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 approximating 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 przybliżania 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 MCMC methods is is *Metropolis-Hastings* algorithm, which constructs a MC using another set of distributions, maybe simpler ones.

In this thesis we work with *locally-informed proposals*, they involve determining *local* distribution – which comes down to finding transition probabilities. 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 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 a sequence of independent random variables, 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 satisfy the Markov property. These properties allow us to simulate processes 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 natural numbers like $\{1, 2, \ldots, N\}$ (where N is 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* $(X_k)_{k\in T}$ *defined on a common probability space* $(\Omega, \mathcal{F}, P)$*, that take values in* S*, such that it satisfies Markov property:*

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

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

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:*

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

In this case we define:

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

Homogeneous Markov chains 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* k *steps* $\mathbf{P}(k)$ *for a MC is a stochastic matrix constructed using transition probabilities:*

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

Definition 2.8. *An initial distribution of a Markov chain is a vector* $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_N) \in \mathbb{R}^N$ *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* $\boldsymbol{\mu}^{(k)} \in \mathbb{R}^N$ *be a distribution of a MC at* k*-th step and* $\boldsymbol{\mu} \in \mathbb{R}^N$ *the initial distribution, then for all* k*:*

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

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 are somehow connected and some of their properties transfer between them. 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.,*

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

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.,*

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

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 starts with 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.*

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

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

The proofs of the Theorems 2.2 and 2.3 can be found in [3]. 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 approximate sampling 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_k) 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 as:

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

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* or a *proposal distribution* 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.*

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.

Algorithm 1 Metropolis-Hastings algorithm

Choose a state $i \in S$.
$X_0 \leftarrow i$
for $k = 1, 2, \ldots$ **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**
        $X_{k+1} \leftarrow j$
    else
        $X_{k+1} \leftarrow X_n$
    end if
end for

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 $c = (e_1, e_2, \ldots, e_k)$ such that only first and last edge are equal ($e_1 = e_k$).*

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*

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

where S_n *is a set of all permutations of vertices and* $w_{i,j}$ *is distance (weight) between state* i *and* j*.*

This 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 constraints 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 compare 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 Approximate solutions

In the previous section we have seen that a deterministic approach of computing salesman tours is infeasible, so we can turn now to probabilistic methods of MCMC.

5.1 Basic idea

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.

It is still not clear, how does it help with the traveling salesman problem – even with fast sampling algorithm. Why are we interested in probabilities instead of weights? It is, because when looking for a maximum of softmax of a vector, we find a maximum of original vector, so in case of our problem:

$$\sigma_{\min} = \arg\min_{\sigma \in S_n} (w_\sigma) = \arg\max_{\sigma \in S_n} \frac{e^{-w_\sigma}}{\sum_{\sigma' \in S_n} e^{-w_{\sigma'}}},$$

where w_σ is distance (weight) of a tour (permutation) σ. When $\sigma_{\min}$ is maximizing probability it means when we sample from this distribution it will appear most often. This is the core of MCMC methods – approximate solution with most probable state.

5.2 Metropolis-Hastings algorithm

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.

5.3 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.

5.4 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 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.

5.4.1 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 its distance (weight):

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

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):

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

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 $w_{\sigma(k-1),\sigma(k)}$, $w_{\sigma(k),\sigma(k+1)}$, $w_{\sigma(l-1),\sigma(l)}$, $w_{\sigma(l),\sigma(l+1)}$ and add $w_{\sigma(k-1),\sigma(l)}, w_{\sigma(l),\sigma(k+1)}, w_{\sigma(l-1),\sigma(k)}, w_{\sigma(k),\sigma(l+1)}$. This is only 8 operations per neighbour, a constant complexity cost (if we can get instantly weights).

5.4.2 Implementation

To sum up all the information we summarize it in Algorithm (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 $U \sim Unif(0, 1)$.
    $w_{\sigma'} \leftarrow w_\sigma - (w_{\sigma(k-1)+\sigma(k)} + w_{\sigma(k)+\sigma(k+1)}, w_{\sigma(l-1)+\sigma(l)} + w_{\sigma(l),\sigma(l+1)})$
    $w_{\sigma'} \leftarrow w_\sigma + (w_{\sigma(k-1)+\sigma(l)} + w_{\sigma(l)+\sigma(k+1)} + w_{\sigma(l-1)+\sigma(k)} + w_{\sigma(k)+\sigma(l+1)})$
    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

5.4.3 Complexity

As mentioned before this algorithm is simple because of uniform distribution. It requires 2 samplings, which in practice both are the same (random sampling) and 8 operations of adding and subtracting which in theory could be instant, but in reality are of linear complexity, because of searching of weights through the list.

Space complexity is constant – we need only two variables to contain current weight and state. In practice we also use a memory for storing weights of the problem.

5.5 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. This is why the method is called *locally-informed proposals* (abbrv. LIP) – we compute a local proposal. 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).

