_{\sigma'} \leftarrow w_\sigma + \boldsymbol{d}_\sigma[j]$
    $\boldsymbol{d}_{\sigma'} \leftarrow \boldsymbol{d}_\sigma$
    for $m = 1, 2, \ldots, N$ **do**
        for $n = m + 1, \ldots, N$ **do**
            if $m \vee n$ in $\{(k-1), k, (k+1), (l-1), l, (l+1)\}$ **then**
                $\boldsymbol{d}_{\sigma'}[(m, n)] \leftarrow$ update difference ▷ neighbour with a swap of m, n.
            end if
        end for
    end for
    $s(\boldsymbol{d}_{\sigma'}) = \text{softmax}(\boldsymbol{d}_{\sigma'})$
    Sample $U \sim Unif(0, 1)$.
    if $\log(U) \leq \min(0, -(w_{\sigma'} - w_\sigma) + \log(s(\boldsymbol{d}_{\sigma'})[(k.l)]) - \log(s(\boldsymbol{d}_\sigma)[(k, l)]))$ **then**
        $X_{i+1}(k), X_{i+1}(l) \leftarrow X_{i+1}(l), X_{i+1}(k)$
        $w_\sigma \leftarrow w_{\sigma'}$
        $\boldsymbol{d}_\sigma \leftarrow \boldsymbol{d}_{\sigma'}$
    else
        $X_{i+1} \leftarrow X_i$
    end if
end for

need to update differences by a common factor. Here we use a term *update difference* to

not blur the algorithm anymore. It means that a sub-algorithm is getting differences for the pair of indices.

5.4 Simulated annealing

There is one important parameter that can be used both with random candidates and locally-informed proposals and is associated with temperature.

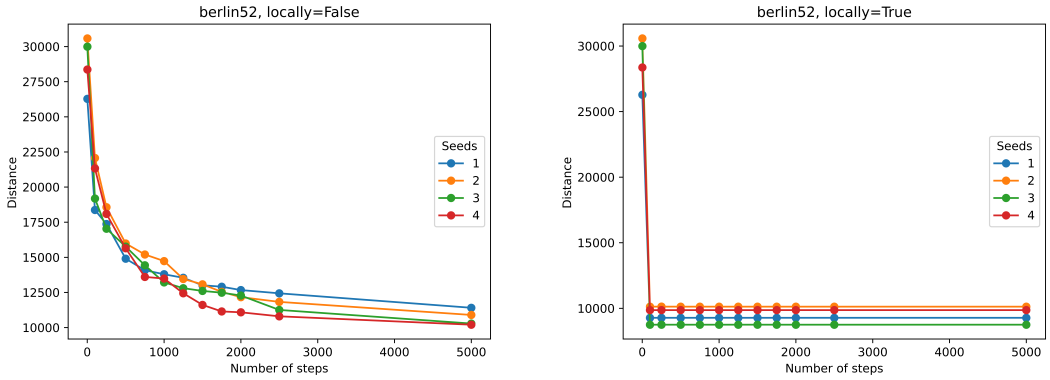
TO DO

6 Results

In this section we will present the results of simulations and compare them with previous methods and true optima.

6.1 Initial condition

First we have to check if changing the initial state makes any difference to the output of an algorithm. This is achieved via setting a different seed and repeating algorithms for the same number of iterations. Seeds guarantee us the different "randomness" with each seed. It is because the generators are only generating *pseudo-random numbers* which we can control. Both algorithms will have the same seeds and in our case these are $1, 2, 3, 4$. We use default parameters of temperature $\tau = 2$ and cooling schedule $t_n = 1$.



(a) Random candidates.

