STAT4799: Statistics Project 2019-2020
Randomized Algorithms

Peng Cheng (UID: 3035329601)
Supervisor: Dr. Cheung Ka Chun
Department of Statistics and Actuarial Science
The University of Hong Kong

Contents

Abstract

Randomized algorithms are a variety of algorithms which make some choices randomly when executing. Given a fixed input, unlike those deterministic algorithms which always give a fixed output in polynomial time, a randomized algorithm incorporates random numbers during the execution and the behaviours may vary when repeating the experiments. Hence, randomized algorithms are also called probabilistic algorithms [12]. Now that the randomized algorithms' performances are uncertain while the deterministic algorithms have no such concerns, why do we still need randomized algorithms? Indeed, randomized algorithms are good approaches to lower the time complexity or space complexity of a deterministic algorithm especially when tackling the NP-hard problems [13]. In any situations of indeterminacy, a randomized algorithm helps by generating random numbers in order to make a decision, which speeds up the brute force approach and obtain a comparatively optimal solution in a fairly short time. Even though the solution may not be the global optimum, it is already good enough for our purpose. On the contrary, the deterministic algorithms may take tremendous amount of time to output the real global optimum. Such trade-off reflects the value of randomized algorithms.

In this thesis, I will introduce my discovery incorporating the ideology of randomized algorithms in three topics, namely the application of Gibbs Sampler on the Vertex Colouring Problem, the application of Simulated Annealing algorithm on the Travelling Salesman Problem and the usage of Comonotonicity to improve the Naive Bayes Classifier. In the first part, a famous Markov Chain Monte Carlo algorithm, Gibbs sampler, is applied to the Vertex Colouring Problem and I have discovered some interesting features about it. In the second part, I proposed two amendments to the traditional simulated annealing algorithm and one of them really boosted the performance when applied to the Travelling Salesman Problem. In the third part, I use comonotonicity and clustering to replace the naive assumption of mutual independence in the Naive Bayes Classifier and it improved the performance significantly when applied to image recognition as well as highly correlated data-sets.

1 Gibbs Sampler on Vertex Coloring Problem

1.1 Background

In this part, I mainly study the application of Gibbs Sampler on the Vertex Coloring Problem (VCP) and discover some interesting features related to this. The VCP states that, given a connected graph which consists of n vertices, e edges and q colors, we are required to assign colors to each vertex such that any two adjacent vertices are in different colors. Former researches about VCP mainly focused on the minimum amount of colors we need to use in order to color the graph, but here I move the focus to two concerns arisen from the application of Gibbs sampler on this problem. All the code was implemented from scratch.

1.2 Two Concerns

1.2.1 About vertices

We call a certain valid way to assign colors to each vertex as one configuration. Assume all configurations are uniformly distributed, what is the average number of vertices which are all occupied a same certain color, say the red color? Intuitively, we may think that it should be the same for each color, which is $\frac{n}{q}$. Then how do we verify it? If there is a method to "traverse" the set of configurations, even if the traverse is not complete (i.e. do not traverse exactly every configuration), will the average number of vertices occupied by a certain color converge during the traversal?

1.2.2 About edges

The colors are numbered from 0 to $\mathbf{q} - 1$. We define the type of an edge by the colors of the two vertices it connects with the smaller one at first, e.g., an edge connecting two vertices of color 0 and color 1 respectively is called type $(0, 1)$. If the assignment of colors is "randomized" enough, i.e., uniformly distributed, then the expected number of edges of type (x, y) should be $\frac{e}{qC2}$. But, how do we show this?

1.3 Reasons for using Gibbs sampler

Obviously, traversing all possible configurations is an NP-hard problem because when we increase the number of vertices or the amount of colors, the number of valid configurations grows exponentially. Thus, it is not practical to count by combinatorics, which makes it difficult to prove our assumptions above. For a complex distribution, since the Markov Chain Monte Carlo (MCMC) methods do some simulation on direct draws [9], it is a feasible way to tackle this issue. Furthermore, one important MCMC method, the Gibbs sampler, is widely used when we wish to obtain characteristics of a marginal density, say $f_X(x)$ given a joint density $f_{X,Y_1 \dots Y_N}(x, y_1 \dots y_N)$. In low dimensional cases, this can be easily computed by multi-integration on variables other than the one we are interested. However, it becomes much more difficult when it comes to high dimensional cases. Gibbs sampler tackles this problem by randomly generating a set of independent and identically distributed samples $x_1 \dots x_N$ which follow $f_X(x)$ [2]. Assume the characteristic function is $h(x)$. Then we can generate a sufficiently large number of samples by running the Markov Chain, $E(h(x)) \approx \lim_{n \to +\infty} \frac{1}{n} \sum_{i=1}^{n} h(x_i)$. In this circumstance, let $\mathbf{S}$ denote the set of all valid configurations and $h(x)$ is a function whose domain is $\mathbf{S}$. We should note that $\mathbf{S}$ can be extremely large. Then according to the theory of Gibbs sampler, we can "simulate" the process by drawing a sample from the domain each time and continue this process for a large number of iterations. Finally, we use the average value to estimate the "real" expectation. However, the core problem is, how to do the sampling following the posterior distribution? In general, we need to construct a "transition mechanism" satisfying the Markov property, i.e., $P(x_{i+1}|x_i, x_{i-1}...x_0) = P(x_{i+1}|x_i) \forall i \in N$. After that, we can run this Markov chain and get a series of samples.

1.4 Transition Mechanism

We apply the Gibbs sampler algorithm on this problem. Suppose $\mathbf{Q}$ is the set of all colors. Generally, in every iteration, we randomly pick one vertex, fix the set $\mathbf{Q'}$ of all the valid colors to this vertex. Then, we uniformly pick one color from $\mathbf{Q'}$ and update the color of this vertex. For simplicity, we can update the vertices sequentially rather than randomly without the loss of generality. That is, in the ith iteration, we update the $i \ mod \ n$th vertex. The updating mechanism remains the same.

The sufficient condition of the validity of MCMC are **aperiodicity** and **irreducibility**. For aperiodicity, since $\mathbf{Q'}$ contains the current color of the vertex, the probability of staying in the same state is non-zero. Hence, aperiodicity is satisfied. For irreducibility, basically, the number of colors really matters. For example, if the graph is fully connected and $\mathbf{q} = \mathbf{n}$, i.e., each vertex occupies a unique color and there is no more color. Then there is no way to update the configuration of this graph. Thus, we set a secure lower bound for the number of colors. Denote $\mathbf{d}$ as the maximum degree among the vertices in the graph, let $\mathbf{q} > 2 * \mathbf{d}$. I present an algorithm to find a series of transitions from a configuration $\mathbf{A}$ to another configuration $\mathbf{B}$.

Suppose $\mathbf{V}$ is the set of vertices of the graph, then $\mathbf{A} = \{a_{v_i} | v_i \in \mathbf{V}\}$ and $\mathbf{B} = \{b_{v_i} | v_i \in \mathbf{V}\}$. Suppose $\mathbf{C}$ is the set of vertices which occupy different colors in $\mathbf{A}$ and $\mathbf{B}$ and let $|\mathbf{C}| = m$, i.e., there are totally m vertices in the graph which occupy different colors in $\mathbf{A}$ and $\mathbf{B}$. Let $\mathbf{A'} = \{a_{c_i} | c_i \in \mathbf{C}\}$ and $\mathbf{B'} = \{b_{c_i} | c_i \in \mathbf{C}\}$.

The algorithm's idea is as follows. We apply this algorithm on A, B and a vertex c_i. *Case 1:* If c_i is modified, then just break. *Case 2:* If the configuration $A \setminus \{a_{c_i}\} \bigcup \{b_{c_i}\}$ is valid, then update A to $A \setminus \{a_{c_i}\} \bigcup \{b_{c_i}\}$. *Case 3:* We randomly pick one color which is not occupied by the neighbors of b_{c_i}, assign the picked color to a_{c_i}, recursively call this algorithm on A, B and c_j, update A to $A \setminus \{a_{c_i}\} \bigcup \{b_{c_i}\}$.

Then we apply the algorithm to each vertex in $\mathbf{C}$ in order. Here are some notations to the algorithm:

1. The function is applied to one single vertex
2. In the third case in which changing the color of c_i to the objective color in **B** directly is invalid, since we have assumed that $\mathbf{q} > 2 * \mathbf{d}$, we can always find a valid color which not occupied by the neighbors of b_{c_i}.
3. In the third case, the adjacent vertex which occupy the objective color in **A** must be in **C**. Otherwise, **A** is no longer valid because **A** **B** share the same color assignment in vertices other than **C**.

Using this algorithm, we can always find a series of transitions between any two valid configurations. Thus, the Markov Chain is **irreducible** and **aperiodic**, which proves that the Gibbs sampler is a valid MCMC method to do the simulation task.

1.5 Objectives of doing this research

Now that we have proved that Gibbs sampler is a good method to do simulation for this problem, we wish to find something interesting related to vertices and edges. The process is mainly to initialize the coloring of the graph randomly, i.e., to generate X_0 randomly, then run the Markov Chain to generate samples X_1, X_2, X_3 .... The state at time t should be generated based on the posterior distribution given the state at time $t-1$. Based on the samples, simulations can be done in a lower computational cost compared with traversing all valid configurations. Thus, as we have discussed before, in terms of vertices, we will mainly discover:

1. For a certain graph, when changing the total number of colors we use or when changing the connectivity of the graph (i.e. to add some more edges to the original graph), if we only focus on the vertices occupying a same color, how will the speed of convergence of the average number of such vertices be affected?
2. If we focus on all colors and construct a $1 * \mathbf{q}$ vector, the ith entry represents the average number of vertices which occupy the color i. Then we focus on the average absolute distance between this vector and the vector of expectation, i.e., a $1 * \mathbf{q}$ vector in which every entry has value $\frac{1}{q}$). What will the speed of convergence of the average absolute distance be affected this time?

In terms of edges, we will mainly discover:

1. Will the number of each type of edges converge to $\frac{e}{qC2}$?
2. If so, will there be any difference in the speed of convergence between different types?

1.6 Experiment

1.6.1 Construction of Graph

I implemented two algorithms to construct the graph:

1. We first random shuffle the vertices list. Then iterate the vertices list. Each time we pop the head of the list, i.e., return the first element of the list and delete it. For the ith vertex, randomly connect it with k vertices among the first $i-1$ vertices. k is a random integer between 1 and a threshold which is set as a hyper-parameter.