5.5.1 Theoretical background

The idea is to balance the increase in the probability of neighbour with decrease of reverse probability, such that it will be easy to compute. One way is to use locally-informed proposals, which will be proportional to the same quotient of target distribution as in M-H step, so:

$$\mathbf{Q}_{i,j} \propto e^{\frac{-(w_j - w_i)}{\tau}}.$$

Here $\tau > 0$ is temperature parameter. The distribution is chosen in such a way, so that we can easily group up the terms in acceptance criterion:

$$\frac{\pi_j \mathbf{Q}_{j,i}}{\pi_i \mathbf{Q}_{i,j}} = e^{-(w_j - w_i)} \cdot \frac{e^{\frac{-(w_i - w_j)}{\tau}}}{e^{\frac{-(w_j - w_i)}{\tau}}} \cdot \frac{C_j}{C_i} = e^{\left(-(w_j - w_i)\left(1 - \frac{2}{\tau}\right)\right)} \cdot \frac{C_j}{C_i},$$

where C_i, C_j are normalizing constants. Setting $\tau = 2$ terms in exponent cancel out, so we are left with normalizing constants. Again using logarithms is beneficial in computer science, so the acceptance criterion has form:

$$\log\left(\frac{\pi_j \mathbf{Q}_{j,i}}{\pi_i \mathbf{Q}_{i,j}}\right) = \left(-(w_j - w_i)\left(1 - \frac{2}{\tau}\right)\right) + \log\left(C_j\right) - \log\left(C_i\right).$$

The temperature parameter that achieve balance is $\tau = 2$ and was considered in [6].

5.5.2 Computational considerations

The observation here is similar to the previous one – when we have differences in weights for all neighbours of a starting tour, we can update them and only some of them change in a significant way. Then we can calculate softmax on neighbour weights differences. One might notice now, that this procedure is not symmetrical, because softmax can be different when the sums dividing exponent of weights differences will be different. 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:

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

Now we need to think of a neighbour connected to some other swap for both of those tours, let's say r and s. When r and s are far away from k and l we have almost the same tours, but with a swap on k and l. So the neighbours $\sigma_{r,s}$, $\sigma'_{r,s}$ of σ and σ' respectively

look like:

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

This time we need differences in weights, so assuming we know it for neighbour $\sigma_{r,s}$ of σ, we can see that difference in weight of neighbour $\sigma'_{r,s}$ of σ' is the same, because the weights changed only on r and s place, which are not connected to k and l. This case covers most of the neighbours.

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

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

This time the difference in weights $\sigma_{r,s}$ and $\sigma'_{r,s}$ cannot be the same, because there are different edges connected to s. That forces us to manually get weights corresponding to different edges and update the differences in weights. It means, that some neighbours (swaps) have to be considered separately. These are the scenarios when r or s are elements of $\{(k-1), k, (k+1), (l-1), l, (l+1)\}$.

5.5.3 Example

To see this problem let us focus on some easier example and then generalize it to get an estimation of number of steps required to update weights.

Let us set $k = 2$ and $l = 7$, then the set of indices to consider is $\{1, 2, 3, 6, 7, 8\}$ and the permutations:

$$\begin{aligned}\sigma &= (1, 2, 3, 4, 5, 6, 7, 8, 9, \ldots)\\ \sigma' &= (1, 7, 3, 4, 5, 6, 2, 8, 9, \ldots).\end{aligned}$$

We can represent neighbours as a tuple of integers, which mean the indices to swap. The table 5.5.1 represents neighbours. To not repeat elements, we can focus only on the upper triangle of a matrix of those tuples.

Light gray cells are neighbours with one index swapped close to considered set, and darker gray cells are neighbours with two indices swapped close to considered set.

Unfortunately there is no other way rather than compute manually weight difference for darker gray cells, because their edges close to $k = 2$ and $l = 7$ are completely different. These do not cause a lot of computation, because there are only $\binom{6}{2} = \frac{6\cdot5}{2} = 15$ cases.

We can say something more about light gray ones, where there is only one index close to the considered set. Let us set $r = 3$ and $s = 9$, which means that we are looking for weight differences of neighbours obtained by swapping 3 and 9. So the neighbours have form:

$$\begin{aligned}\sigma_{3,9} &= (1, 2, 9, 4, 5, 6, 7, 8, 3, \ldots)\\ \sigma'_{3,9} &= (1, 7, 9, 4, 5, 6, 2, 8, 3, \ldots).\end{aligned}$$

	2	3	4	5	6	7	8	9	10
1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)	(1,9)	$\cdots$
2		(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)	(2,9)	$\cdots$
3			(3,4)	(3,5)	(3,6)	(3,7)	(3,8)	(3,9)	$\cdots$
4				(4,5)	(4,6)	(4,7)	(4,8)	(4,9)	$\cdots$
5					(5,6)	(5,7)	(5,8)	(5,9)	$\cdots$
6						(6,7)	(6,8)	(6,9)	$\cdots$
7							(7,8)	(7,9)	$\cdots$
8								(8,9)	$\cdots$
9									$\ddots$

Table 5.5.1: The Table representing neighbours.