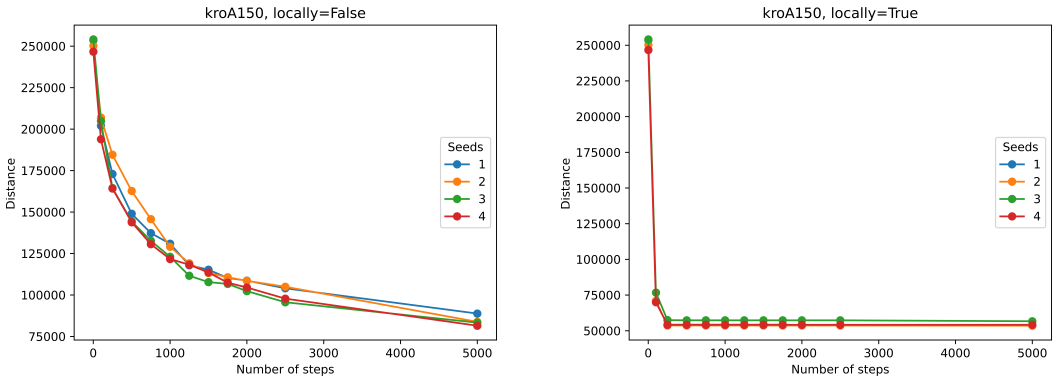
(b) Locally-informed proposals.

Figure 6.1.1: Distances over number of steps for *berlin52* dataset.

Both plots 6.1.1 and 6.1.2 suggest that initial state matters for finding a better tour, but it is not changing the behavior of algorithms.

6.2 Simulated annealing

Before we proceed with comparing algorithms we need to find out if simulated annealing is improving them. We will compare parameters $t_n = 1$ and $t_n = \frac{3}{\log(n+2)}$ where n is a number of step. The temperature parameter for locally-informed proposals will stay at default $\tau = 2$ and seed 1.



(a) Random candidates.

(b) Locally-informed proposals.

Figure 6.1.2: Distances over number of steps for *kroA150* dataset.

7 Conclusions

In this section we will share with our conclusions regarding a new approach of locally-informed proposals.

8 Codebase

In this section we will briefly cover the codebase of our simulations and will walk a reader through, so that he can have a basic understanding of code and will be able to use it by him/herself. The codebase can be found on github [1].

The project has two packages: *mcmc* and *tsp*. The *mcmc* package is a backbone of whole project, as it contains probability constructs like MC and M-H algorithm. The package *tsp* contains classes representing traveling salesman problem.

8.1 mcmc

The most important modules of this package contain classes representing some probabilistic constructs:

- *StochasticProcess* – represents stochastic processes, it is a building block of other classes, as they inherit from it. It needs a function of *next_state*, that tells a process how to move in time. It has ability to remember its past and sample from this process.

- *MarkovChain* – it is derived from *StochasticProcess* and represents a Markov chain. It implements the function *next_state* as a function that does not need to know the past, only the current state.

- *MonteCarloMarkovChain* – it is derived from *MarkovChain* and represents a Markov chain constructed with Metropolis-Hastings algorithm. It is an abstract class which needs to implement *next_candidate* and *log_ratio* among many others. These are different depending on a procedure one is following, one does not need to specify a candidate matrix, because it is cumbersome, rather a procedure of selecting a candidate. Depending on that *log_ratio* will be a different function too.

Besides those classes there are some that are toy examples like *HomogenousMarkovChain* and *MetropolisHastings* that were used for testing. There are also some helpful functions that are used throughout the package.

8.2 tsp

This package contains classes that are required to represent traveling salesman problem and its solution using MCMC methods. There are two important classes:

- *TSPath* – represents a salesman tour and is mostly used as a container for attributes of that tour (so problem information, weight, neighbours *etc.*). It has functions that are finding edges, its weights and computing neighbour weights.
- *TravelingSalesmenMCMC* – represents a Markov chain that is traveling through tour space. It is derived from *MonteCarloMarkovChain* so it implements all needed functions. It can be used both with random candidates or with locally-informed proposals methods.

There is also whole dataset used for simulations and all results obtained while working on this thesis. The package of course contains many more modules or scripts that are used for results exploration.

References

[1] Homeomorphistic/locally-informed-proposals-mcmc: Locally-informed proposals in metropolis-hastings algorithm with applications. https://github.com/Homeomorphistic/Locally-informed-proposals-MCMC. (Accessed on 06/03/2022).

[2] Tsplib. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/index.html. (Accessed on 06/01/2022).

[3] O. Häggström et al. *Finite Markov chains and algorithmic applications*, volume 52. Cambridge University Press, 2002.

[4] C. Karpiński. On use of monte carlo markov chains to decode encrypted text and to solve travelling salesman problem. 2020.

[5] C. J. Maddison, D. Duvenaud, K. J. Swersky, M. Hashemi, and W. Grathwohl. Oops i took a gradient: Scalable sampling for discrete distributions. 2021.