2. Since for every connected graph, we can always delete some edges so that it can become a tree. We can generate a tree, which is already connected and then add some more edges randomly afterwards. When generating the tree, we also random shuffle the vertices list. Then every time we pop the list and set the poped vertex as the right-most node of the current tree. The number of sub-trees for each non-leaf node is randomly assigned. After the construction of the tree, we add some more edges to this tree.

Generally, the first method can generate a graph with larger connectivity, which is measured by average degree.

1.6.2 Initialization of Colors

As we have stated before, denote the maximum degree among the vertices in a graph as $\mathbf{d}$, then for security, we let $\mathbf{q} > 2 * \mathbf{d}$. Initially, we set Null to all vertices' colors. We assign the colors to vertices in a random order. Then we iterate the shuffled list. Each time we search the colors of the neighbours of the current vertex and uniformly choose one from all the valid colors. Since $\mathbf{q} > 2 * \mathbf{d}$, there must exist available colors in every iteration.

1.6.3 Iteration

Basically, when running the Markov Chain, every time we need to select a vertex randomly and update its color. Let $\mathbf{Q_i}$ be the set of colors occupied by the neighbors of vertex i. The new color is selected uniformly from $\mathbf{Q} \setminus \mathbf{Q_i}$. Thus, it is possible to stay in the same configuration after an iteration. Without the loss of generality, we update the colors of vertices sequentially, i.e., update the color of vertex $imodn$ in the ith iteration.

1.6.4 Methods to compare speed of convergence of two sequences

Since we are concerned about convergence, we need a practical method to compare the speed of convergence of sequences which are generated when running the Markov Chain. The method that I have tried are as follows.

Table 1: Result of Method 1

Number of Vertices	Max connection with former vertices	Avg degree	Max degree
50	10	16.0	27
100	10	18.0	35
150	10	18.67	39
200	10	19.0	46

Order of convergence Mathematically, a practical method to calculate the order of convergence is to calculate the following sequence which converges to q :

$$q \approx \frac{log|\frac{x_{k+1}-x_k}{x_k-x_{k-1}}|}{log|\frac{x_k-x_{k-1}}{x_{k-1}-x_{k-2}}|}$$

For two convergent sequences, it is feasible to compare their speed of convergence by computing their order of convergence respectively as long as q converges for when k becomes large.

Ratio Test It is also feasible to employ ratio test when comparing the speed of convergence of two convergent sequences. Suppose there are two convergent positive sequences $\{a_n\}$ and $\{b_n\}$, if the sequence $\{\frac{a_n}{b_n}\}$ converges to 0, then $\{a_n\}$ converges faster. If it converges to any positive real number, then the speed of convergence for the two sequences are similar. If it diverges, then $\{b_n\}$ converges faster.

Show by Plots The most intuitive and straightforward way to compare the speed of convergence is by plotting the sequences and compare them directly. According to the results of some tentative experiments, it is impractical to compute the order of convergence because there is always no such a q which is stable when k becomes large. Meanwhile, ratio test is also proved impractical because the sequence of ratio neither converge nor diverge to positive infinity. Thus, the only way we can use is to show by plots.

1.6.5 Initialization of Graph

Before doing the experiments, I tested the two methods for the initialization of the graph. The setting of hyper-parameters are shown in Tables 1 and 2. Please note that this procedure aims to compare the two methods rather than finding the best hyper-parameters. Meanwhile, the setting of hyper-parameters is not the focus of this research:

Table 2: Result of Method 2

Number of Vertices	Max subtree	Max added edges	Avg degree	Max degree
50	4	8	2.28	5
100	4	8	2.14	6
150	4	8	2.093	6
200	4	8	2.07	6

We can see from Table 1 and Table 2 that method 1 initializes a graph with larger connectivity since the graph has larger maximum degree as well as larger average degree. Thus, we need to use more colors if method 1 is used to initialize the graph since our premise is $\mathbf{q} > 2*\mathbf{d}$. Moreover, method 2 is more "stable" because the average degree and maximum degree remains almost unchanged when we increase the number of vertices. Therefore, we employ method 2 for the following experiments.

1.6.6 Discovery 1: Focusing on One Color

In this experiment, we wish to discover the average number of vertices which occupy the same certain color when running the Markov Chain. We denote n_{i_j} as the average number of vertices which occupy color i from the beginning to the jth iteration. The color index i is randomly picked from $\mathbf{Q}$ and it is trivial. Theoretically, the sequence $\{n_{i_j}\}$ converges to $\frac{n}{q}$. Since when we change q, $\frac{n}{q}$ will also change, it is necessary to adjust the sequences to $\{n_{i_j} - \frac{n}{q}\}$, which theoretically should converge to 0. Moreover, in order to reduce random error, for each setting of number of colors, we repeat the experiment for 50 times and use the average. As we have discussed before, we do the following two sets of experiments:

Change Total Number of Colors Basically, we try different number of colors on the same graph and the index of selected color is trivial. Firstly, we need to show that the Gibbs Sampler really produces a convergent result. I set the number of vertices as 150. The number of colors were $20, 25, 30, 35, 40$. I get the following plot in which the x-axis is iterations and the y-axis is $n_{i_j} - \frac{n}{q}$.

We can see from Figure 1 that no matter how many colors we use, ultimately they will converge. However, the difference in speed of convergence is not obvious. We need to enlarge the difference in the number of colors. Meanwhile, the sequences have converged after 15,000 iterations, thus, we are able to reduce the number of iterations in order to shorten the running time. We did the experiments as follows:

We can see from Figure 6 that there exists a tendency of faster convergence when we increase the number of colors, it seems to converge faster. However, the evidence is not obvious because there exists some cases in which we increase the number of colors while the chain converges slower. Since we only consider the number of vertices occupying one

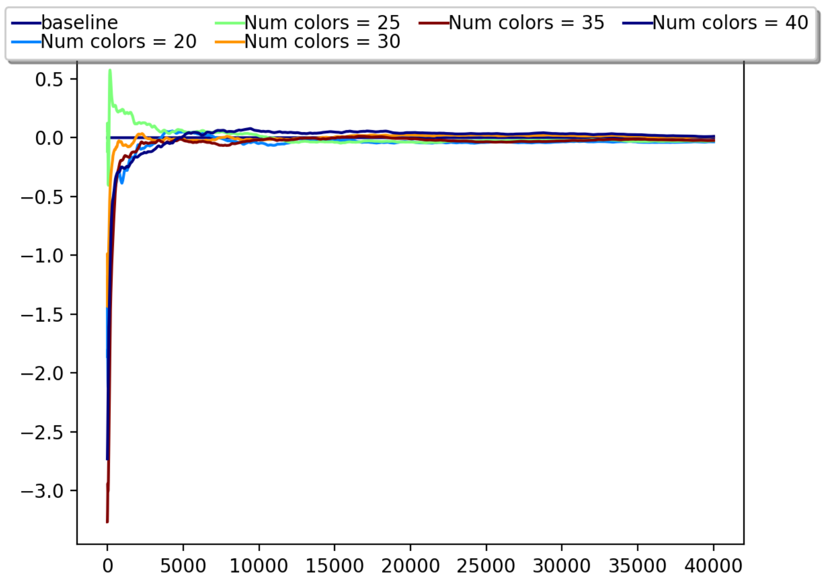


Figure 1: Convergence check

certain color, random error is likely to affect the result.

Change Connectivity Basically, we may add more edges to the existing graph so that the connectivity of the graph can be increased. If the conjecture in the previous section is true, i.e., more valid configurations, faster convergence, then when we increase the connectivity of the graph, there will be less valid configurations and the convergence should be slower. We conducted experiments on graphs with vertices $50, 100, 150, 200$. In each experiment, the number of edges that we add is proportional to the number of vertices. The plots are shown in Figure 11:

Roughly speaking, the light blue curves in Figures 7 to 10 which represent the initial graph converges faster than the others. When we add more edges to the graph, they converge slower in different levels. Therefore, our conjecture is right. The more edges we add to the graph, the less valid configurations and the slower the convergence. Similar to the set of experiments above regarding the number of colors, the results are not stable. There are also some cases in which graphs with larger connectivity converge faster than those with smaller connectivity. Thus, we need to further reduce the random error. Since we have only focused on one color, we turn to focusing on all colors in the following experiments.

1.6.7 Discovery 2: Focusing on all Colors

Now we focus on all colors. Theoretically, the expectation of the number of vertices which occupy any certain color equals to $\frac{n}{q}$. Thus, in this set of experiments, our idea is like this. We construct two vectors of length q, the first vector is called the expectation