Let us denote $d_{3,9} = w_\sigma - w_{\sigma_{3,9}}$ as a difference between current permutation and its neighbour connected to swap $r = 3$ and $s = 9$. Similarly $d'_{3,9} = w_{\sigma'} - w_{\sigma'_{3,9}}$ is a difference between neighbour of current permutation and its neighbour. Assume we know vector of differences for σ and its weight w_σ. Given that, we also know $w_{\sigma_{3,9}}$, because

$$w_{\sigma_{3,9}} = w_\sigma + d_{3,9}.$$

Consider difference of weight differences here:

$$d_{39} - d'_{3,9} = w_\sigma - w_{\sigma_{3,9}} - (w_{\sigma'} - w_{\sigma'_{3,9}}),$$

after rearranging we have:

$$d'_{3,9} = d_{3,9} - (w_\sigma - w_{\sigma'}) + (w_{\sigma_{3,9}} - w_{\sigma'_{3,9}}).$$

We are interested in getting $d'_{3,9}$. The only term in this equation not known to us is $w_{\sigma_{3,9}} - w_{\sigma'_{3,9}}$ (we know $(w_\sigma - w_{\sigma'})$ because we know differences in weights of σ). It is easy to see when looking at $\sigma_{3,9}$ and $\sigma'_{3,9}$ that :

$$w_{\sigma_{3,9}} - w_{\sigma'_{3,9}} = (w_{1,2} + w_{2,9} + w_{6,7} + w_{7,8}) - (w_{1,7} + w_{7,9} + w_{6,2} + w_{2,8}).$$

We can generalize that equation for any $p \notin \{1, 2, 3, 6, 7, 8\}$:

$$w_{\sigma_{3,p}} - w_{\sigma'_{3,p}} = f_3(p) = C + w_{2,p} - w_{7,p}.$$

This difference is some function of p, changing this index will not change values of any other weights. So we can have a general formula for any $r \in \{1, 2, 3, 6, 7, 8\}$ and $s \notin \{1, 2, 3, 6, 7, 8\}$, constant C and indices $2, 7$ will change depending on k, l and their relative position to r.

Now we know, that for a given row from $\{1, 2, 3, 6, 7, 8\}$ we can compute the difference $d'_{3,9}$. We can do it efficiently using vector operations – we need one vector subtraction and one computation of constant C. In reality vector subtraction is just subtracting each number for each other, so for each $r \in \{1, 2, 3, 6, 7, 8\}$ we have at most n (number of vertices) operations and we have 6 distinct choices of r, so $6 \cdot n$ operations.

5.5.4 Implementation

To sum up all the information we summarize it in Algorithm (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 differences and updating them after each step of a main loop.

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.

Working on differences can alleviate this problem (for some cases). 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: $\boldsymbol{d}_\sigma[i] = w_\sigma + d_i$. By $\boldsymbol{d}_\sigma[(i,j)]$ we mean the index of a swap i, j (it needs to be found).

Algorithm 3 Locally-informed proposals algorithm

Choose a tour $\sigma \in S_n$.
$X_0 \leftarrow \sigma$
Compute weight w_σ.
Compute all neighbour weight differences $\boldsymbol{d}_\sigma$.
$s(\boldsymbol{d}_\sigma) = \text{softmax}(\boldsymbol{d}_\sigma)$
for $i = 1, 2, \ldots$ **do**
    Sample $\sigma' \sim s(\boldsymbol{d}_\sigma)$.
    Find k, l connected with swapping.
    $w_{\sigma'} \leftarrow w_\sigma + \boldsymbol{d}_\sigma[(k,l)]$
    $\boldsymbol{d}_{\sigma'} \leftarrow \boldsymbol{d}_\sigma$
    $\boldsymbol{d}_{\sigma'} \leftarrow \mathit{update\_differences}(\boldsymbol{d}_{\sigma'})$
    $s(\boldsymbol{d}_{\sigma'}) = \text{softmax}(\boldsymbol{d}_{\sigma'})$
    Sample $U \sim Unif(0,1)$.
    if $\log(U) \leq \min\left(0, -(w_{\sigma'} - w_\sigma) + \log(s(\boldsymbol{d}_{\sigma'})[(k,l)]) - \log(s(\boldsymbol{d}_\sigma)[(k,l)])\right)$ **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

The main algorithm uses a sub-algorithm 4 which updates weights in swaps that we need to consider separately. It uses function *get_difference*, which is just getting appropriate weights to remove and add.

Alternate algorithm explanation: ??