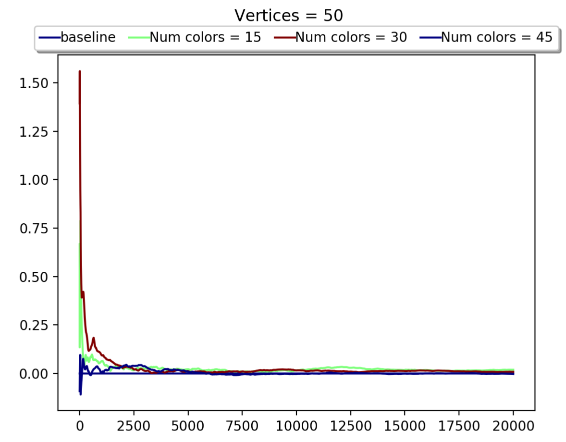


Figure 2: v=50

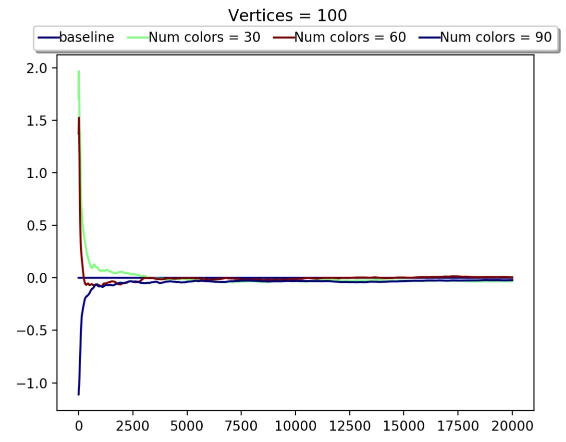


Figure 3: v=100

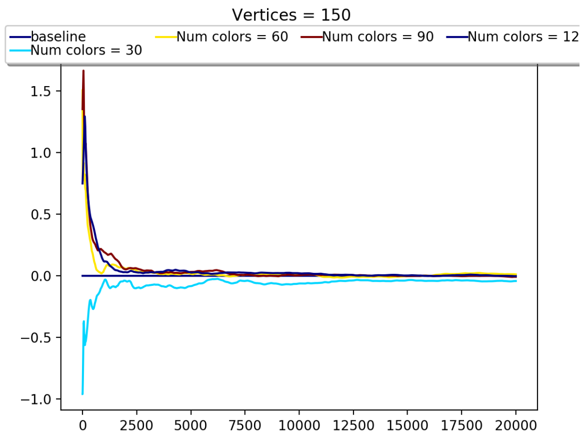


Figure 4: v=150

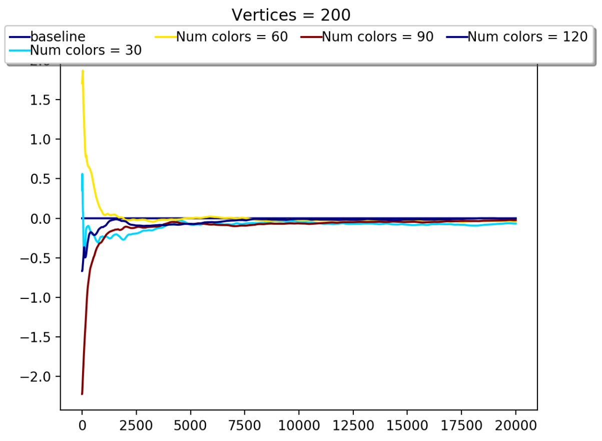


Figure 5: v=200

Figure 6: Change Total Number of Colors, Discovery 1, VCP

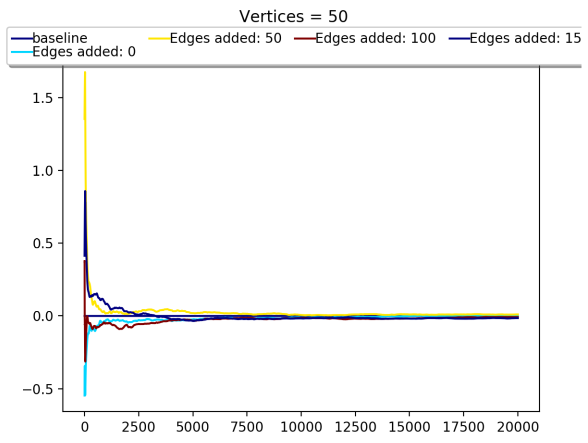


Figure 7: v=50

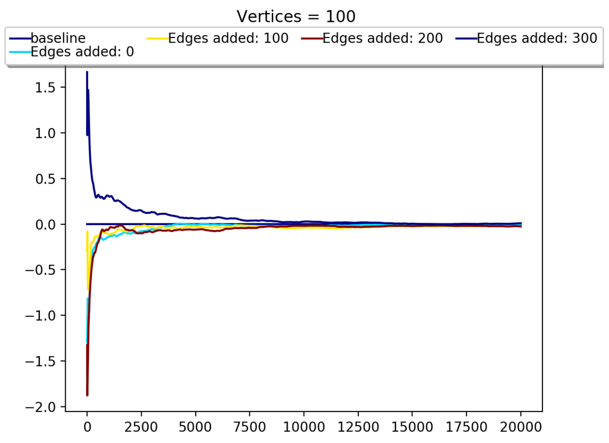


Figure 8: v=100

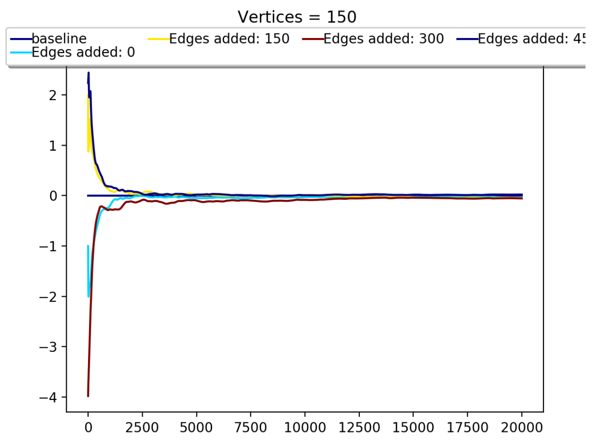


Figure 9: v=150

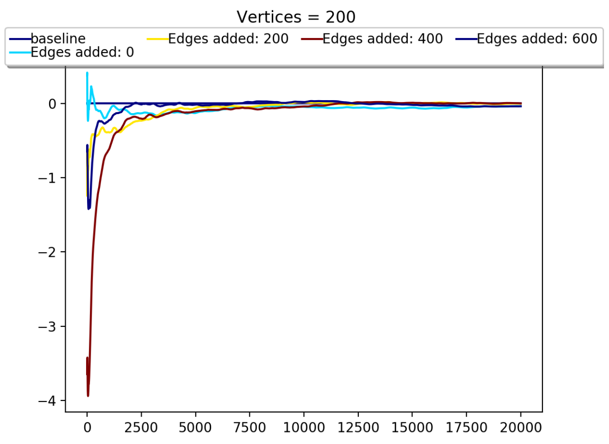


Figure 10: v=200

Figure 11: Change Connectivity, Discovery 1, VCP

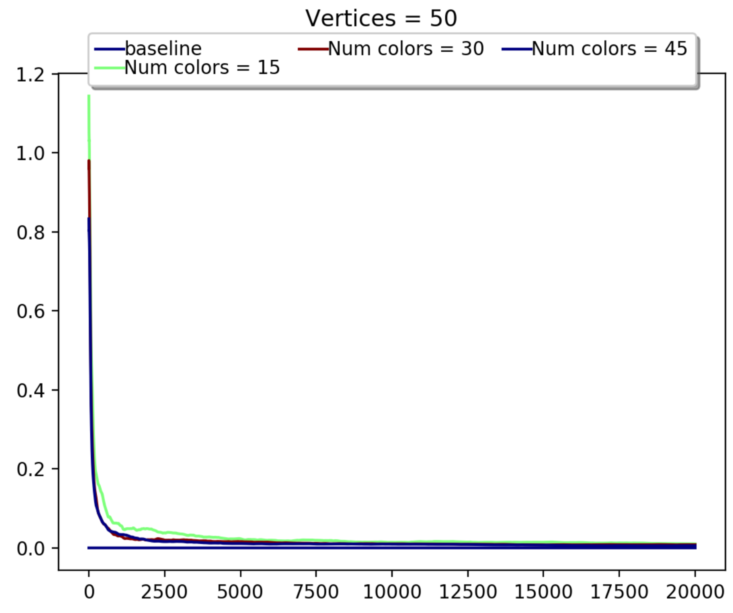


Figure 12: Convergence check

vector **E** in which each entry equals to $\frac{n}{q}$ and the second vector is called the reality vector $mathbfR$ in which the ith entry is the average number of vertices which occupy the color i from the beginning to the current iteration. Thus, R is updated in every iteration when running the Markov Chain. We use R^j to define the R vector in the jth iteration. We also use the average of Manhattan distance to measure the distance between these two vectors, which is $d_{ER^j} = \frac{1}{q} * \sum_{i=0}^{q-1} |E_i - R_i^j|$. Theoretically, the sequence $\{d_{ER^j}\}$ should converge to zero. The advantage of using d_{ER^j} instead of the average number of vertices which occupy a certain color is that it eliminates the random error of one single color. We do the same thing as what we have done in the previous discovery about focusing on one color.

Change Total Number of Colors I set the number of vertices as 50 and the try $15, 30, 45$ as the number of colors. I get Figure 12:

Figure 12 illustrates that:

1. After using d_{ER^j} to measure, the sequences become much more stable because the oscillation becomes much smaller than what we got in Discovery 1.

2. When we increase the number of colors, the sequence converges faster, which indicates the truth of our conjecture. However, the difference is not significant. We should increase the number of vertices so that we can enlarge the difference in number of colors.

Thus, we have to change the setting of vertices. Figure 17 shows the final results:

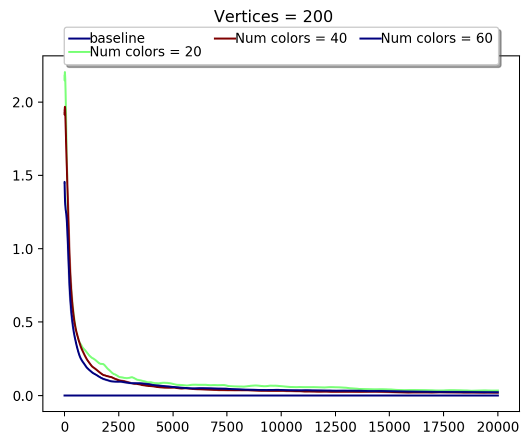


Figure 13: v=200

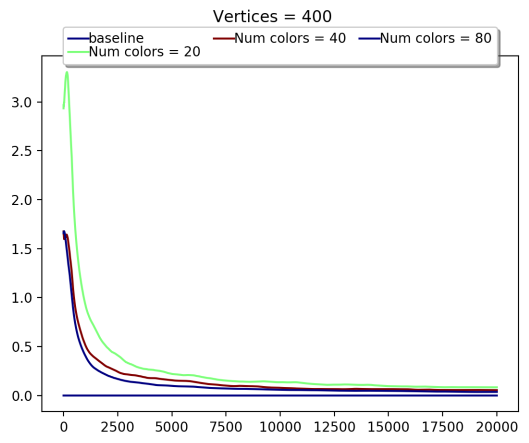


Figure 14: v=400

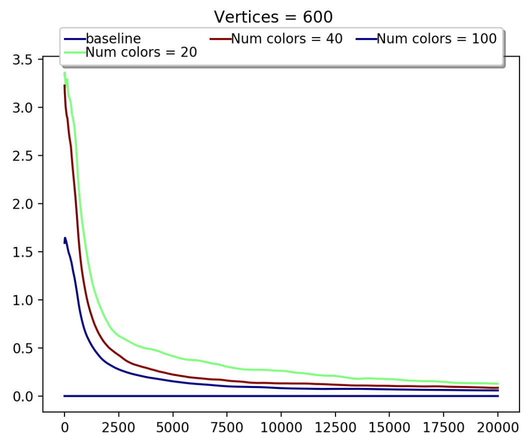


Figure 15: v=600

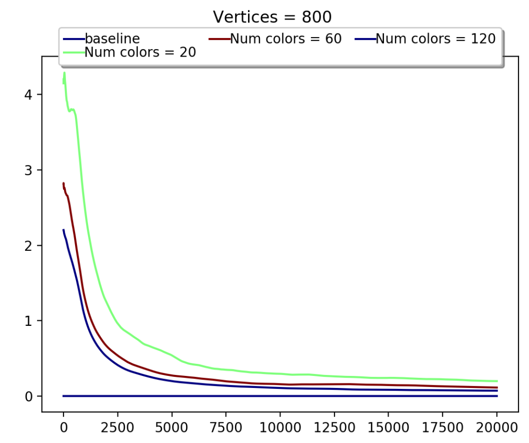


Figure 16: v=800

Figure 17: Changing Color in Four Graphs, Discovery 2, VCP

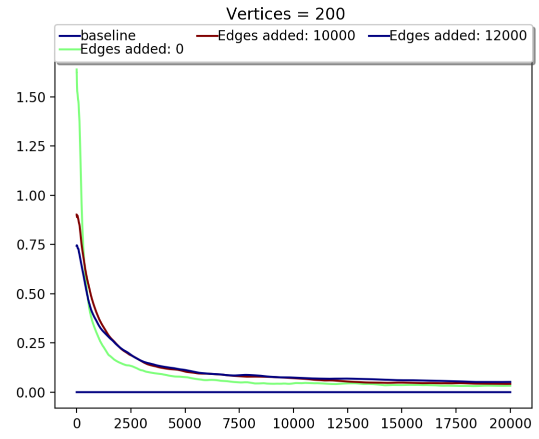


Figure 18: v=200

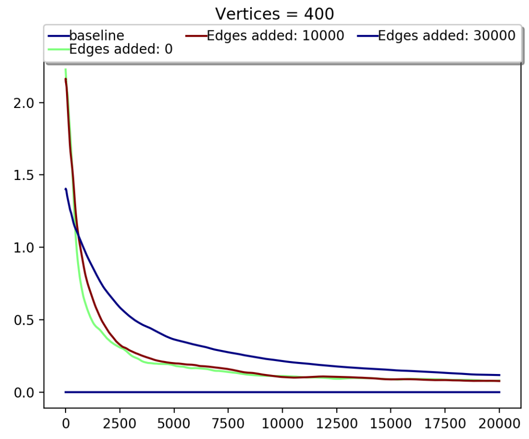


Figure 19: v=400

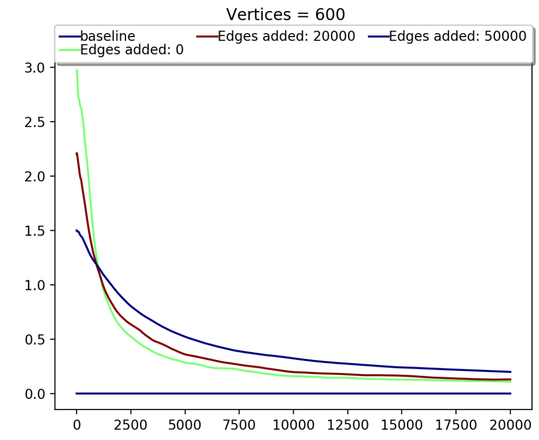


Figure 20: v=600

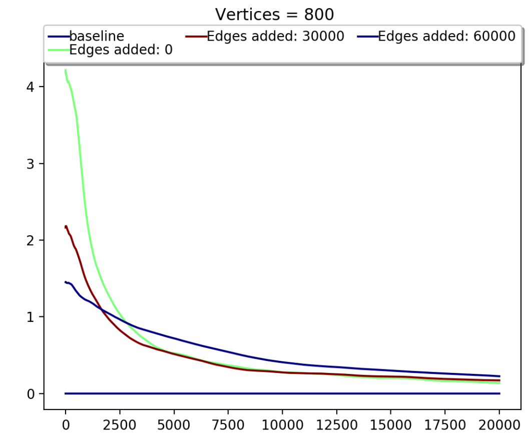


Figure 21: v=800

Figure 22: Changing Connectivity in Four Graphs, Discovery 2, VCP

The four plots in Figure 17 show the truthfulness of our conjecture clearer. On the one hand, the sequences are more stable and they all converge to 0 smoothly. On the other hand, in every plot, the larger the number of colors, the faster the sequence converges. Apart from changing the number of colors, we also need to test the result of changing connectivity.

Change Connectivity Note that the average degree of the graph generated by Method 2 is around 2, so the total number of edges in the initial graph is approximately the number of vertices. And also, the connectivity of the initial graph is quite low. Thus, after some preliminary experiments, we add edges over 10000 in order to get sequences with significant differences. We change the connectivity and get the results as shown in Figure 6:

From the plots in Figure 22 we can see that the sequences becomes much more

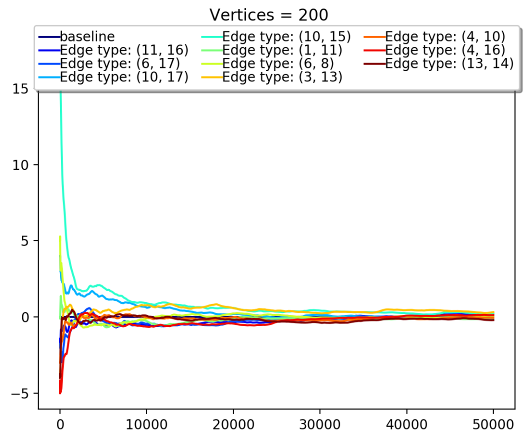


Figure 23: v=200

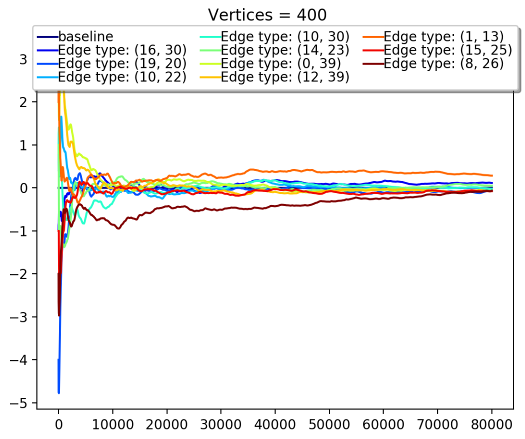


Figure 24: v=400

Figure 25: Focusing on Edges, Discovery 3, VCP

smooth because of the less oscillation. In every graph, the light green curve representing the sequences for the original graph converges fastest. The dark blue curve representing the graph with the largest connectivity converges the slowest.

Based on our experiment results in both changing the number of colors and changing the connectivity, we find that:

1. The more colors we use in the graph, the more valid configurations for the graph, the faster the chain converges.

2. The smaller the connectivity of the graph, the more valid configurations for the graph, the faster the chain converges.

Therefore, we have the following generalized rule: *No matter what factor do we change, as long as it can increase the total number of valid configurations, the sequence of the average number of vertices which occupy the same color converges faster and vice versa.*

1.6.8 Discovery 3: Focusing on edges

Now I change our focus to edges. This part dedicates to be a proof of the randomness of our former experiments. I define the type of an edge by the colors of the two vertices it connects. Then there should be $qC2$ types of edges. I propose a hypothesis that when running the Markov Chain for a large number of iterations, the average occurrences of each type of edge should be $\frac{e}{qC2}$ where e represents the total number of edges in the graph. Here I denote e_{i_j} as the average number of occurrences of edge type i from the first iteration to the jth iteration. For simplicity, I plot $\{e_{i_j} - \frac{e}{qC2}\}$. Then this sequence should converge to zero for any i. Moreover, since there are many types of edges, we just randomly choose 10 types of edges. I get Figure 25:

From the plots in Figure 25, we can see that for any type of edge, when running the Markov Chain, the average occurrences of it converges to $\frac{e}{qC2}$. Thus, our hypothesis is true. This to some extents, proves the randomness of Gibbs sampler.

1.7 Conclusion

In this section, we are interested in applying Gibbs Sampler on the Vertex Coloring Problem. Basically, Gibbs Sampler is one kind of MCMC algorithm which constructs an irreducible and aperiodic Markov Chain, and then generate samples based on the posterior distribution. The size of the transition probability matrix is always quite large and there is no need to know the exact size of the matrix. All we need to do is to have an initial state and transfer to the next state based on a certain probability distribution. With the help of Gibbs sampler, we can achieve high-dimensional sampling. After running the Markov Chain for a large number of iterations, we use the average of a property of the samples generated by the Markov process to simulate the expectation of such property.

By the ideology of MCMC theories, we have achieved our research objectives on vertices and edges. For vertices, we are interested in the average number of vertices which occupy the same color. After changing our focus from one certain color to all colors and take the arithmetic average of the absolute difference between the "hypothesized" expectation and the real average number of vertices in this color, we got smooth and convergent sequences. The chain converges faster when we increase the number of colors or decrease the connectivity. Thus, we can generalize a rule to state that, no matter what factor we change, as long as it increases the number of valid configurations, then the chain will converge faster and vice versa. For edges, results have shown that the average occurrences of each type of edge converge to $\frac{e}{qC2}$, and there is no significant difference between different types of edges.

2 Simulated Annealing on Travelling Salesman Problem

In this section, we turn to another randomized algorithm, the Simulated Annealing Algorithm. We use it get a comparatively optimal solution to the Travelling Salesman Problem (TSP). Furthermore, I have designed and tested two alternative approaches which aim to enhance the performance of traditional Simulated Annealing. Fortunately, one of them surpasses the traditional Simulated Annealing not only in the number of iterations required and the running time required before convergence.

2.1 Problem Background

Travelling Salesman Problem (TSP) is a famous NP-hard problem in combinatorial optimization. In this problem, we are given a list of cities and the distance between any two of them. We are required to find the shortest Hamilton path which starts from a city and visits each remaining city exactly once, and finally return to the origin. Actually, there are many researches about this problem in theoretical computer science and operations research.

The problem is equivalent to finding such a permutation with the least cost. Suppose the list consists of **n** cities. Then, there should be **n**! permutations in total. If by brute force, we have to compute the cost of all the **n**! paths and select the least one. The time complexity is of $O(n!)$, and there is no doubt for the in-feasibility of such method. Even if there are only 20 cities, there will be over 10^{18} paths. Therefore, we need more efficient methods to solve this problem. Up till now, various methods including deterministic algorithms and heuristic algorithms have been proposed. Even though the problem is computationally intense, some instances with even millions of cities can also be solved within 1% of the optimal solution [8]. In the next part, we will get a closer look into the method I used in this research, which was Simulated Annealing algorithm.