5.5.5 Complexity

This algorithm is more complicated and it significantly lowers the performance. This time for each step of M-H algorithm we need to compute softmax to get probabilities, which

Algorithm 4 update_differences

Require: $\boldsymbol{d}_{\sigma'}$
Ensure: $\boldsymbol{d}_{\sigma'}$
1: **for** $r = 1, 2, \ldots, n$ **do**
2: **for** $s = r + 1, \ldots, n$ **do**
3: **if** r or s in $\{(k-1), k, (k+1), (l-1), l, (l+1)\}$ **then**
4: $\boldsymbol{d}_{\sigma'}[(m, n)] \leftarrow get\_difference(\boldsymbol{d}_{\sigma'}[(r, s)])$
5: **end if**
6: **end for**
7: **end for**

Algorithm 5 update_differences_alt

Require: $\boldsymbol{d}_{\sigma'}$
Ensure: $\boldsymbol{d}_{\sigma'}$
1: **for** $r = k - 1, k, k + 1, l - 1, l, l + 1$ **do**
2: **for** $s = r + 1, \ldots, n$ **do**
3: $\boldsymbol{d}_{\sigma'}[(m, n)] \leftarrow get\_difference(\boldsymbol{d}_{\sigma'}[(r, s)])$
4: **end for**
5: **end for**
6: **for** $r = k + 2, k + 3, \ldots, l - 2$ **do**
7: **for** $s = l - 1, l, l + 1$ **do**
8: $\boldsymbol{d}_{\sigma'}[(m, n)] \leftarrow get\_difference(\boldsymbol{d}_{\sigma'}[(r, s)])$
9: **end for**
10: **end for**

is already a hard task, because of computing exponents. Besides that we need to sample from (sometimes) long vector. There is also a sub-procedure with two loops.

The function *get_difference* is not implemented as in example, because we wanted to avoid working with large matrices and so avoid problems with memory. In practice one would need to keep matrix of weights, which would be of size $n \times n$ and the same for differences. More than a half of its elements would be empty (zeros) so we would waste a lot of memory and also do many unnecessary operations when adding or subtracting vectors from those matrices. Not using large matrices was also our goal when choosing MCMC methods for this problem.

Space complexity is also constant, but a lot bigger than previously, because we need to keep two long vectors of probabilities (or weight differences).

5.6 Simulated annealing

There is one important parameter that can be used both with random candidates and locally-informed proposals and is associated with cooling. It has its roots within physics and is connected to energetic states of particles and Boltzmann distribution. The idea is to describe a probability of state using a temperature parameter (different than τ in LIP) such that it reminds the cooling of a metal:

$$\pi_i = \frac{e^{\frac{-E_i}{t_n}}}{C},$$

where C is normalizing constant.The quotient of probabilities then is:

$$\frac{\pi_j}{\pi_i} = e^{\frac{E_i - E_j}{t_n}}$$

Setting this parameter high constitutes to all of the states are equally likely and with low value of parameter we have probability concentrated in the state with minimal energy.

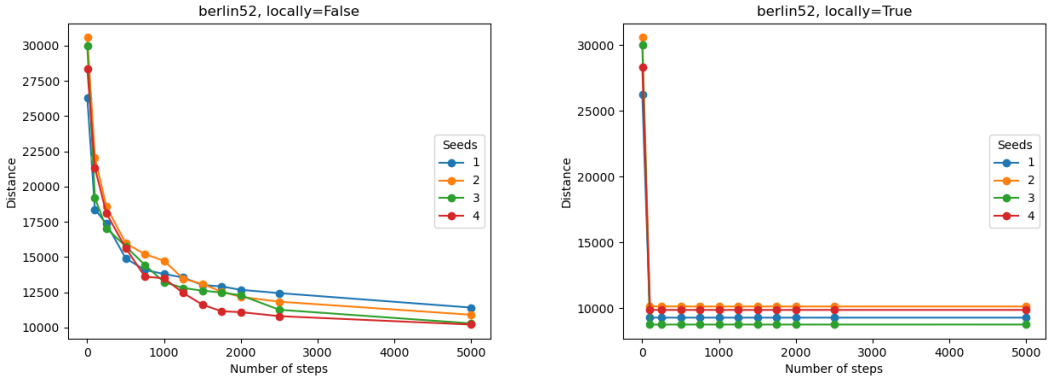
Using t_n as a function of step in acceptance criterion will give us a non-homogeneous Markov chain. Setting it to a constant value will produce a homogeneous Markov chain.

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 parameter $t_n = 1$.



(a) Random candidates.

(b) Locally-informed proposals.

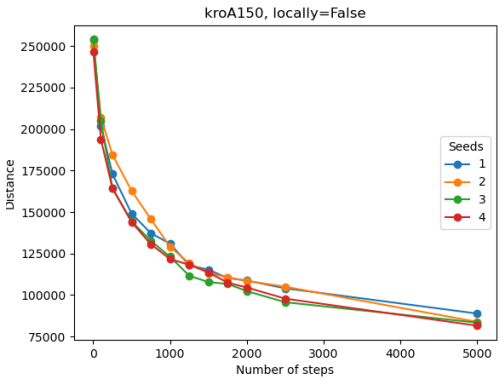
Figure 6.1.1: Different initial states for *berlin52*.

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.

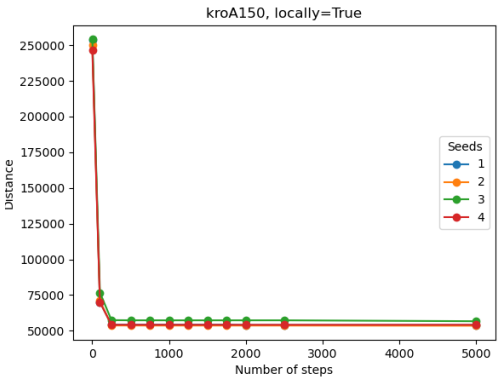
6.2 Simulated annealing

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.