2.2 About Simulated Annealing

For those continuous and differentiable functions, gradient descent which updates the parameters in the model by subtracting the first order gradient of the objective function multiplied by a learning rate, is widely used in searching for the global minimum point [17] of continuous and differentiable functions. However, in TSP, since the search space is discrete, it is unsuitable to use the gradient-based methods.

Here we consider the Simulated Annealing method, which is based on MCMC. The idea of Simulated Annealing comes from annealing in condensed matter physics, a technique which involves heating up a solid by rising temperature to a maximum value so that every particle will arrange themselves randomly followed by a process of controlled cooling to the solid. As long as the initial temperature is sufficiently high and the annealing schedule is slow enough, all particles would be in the state of low energy of the corresponding lattice [1]. It is a probabilistic technique to approximate the global optima of discrete functions with a large search space. Similar to its origin in matter physics, the elements in the search space are called states. In TSP, a specific Hamilton path is a state and we need to design a transition mechanism for a Markov Chain to search the domain of all states. In order to find the path with smallest cost after a number of iterations, the Markov Chain should have a unique stationary distribution in which most of the probabilities are placed in states having small value of cost. In order to achieve this, we need to define $f_T(x)$, a set of densities which is proportional to $[f(x)]^{1/T}$ where the parameter T defines a temperature. In the kth iteration, we use MCMC sampling to draw a sample $\mathbf{X}^k$ from the search space based on the probability distribution $f_{Tk}(x)$ [16]. In order to make the distribution have a sharper peak at the global maxi-ma, we should gradually reduce the temperature as the Markov Chain is running. Thus, every time we reduce the temperature, a new Markov Chain starts. The design of the Markov Chain and the setting of temperature are of vital importance.

2.2.1 Design of Markov Chain

The MCMC method works if and only if the Markov Chain is irreducible and aperiodic. As for TSP, our objective is to find a permutation of the cities such that the total cost is the lowest. Hence, we treat each permutation of the cities as one state of the Markov Chain. Then this Markov Chain has $\mathbf{n}!$ states, which makes the transition probability matrix really large. Similar to the previous section, we do not need to use the whole matrix as we run the Markov Chain. What we need to do is to define "neighboring" states, establish the stationary distribution and determine the transition probability.

Definition of Neighborhood There are several ways to define neighboring states in TSP. Here I have implemented two kinds of neighborhood:

1. **Reverse:** For any two states m and n, if there exist two positions i and j such that n can be got by reversing the sub-sequence of m from i to j while the other positions remain the same, then and m and n are neighbors to each other.
2. **Swap:** For any two states m and n, if there exist two positions i and j such that n can be got by swapping the city at position i and the city at position j in m, then m and n are neighbors to each other.

Stationary Distribution Since we wish to put more probability mass on those states with a lower cost, suppose the state space is $\mathbf{S}$ and the cost function is $f : \mathbf{S} \to \mathbf{R}$. The cost function measures the total distance of the path regarding to the permutation. We need to find a distribution in which the smaller the cost, the larger the probability mass. Moreover, as we have stated before, most of the probability should be placed in the optimal state as we reduce the temperature. Thus, the general idea of designing such stationary distribution comes from Boltzmann distribution. The Boltzmann distribution $\pi_{f,T}$ satisfies $\pi_{f,T}(s) = \frac{1}{Z_{f,T}} * \exp(\frac{-f(s)}{T})$ for any $s \in \mathbf{S}$ where $Z_{f,T} = \sum_{s \in \mathbf{S}} \exp(\frac{-f(s)}{T})$ [16]. In this way, the smaller the cost, the larger the probability will be. Moreover, as the temperature tends to zero, we want the probability mass of the global minimum point tends to 1. The proof is as follows:

Without the loss of generality, suppose the state s_1 is the unique global minimum point of the function f and denote $\mathbf{S'} = \mathbf{S} \setminus \{s\}$. We also suppose that s_2 satisfies $f(s_2) = min_{s\in\mathbf{S'}} f(s)$. Let $a = f(s_1)$ and $b = f(s_2)$. Then we have:

$$\pi_{f,T}(s_1) = \frac{1}{Z_{f,T}} * \exp(\frac{-f(s)}{T}) = \frac{\exp(\frac{-a}{T})}{\exp(\frac{-a}{T}) + \sum_{s'\in\mathbf{S'}} exp(\frac{-f(s')}{T})} \geq \frac{exp(\frac{-a}{T})}{\exp(\frac{-a}{T}) + |\mathbf{S'}| \exp(\frac{-b}{T})}$$

Note that the last fraction converges to 1 as T tends to 0. Therefore, as we gradually reduce the temperature, the probability mass of the global minimum point tends to 1. Therefore, in order to find such s_1, we can construct a Markov Chain to simulate the Boltzmann distribution $\pi_{f,T}$ on $\mathbf{S}$.

Transition Probability As I have stated before, the temperature is a parameter in computing the transition probability in the Markov Chain. Hence, the transition probability matrix should also change as the temperature changes. Therefore, the whole algorithm can be viewed as an in-homogeneous Markov Chain in which there are **N** Markov Chains running one by one. They have the same set of states but different transition probability matrices. Let the kth Markov Chain has transition probability matrix $\mathbf{P_k}$ and it runs for N_k iterations. Now that I have defined the neighborhood in two ways, the next problem is the transition probability. According to the idea in the construction of Metropolis chain, given the stationary distribution, the transition probability can be computed as the following:

$$P_{i,j} = \begin{cases} \frac{1}{d_i} * min\{\frac{\pi_j d_i}{\pi_i d_j}, 1\} & if\ s_i\ s_j\ are\ neighbors \\ 0 & if\ s_i\ \neq\ s_j\ and\ they\ are\ not\ neighbors \\ 1 - \sum_{s_l \in I} \frac{1}{d_i} * min\{\frac{\pi_l d_i}{\pi_i d_l}, 1\} & if\ i = j \end{cases}$$

Note that I is the set of the neighbors of s_i and d_i is the cardinality of I. In the third case, we sum over all the neighbors of s_i [16].

Generally, people construct the Markov Chain by the method of Metropolis chain when implementing Simulated Annealing. Based on this, I propose two alternative ways to define the transition probability. The three methods are listed as the following:

1. We directly construct a Metropolis chain. Here we should note that for any state s, there are $nC2$ ways to choose the two cities for reversing or swapping. Thus, the number of neighbors for any state s should be $nC2$ so we can easily cancel the d_i and d_j which are the number of neighbors of the two states as we have stated previously. Moreover, the term $Z_{f,T}$ can also be cancelled, which frees us from computing $\exp(-\frac{f(s)}{T})$ for the whole state space. The following is the formula of the transition probability from state s_i to s_j.

$$P_{i,j} = \begin{cases} \frac{2}{n(n-1)} * min\{\exp(-\frac{f(s_j)-f(s_i)}{T}), 1\} & if\ s_i\ s_j\ are\ neighbors \\ 0 & if\ s_i\ s_j\ are\ not\ neighbors \\ 1 - \sum_{s_l\in I} \frac{2}{n(n-1)} * min\{\exp(-\frac{f(s_l)-f(s_i)}{T}), 1\} & if\ i = j \end{cases}$$

Therefore, we can deduce the transition mechanism. Suppose the current state is s_i. We first uniformly select two distinct positions from 1 to n so that we can generate a neighbor of the current state s_j. Then if $f(s_j) < f(s_i)$, we transit directly. Otherwise, we uniformly generate a random number y between 0 and 1. If $y < \exp(-\frac{f(s_j)-f(s_i)}{T})$, we transit to s_j. Otherwise, we stay in state s_i.

2. In the first method, we just randomly generate a neighbor of the current state and determine whether to transit. The probability of getting any neighboring state is the same no matter whether the cost is smaller or not. Therefore, in this method, I give more weights to those neighboring states with a smaller cost, which will increase the probability to transit into a state with a smaller cost. Suppose we assign weight ρ to the states with smaller costs. Empirically, ρ is set to 0.9. It means that the sum of the probabilities to generate states with smaller costs equals to ρ. Thus, the sum of the probabilities to generate states with larger costs equals to $1-\rho$. In practice, we uniformly generate a random number between 0 and 1. If the number is smaller than ρ, then we keep on generating neighboring states until we get a state with a smaller cost and transit into this new state. Otherwise, we keep on generating neighboring states until we get a state with a larger cost. The transition probability would then be $\exp(-\frac{f(s_j)-f(s_i)}{T})$. Therefore, we again uniformly generate a number between 0 and 1. If the number is smaller than the transition probability, we transit to the new state s_j. Otherwise, we stay in s_i. The transition probability is defined as follows:

$$P_{i,j} = \begin{cases} \frac{\rho}{l_i} & if\ s_i,\ s_j\ are\ neighbors\ and\ f(s_j) < f(s_i) \\ \frac{1-\rho}{h_i}\exp(-\frac{f(s_j)-f(s_i)}{T}) & if\ s_i\ s_j\ are\ neighbors\ and\ f(s_j) \geq f(s_i) \\ 0 & if\ s_i\ s_j\ are\ not\ neighbors \\ 1-\rho-\sum_{s_h\in\mathbf{H_i}}\frac{1-\rho}{h_i}\exp(-\frac{f(s_h)-f(s_i)}{T}) & if\ s_i = s_j \end{cases}$$

Note that l_i represents the number of neighboring states having smaller cost and h_i represents the number of neighboring states having larger cost. $\mathbf{H_i}$ is the set of neighboring states with larger cost.

3. Now that we have came up with the idea to assign larger weights to those neighboring states with smaller cost, we can further consider assigning even more weights to those neighboring states which reduce more cost. In this way, the more cost a state can reduce, the more likely it will be picked. Inspired by the softmax function in Machine Learning, this method modifies the previous one by assigning weights to each neighboring states according to how much it reduces or increases the cost. Therefore, the idea is as the following. We also assign a weight ρ to the neighboring states with smaller cost and $1-\rho$ to the other neighboring states. Then we uniformly generate a random number $x \sim U(0,1)$. If $x < \rho$, we generate a set of sample neighbors having smaller costs. Otherwise, we generate a set of sample neighbors having larger costs. Then we assign weights to the set of sample neighbors based on their cost. Since there exist some tricks in weights assignment, more details will be provided in the part of experiment design.