This time to see the difference between cooling parameters we had to extend number of iterations to 1 million. This number of steps is not viable for locally-informed proposals,

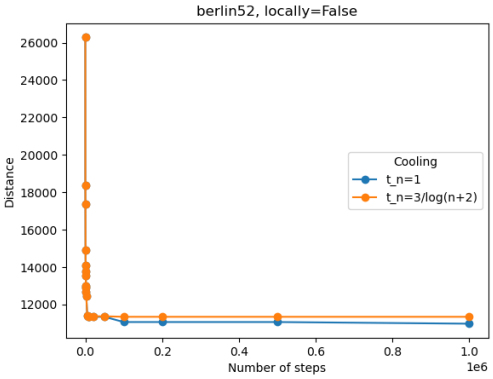


(a) Random candidates.

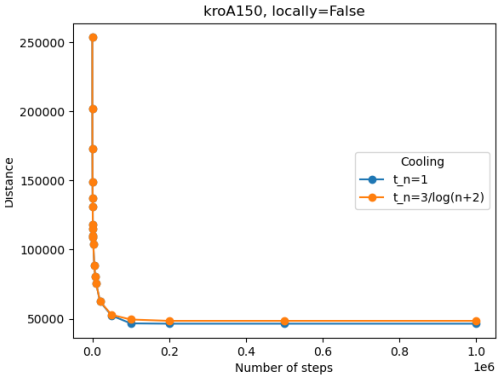


(b) Locally-informed proposals.

Figure 6.1.2: Different initial states for *kroA150*.

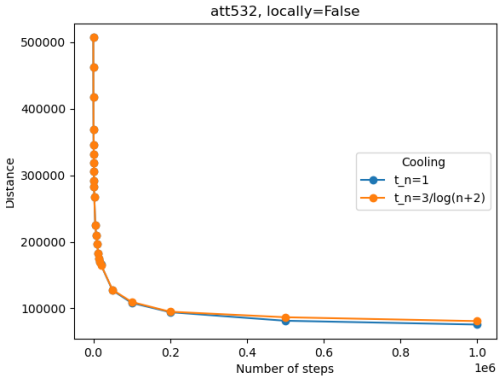


(a) berlin52.

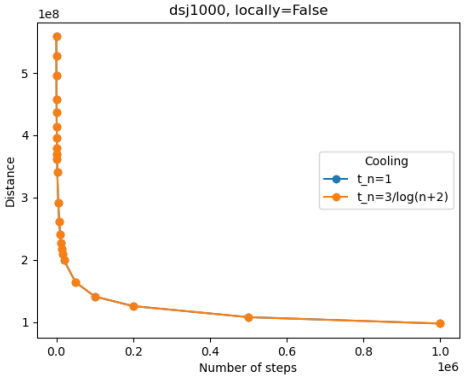


(b) kroA150.

Figure 6.2.1: Different cooling parameters for *berlin52* and *kroA150*.



(a) att532.



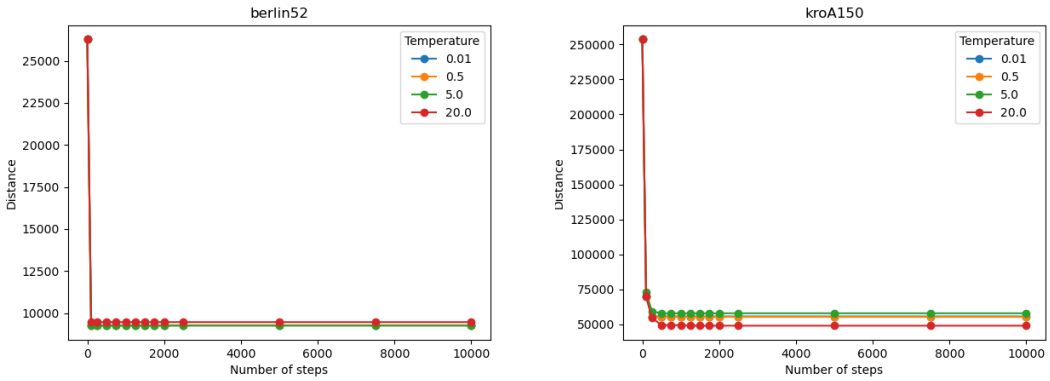
(b) dsj1000.

Figure 6.2.2: Different cooling parameters for *att532* and *dsj1000*.

so only random candidates were used. We can see slight the improvement at the end of iterations, which has potential when the distances are huge.