2.2.2 Annealing

The temperature is one of the most important hyper-parameters in applying Simulated Annealing algorithm on TSP. It determines the probability of transition to the new state if the new state has a larger cost than the older one. According to the formula, given that the generated state is s_j and it has a larger cost than the old state s_i, the transition probability should be $\exp(-\frac{f(s_j)-f(s_i)}{T})$. This positive probability has distinguished the Simulated Annealing algorithm with the Greedy algorithm in which the local optimal is chosen in every step. The Greedy algorithm has a severe drawback. It may get stuck in the local minimum if every time the chain either transits to a state with smaller cost or stays unchanged. In TSP, there might also exist some local optimal states which have the smallest cost among all their neighbors. Greedy algorithm will never get out of such states once getting in and the chain will never converge to the global optimal.

Thus, the probability of "temporarily getting into a worse state" plays an important role in the optimization process. From the formula we can see that the smaller the temperature T, the smaller the transition probability. As the in-homogeneous chain runs, the temperature should be gradually reduced. At the beginning, the temperature is usually set high and it enables the chain to avoid being stuck in the local minimum. After many iterations, the chain converges and the comparatively lower temperature fixes the chain at the neighborhood of the global minimum. Previous researches have verified that if T approaches 0 sufficiently slowly, then the probability for the chain to get to the global minimum will tend to 1 as the number of iterations goes to + [7]. The trade-off is that, on the one hand, we need to reduce the temperature quicker in order to get the result in a reasonably short time. On the other hand, we need to reduce the temperature slower to avoid local optimal.

Empirically, the initial temperature is relevant to the difference of cost among different permutation. For example, if both measured by kilometers, the initial temperature for the case in which all the cities are within one province must be different from the case in which the cities are from different continents. As for the annealing schedule, although some theorems state that sufficiently slow annealing guarantees the convergence to the global minimum, in practice, we have to use faster annealing otherwise the algorithm will take quite long time. In this research, for simplicity, I update the temperature by multiplying it with a discount factor, e.g., 0.9 after certain number of iterations. This number should also be adjusted for each of the three transition mechanism as I have stated above because in practice, the second and third transition method takes far more time for one single iteration than the first one. We will delve deeper into this in the next part about experiment design.

2.3 Verification

2.3.1 Graph Initialization

When it comes to the graph for the cities, all we need is an adjacency matrix which demonstrates the distance between any two cities. For simplicity, we can generate distances randomly rather than using real-world data. However, for the distance between each pair of cities, we cannot just uniformly generate a number between 0 and a positive number, i.e. the maximum distance between any two cities, because the graph may violate triangular inequality. (i.e. distance between A & B $>$ distance between A & C + distance between B & C) Therefore, I initialize the graph in the following steps:

1. Set a distance d. d is actually the length of a square which covers all the cities.
2. Uniformly generate two independent numbers between 0 and d. Set these two numbers as the xy-coordinates of a city.
3. Repeat the second step for $N - 1$ more times.
4. Compute the distance between any two cities.

2.3.2 Experiment Design

Now that we have had two definitions for neighborhood and three ways of transition, there are six combinations and we need experiments to rank the superiority. Basically, experiments should be designed to show from two perspectives: whether and where the Markov chain converges, how fast does the chain converge. The first can be evaluated by plotting the cost of the current permutation after each iteration. Ideally, after a sufficient number of iterations, the temperature T will tend to 0 and the chain will no longer transit. As for the speed of convergence, it can further be evaluated from two perspectives including the number of iterations before convergence and the running time until convergence. This cannot be evaluated by simply regarding to the plot. Thus, here we need one more clarification about the judgement of convergence. In this research, we use the ceiling of the arithmetic average of the last 10 costs as the threshold of convergence. The chain has converged at some iteration if and only if the costs of the following 100 permutations are all smaller than the threshold.

Therefore, in general, the evaluation can be divided into four parts. Their contents are shown below. Note that the "Reverse" or "Swap" inside the parenthesis defines the neighborhood.

1. (Reverse) Evaluate the relationship between the cost of permutation and number of iterations.
2. (Swap) Evaluate the relationship between the cost of permutation and number of iterations.
3. (Reverse) Evaluate the relationship between the cost of permutation and running time.
4. (Swap) Evaluate the relationship between the cost of permutation and running time.

Before starting the experiments, we still need to clarify some details about the algorithm implementation. In the second and third transitions, the cost of candidate permutations must in accordance with a certain principle. That is, all have smaller costs or all have larger costs. Especially in the third one, since we are not able to consider all the permutations having smaller or larger cost than the current permutation, we need to generate a set of samples. However, as the Markov Chain runs, it will become more and more difficult to find neighboring states with smaller costs. Hence, every time we are trying to search for a neighboring state with a smaller cost, the chain will stay in the same state if the we fail to search such neighboring state after 1000 trials. Otherwise, if the chain has entered into a local mini-ma, then there is no chance to find a neighboring state with a smaller cost, so the chain is not able to continue.

2.3.3 Experiments

In every part, we validate by applying the algorithm to graphs with $50, 100, 150, 200$ cities respectively. After generating the four graphs, we store them into files and no longer generating new ones in the future. As for the plots, since the second and third transitions take much fewer iterations to converge than the first one, it is necessary to plot the second and third transitions individually.

In order to get the best results for each of the transitions, I tune the hyper-parameters including the initial temperature and annealing schedule before comparison.

For reference, since the Markov Chain will converge to the global mini-ma if the annealing is sufficiently slow. Thus, I set a slow annealing schedule and ran for over $100,000$ iterations before the experiment for the four graphs in order to get the approximate global minimum cost beforehand. Note that this is just for reference in case that the error is too large during the experiments, not exactly the global minimum. Actually, we have no way other than the brute force method to get the real global minimum.

Since the convergent point may differ even if I repeat the same experiment, I use the **multiprocessing** package in Python to repeat the experiment and use the one with the smallest convergent point.

Part 1 In this part, two states are neighbors if and only if there exists a sub-sequence such that one can be obtained by **reversing** the sub-sequence of the other. We wish to find the relationship between the cost of current permutation and the **number of iterations**. The key hyper-parameters are listed below:

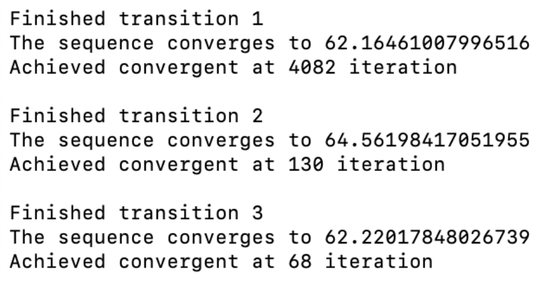
Here are the experiment results I got. Note that for each case, in terms of number of transitions, Transition 1 > Transition 2 > Transition 3, I plot Transition 2 and 3 individually. The first is the result printed on the terminal. It indicates where the Markov Chain converges and how many iterations it take.

For each graph, I first show the numerical results which include the point that the sequence converges to and the number of iteration at which the sequence achieved convergence. Then I show the combined result and isolated results for Transition 2 and 3.

The results in Figures 35, 40, 45 illustrate the followings:

Table 3: Key Hyper-parameters for Part 1

Vertices	Transition	Initial Temperature	Annealing Schedule	Max Iteration
50	T1	1	300	10000
50	T2	1	30	300
50	T3	1	20	150
100	T1	2	200	30000
100	T2	2	30	600
100	T3	2	20	250
150	T1	5	1000	60000
150	T2	2	50	1000
150	T3	2	40	320
200	T1	2	1000	100000
200	T2	2	50	1500
200	T3	2	40	500

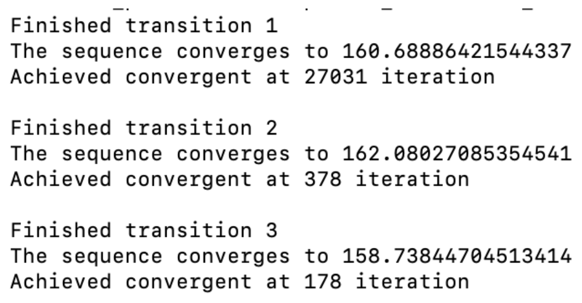


```
Finished transition 1
The sequence converges to 62.16461007996516
Achieved convergent at 4082 iteration

Finished transition 2
The sequence converges to 64.56198417051955
Achieved convergent at 130 iteration

Finished transition 3
The sequence converges to 62.22017848026739
Achieved convergent at 68 iteration
```

Figure 26: v=50

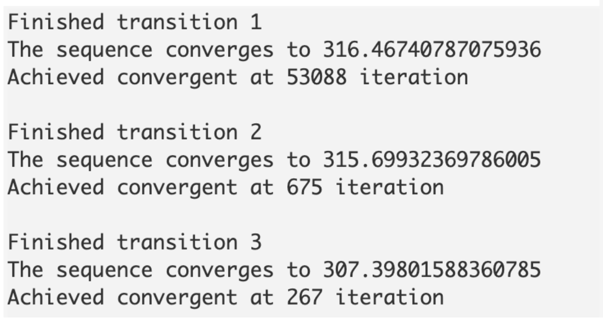


```
Finished transition 1
The sequence converges to 160.68886421544337
Achieved convergent at 27031 iteration

Finished transition 2
The sequence converges to 162.08027085354541
Achieved convergent at 378 iteration

Finished transition 3
The sequence converges to 158.73844704513414
Achieved convergent at 178 iteration
```

Figure 27: v=100

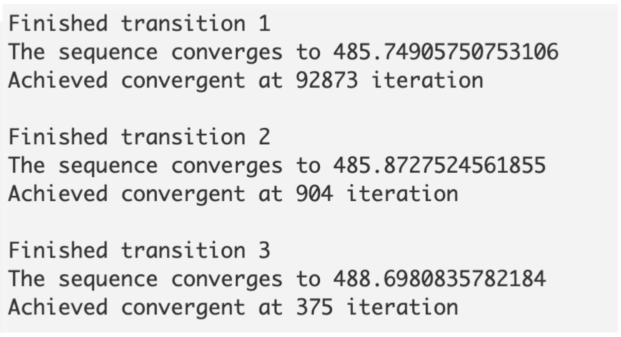


```
Finished transition 1
The sequence converges to 316.46740787075936
Achieved convergent at 53088 iteration

Finished transition 2
The sequence converges to 315.69932369786005
Achieved convergent at 675 iteration

Finished transition 3
The sequence converges to 307.39801588360785
Achieved convergent at 267 iteration
```

Figure 28: v=150



```
Finished transition 1
The sequence converges to 485.74905750753106
Achieved convergent at 92873 iteration

Finished transition 2
The sequence converges to 485.8727524561855
Achieved convergent at 904 iteration

Finished transition 3
The sequence converges to 488.6980835782184
Achieved convergent at 375 iteration
```

Figure 29: v=200

Figure 30: Numerical Results, Part 1, Simulated Annealing

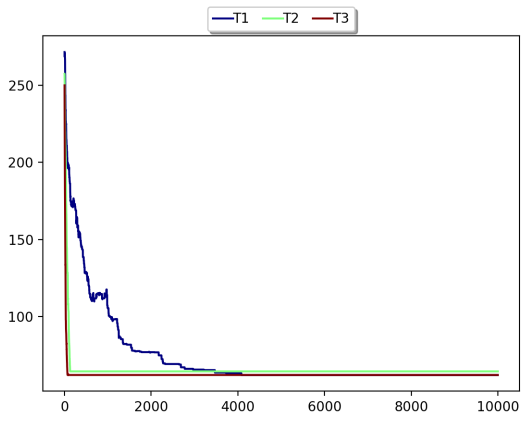


Figure 31: v=50

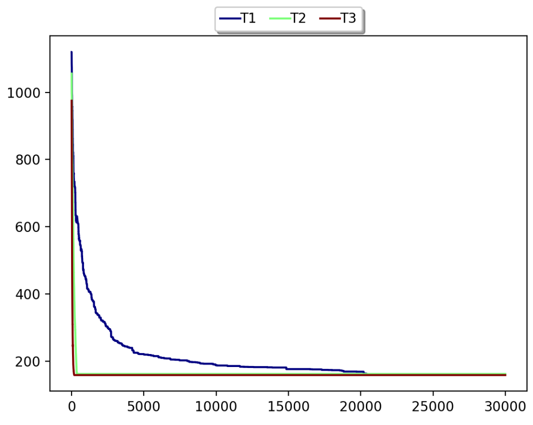


Figure 32: v=100

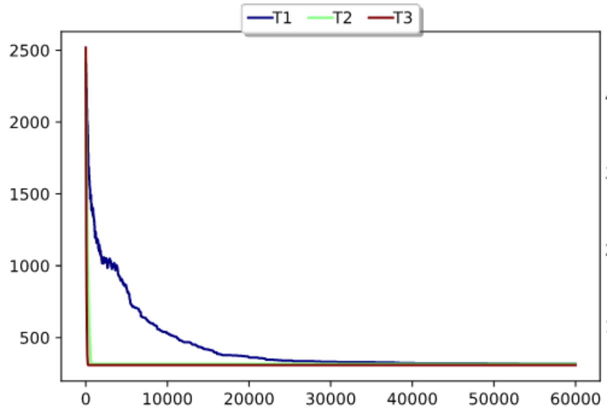


Figure 33: v=150

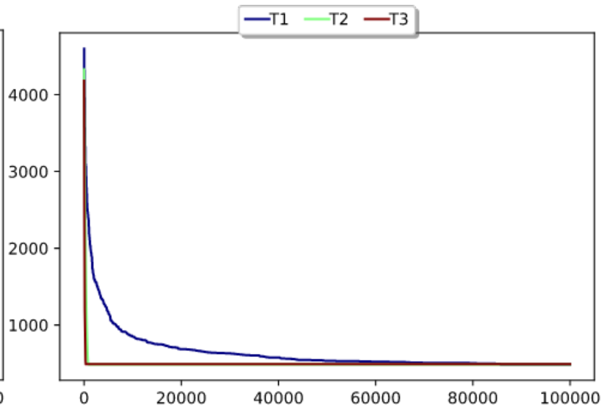


Figure 34: v=200

Figure 35: Combined Results, Part 1, Simulated Annealing

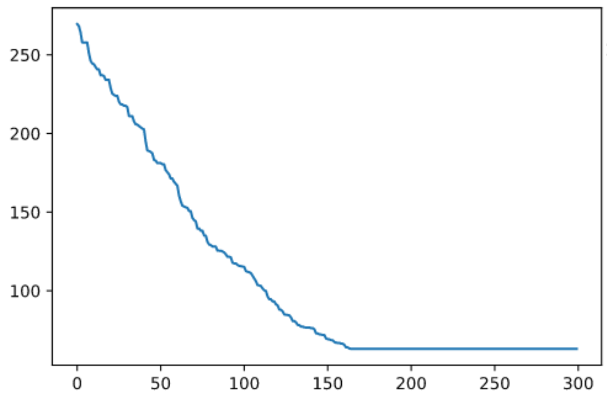


Figure 36: v=50

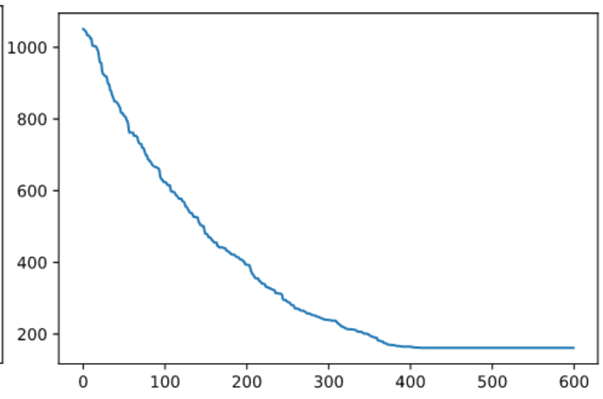


Figure 37: v=100

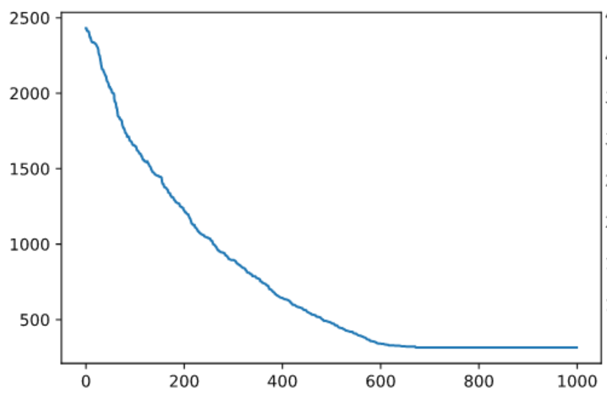


Figure 38: v=150

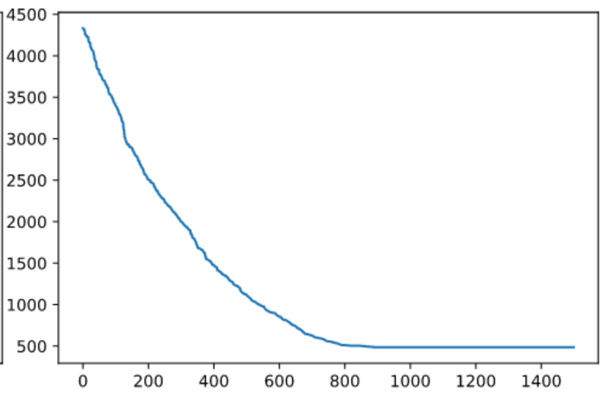


Figure 39: v=200

Figure 40: Isolated Results for Transition 2, Part 1, Simulated Annealing

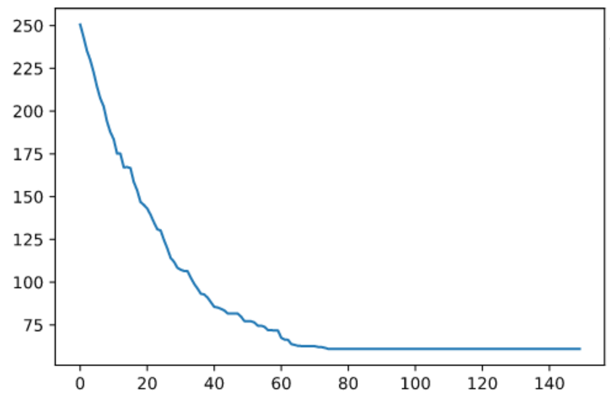


Figure 41: v=50

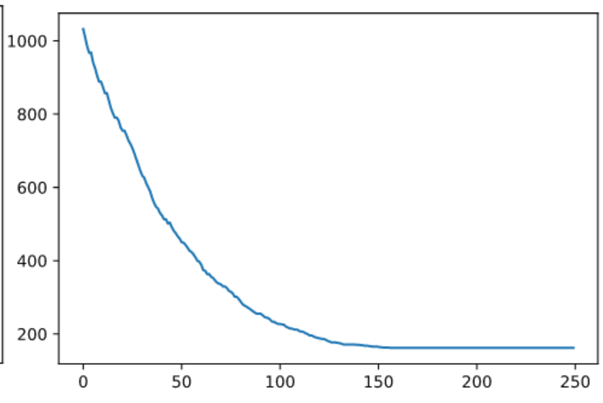


Figure 42: v=100

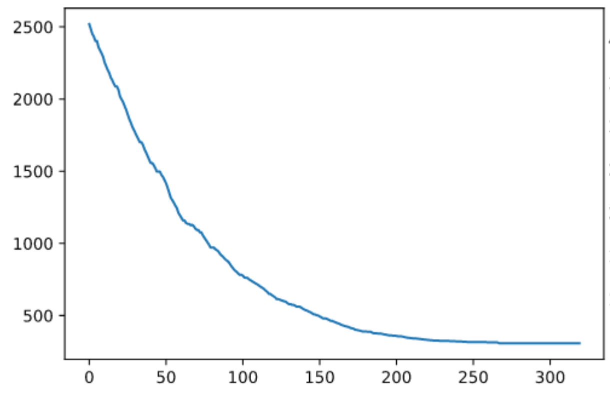


Figure 43: v=150

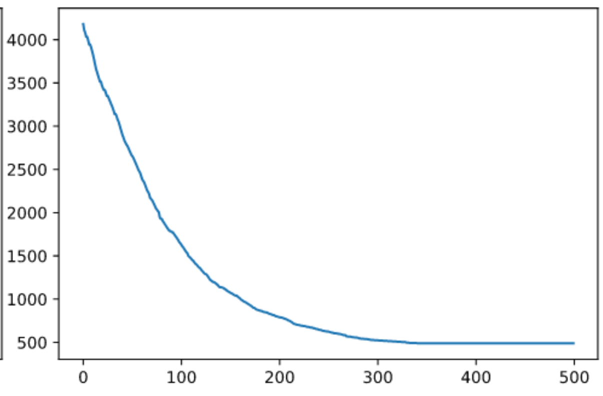


Figure 44: v=200

Figure 45: Isolated Results for Transition 3, Part 1, Simulated Annealing

Table 4: Key Hyper-parameters for Part 2

Vertices	Transition	Initial Temperature	Annealing Schedule	Max Iteration
50	T1	2	1000	20000
50	T2	1	50	600
50	T3	1	10	150
100	T1	5	1500	50000
100	T2	4	40	1200
100	T3	1	10	600
150	T1	5	3000	13000
150	T2	2	100	2500
150	T3	1	20	500
200	T1	5	3500	140000
200	T2	2	150	3000
200	T3	1	30	550

1. The number of iterations required before convergence: Transition 1 >> Transition 2 > Transition 3. Time required for one iteration: Transition 3 >> Transition 2 > Transition 1.