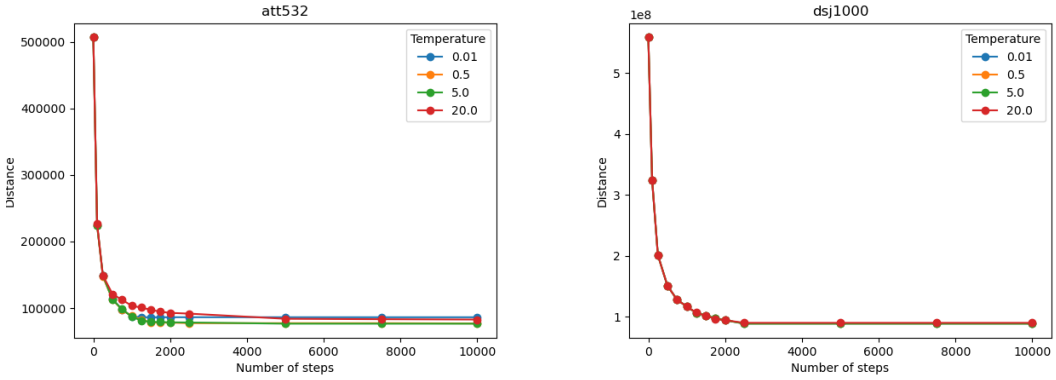
6.3 Temperature

The same needs to be done with temperature parameter τ. It was proven in [6] that $\tau = 2$ is optimal for balancing out likelihoods, but maybe for this problem it is worth considering increasing some probability by this parameter.



(a) berlin52. (b) kroA150.

Figure 6.3.1: Different temperature parameters for *berlin52* and *kroA150*.



(a) att532. (b) dsj1000.

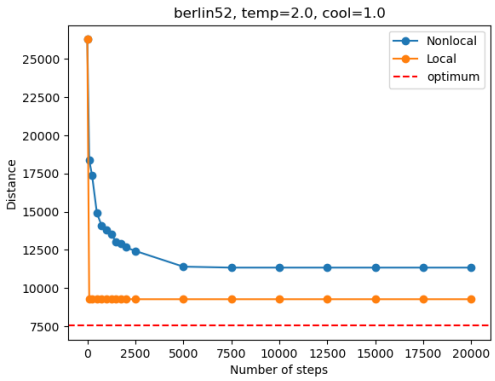
Figure 6.3.2: Different temperature parameters for *att532* and *dsj1000*.

The plots 6.3.1 and 6.3.2 present LIP algorithm with the same initial state and different temperature parameter $\tau = 0.01, 0.5, 5, 20$. For two datasets the results are almost identical, while on two others some temperatures offer better results, but they are not consistent with each other. This is why we opted for using $\tau = 2$.

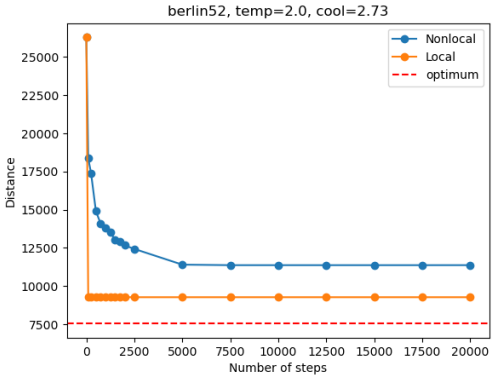
6.4 Algorithms comparison

Now we are able to present all the results, we will be using $\tau = 2$, two different cooling schedules $t_n = 1$, $t_n = \frac{3}{\log(n+2)}$ and two different methods: random candidates and locally-informed proposals and the number of iterations is 5000.

The plots 6.4.1, 6.4.2, 6.4.4, 6.4.5 present distance of a solution as a function of number of steps. Both algorithms start at the same state and have the same seed. We can se how quickly they diverge. First steps of locally-informed proposals algorithm is taking

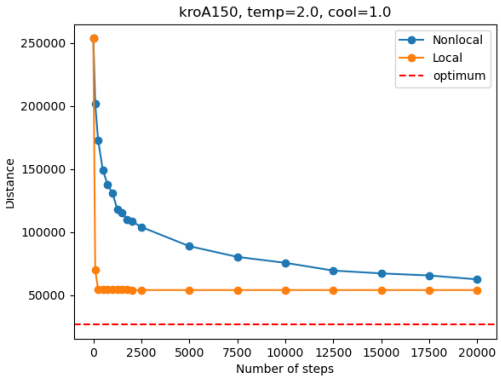


(a) $t_n = 1$.

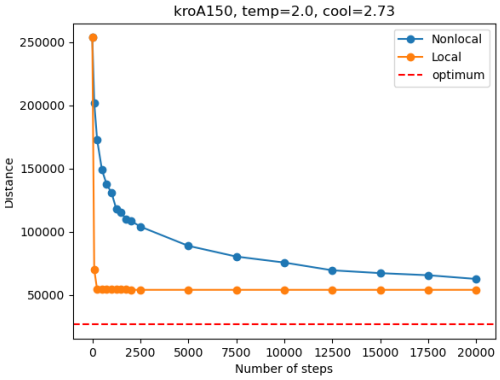


(b) $t_n = \frac{3}{\log(n+2)}$.

Figure 6.4.1: Comparing random neighbours and LIP for *berlin52* with different cooling parameters.

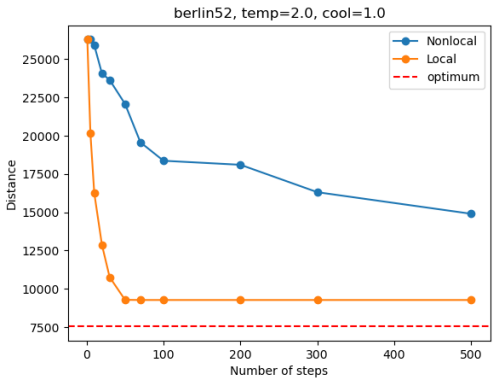


(a) $t_n = 1$.

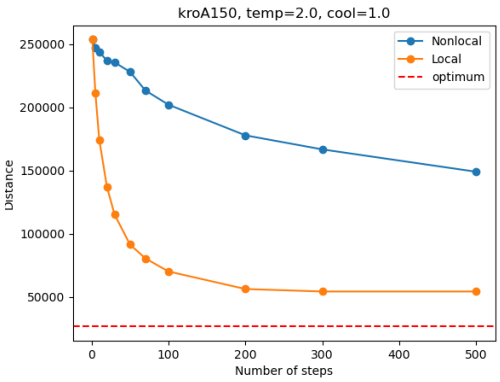


(b) $t_n = \frac{3}{\log(n+2)}$.

Figure 6.4.2: Comparing random neighbours and LIP for *kroA150* with different cooling parameters.

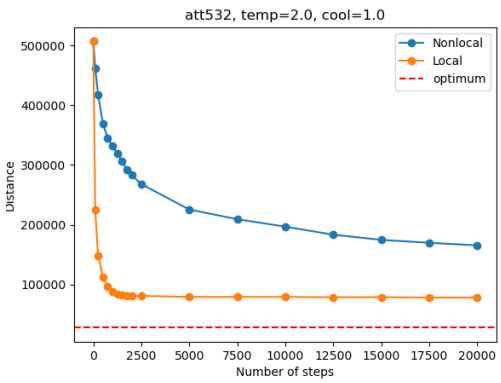


(a) berlin52, $t_n = 1$.

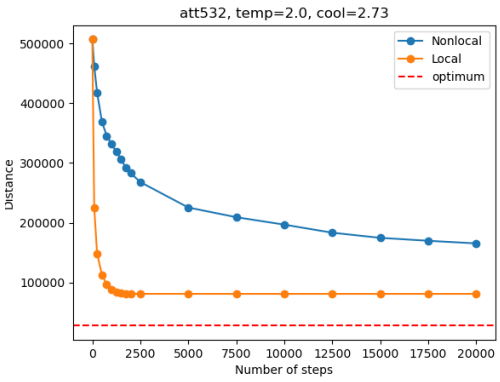


(b) kroA150, $t_n = 1$.

Figure 6.4.3: Comparing random neighbours and LIP for *berlin52* and *kroA150* with low number of iterations.

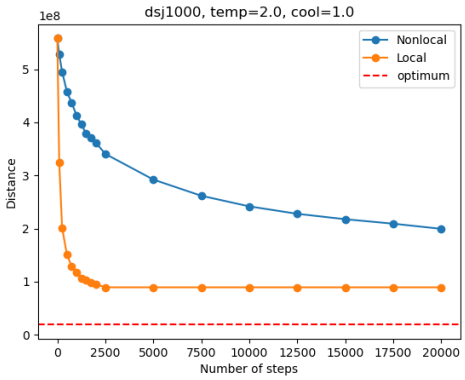


(a) $t_n = 1$.

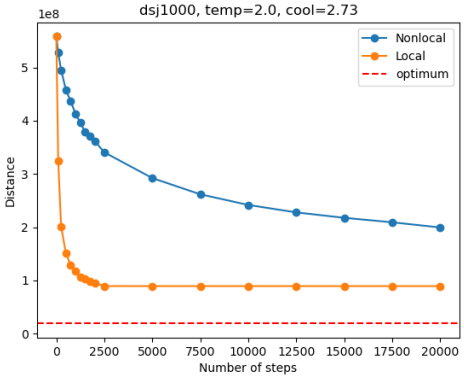


(b) $t_n = \frac{3}{\log(n+2)}$.

Figure 6.4.4: Comparing random neighbours and LIP for *att532* with different cooling parameters.



(a) $t_n = 1$.



(b) $t_n = \frac{3}{\log(n+2)}$.

Figure 6.4.5: Comparing random neighbours and LIP for *dsj1000* with different cooling parameters.

the best neighbours it can and stays there or makes small improvements. Prolonged stays happen in smaller datasets as there is less deviation at each step and it is harder to choose a better neighbour. At the beginning LIP algorithm has obviously better results in all cases.

Additionally the plot 6.4.3 compares two methods but on a shorter distances and only for two smaller dataset. The number of vertices is smaller, so LIP finds better solutions quickly and it is easier to compare the methods with fewer iterations. We can see, that less than 100 iterations are enough to create a massive gap between these methods.

The tables 6.4.1 and 6.4.2 present comparison of those two methods for different cooling parameters. They contain distances of tours, time the algorithm took and ratio of the solution to the optimum. For bigger datasets the ratio is growing, which is behaving as expected – the more vertices to search through, so more iterations are needed to reach optimum. The LIP algorithm outperforms random candidates, given small amount of iterations. We can also notice that the cooling parameter does not cause any significant difference.

The table 6.4.3 presents time required for random neighbours algorithm to reach the

	Random neighbours			LIP		
dataset	dist.	time $[s]$	ratio	dist.	time $[s]$	ratio
berlin52	11344	1.32	1.50	9276	116	1.23
kroA150	62467	1.24	2.36	53988	454	2.04
att532	165445	1.38	5.98	78150	2462	2.82
dsj1000	199343433	1.5	10.68	89105687	6439	4.78

Table 6.4.1: Methods comparison, $t_n = 1$.

	Random neighbours			LIP		
dataset	dist.	time $[s]$	ratio	dist.	time $[s]$	ratio
berlin52	11374	1.27	1.51	9276	113	1.23
kroA150	62591	1.28	2.36	53988	450	2.04
att532	165387	1.38	5.97	80969	2467	2.92
dsj1000	199343433	1.55	10.68	89105687	6437	4.78

Table 6.4.2: Methods comparison, $t_n = \frac{3}{\log(n+2)}$.

dataset	time [s]
berlin52	> 500
kroA150	3.43
att532	49.43
dsj1000	169.57

Table 6.4.3: Time to reach the same result for random neighbours.

same result as LIP algorithm (with the same seeds). As we can see is much lower than for its competitor with the exception of *berlin52* dataset. This one could not converge to the same distance.

7 Conclusions

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

Every result confirms that LIP algorithm is decreasing the distance quicker than randomly choosing neighbours, no matter the dataset. This is understandable because at the beginning it easily chooses the best neighbours. As time (iterations) passes by, there is a difficulty in finding neighbour that improves the distance, so locally-informed proposal is flattening and algorithm slowly converges to some value. What is important, that this value is almost always smaller than the random candidates algorithm. Given more time (iterations) this randomness should reach also the better result.

Only datasets with small number of vertices (52, 150) have close results, as the number of neighbours is smaller, so random picking is able to reach smaller distances quicker.

This of course, comes at a price of computational complexity. The LIP algorithm is considerably slower. While random algorithm oscillates within 1.5 seconds for each dataset, LIP algorithm reaches over 1.5 hour for the biggest dataset.

It may seem that the time here is an obvious drawback and there is no point in further studying this approach. However, there are still many areas to improve, which makes this

method promising.

7.1 Areas to improve

There are some important areas to improve upon, that could significantly speed up the algorithm. There are two significant problems: updating weights and sampling from the softmax of differences in weights.

Updating differences in weights depends on problem, so for some it could be done really efficiently. As we mentioned earlier, getting weights for edges is not a constant complexity, because we need to search through a vector of weights for many edges (there many more than vertices), which grows quadratically with number of vertices. One way to alleviate this problem is to use *interning*, a method in *Python3* language, which saves objects into Python's memory for quick access. Of course, better knowledge of data structures could be helpful, for example, maybe looking through dictionaries is faster than through lists. This is lowering the complexity.

The second problem of sampling is actually a close problem to the one that we are trying to solve now – how to sample efficiently using MCMC methods. Sampling in this thesis is done by sampling exactly from a long vector, but it seems that the locally-informed proposal is a perfect candidate for M-H algorithm. We know that quotient of LIP is:

$$\frac{\mathbf{Q}_{j,i}}{\mathbf{Q}_{i,j}} = \frac{e^{\frac{-(w_i - w_j)}{\tau}}}{e^{\frac{-(w_j - w_i)}{\tau}}} \cdot \frac{C_j}{C_i} = e^{\frac{-2}{\tau}(w_i - w_j)} \cdot \frac{C_j}{C_i},$$

and logarithm of that has form:

$$\log\left(\frac{\mathbf{Q}_{j,i}}{\mathbf{Q}_{i,j}}\right) = \frac{-2}{\tau}(w_i - w_j) + \log(C_j) - \log(C_i).$$

So if we take this as a acceptance criterion in M-H algorithm, we just need 2 weights and updating normalizing sums. The open question is, how to choose a proposal for this algorithm, such that it will be an efficient way of sampling. This has enormous potential in reducing complexity cost.

There is also a problem of choosing temperature and cooling parameters. It could be done by cross-validation, but it adds another complexity level on top of already slow algorithm.

To summarize, there is plenty of ways to improve the algorithm, not only through mathematical analysis but also through smarter employment of specialized tools and improving quality of code in *Python3*. The LIP algorithm has a potential for finding minima and solving many NP-hard problems with fewer iterations and maybe in the future with amount of less time.

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.

[6] G. Zanella. Informed proposals for local mcmc in discrete spaces. *Journal of the American Statistical Association*, 115(530):852–865, 2020.