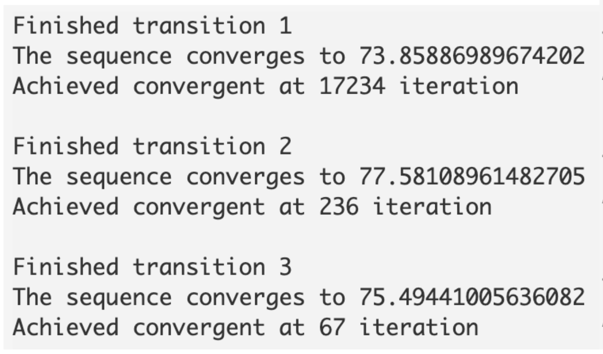
2. With proper hyper-parameter adjustment, all of the three transitions converge to approximately the same cost.

3. Transition 2 and Transition 3 are more "hyper-parameter neutral". It means that when changing the number of vertices, we do not need to change the initial temperature and annealing schedule too much. However, for Transition 1, we have to make more changes. This illustrates that Transition 2 and Transition 3 might be more useful because it takes less time to tune the hyper-parameters.

Part 2 In this part, two states are neighbors if and only if one can be obtained by **swapping** the positions of two cities in the other. We wish to find the relationship between the cost of current permutation and the **number of iterations**. The key hyper-parameters are listed below:

The experiment results are shown in Figures 55, 60, 65, 70.

The number of iterations needed before convergence satisfy: Transition 1 > Transition 2 > Transition 3. Through arduous adjustment to the hyper-parameters, I find that the annealing schedule should be more "mild" in swapping than reversing. In other words, swapping takes more iterations and a slower annealing process before convergence. Meanwhile, the convergence point of swapping is larger than reversing. Hence, reversing is more eligible to define neighborhood.

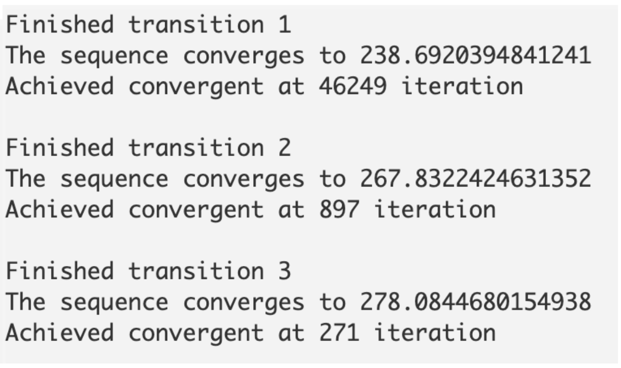


```
Finished transition 1
The sequence converges to 73.85886989674202
Achieved convergent at 17234 iteration

Finished transition 2
The sequence converges to 77.58108961482705
Achieved convergent at 236 iteration

Finished transition 3
The sequence converges to 75.49441005636082
Achieved convergent at 67 iteration
```

Figure 46: v=50

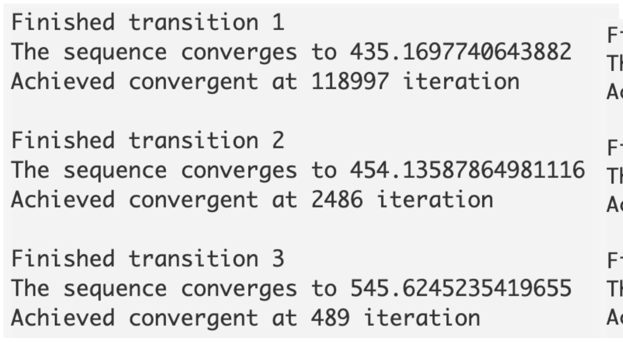


```
Finished transition 1
The sequence converges to 238.6920394841241
Achieved convergent at 46249 iteration

Finished transition 2
The sequence converges to 267.8322424631352
Achieved convergent at 897 iteration

Finished transition 3
The sequence converges to 278.0844680154938
Achieved convergent at 271 iteration
```

Figure 47: v=100

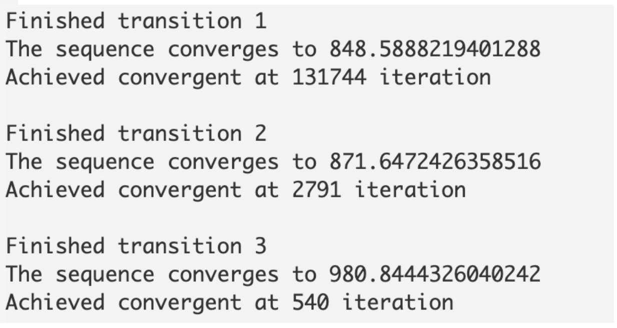


```
Finished transition 1
The sequence converges to 435.1697740643882
Achieved convergent at 118997 iteration

Finished transition 2
The sequence converges to 454.13587864981116
Achieved convergent at 2486 iteration

Finished transition 3
The sequence converges to 545.6245235419655
Achieved convergent at 489 iteration
```

Figure 48: v=150



```
Finished transition 1
The sequence converges to 848.5888219401288
Achieved convergent at 131744 iteration

Finished transition 2
The sequence converges to 871.6472426358516
Achieved convergent at 2791 iteration

Finished transition 3
The sequence converges to 980.8444326040242
Achieved convergent at 540 iteration
```

Figure 49: v=200

Figure 50: Numerical Results for Part 2, Simulated Annealing

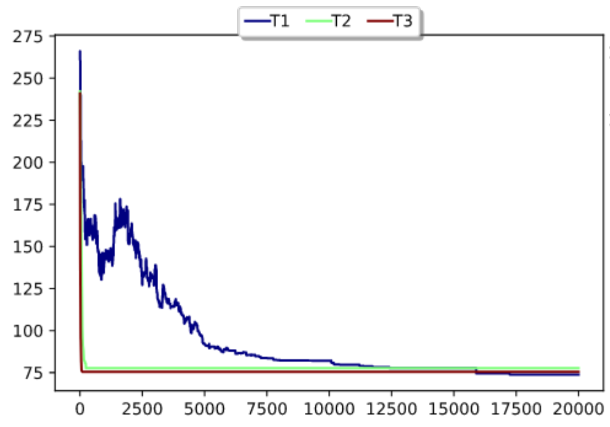


Figure 51: v=50

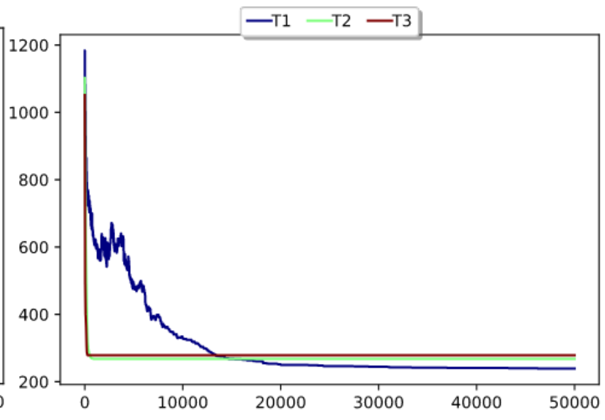


Figure 52: v=100

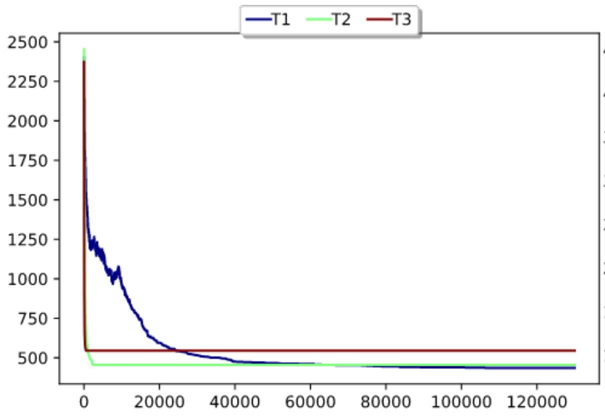


Figure 53: v=150

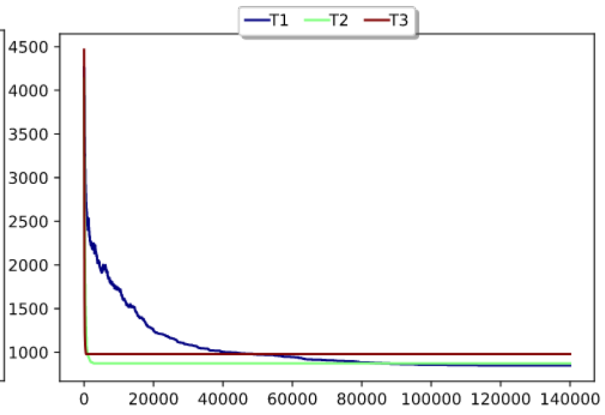


Figure 54: v=200

Figure 55: Combined Results for Part 2, Simulated Annealing

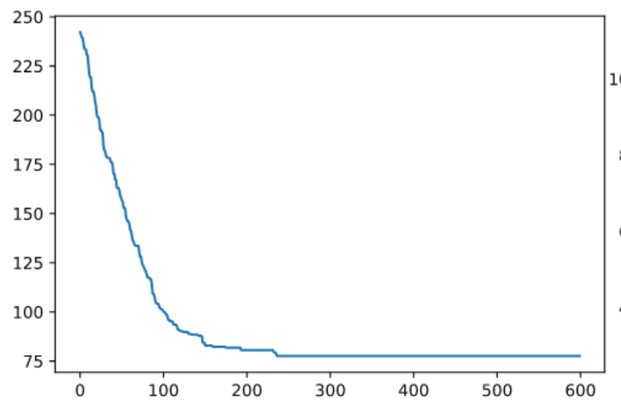


Figure 56: v=50

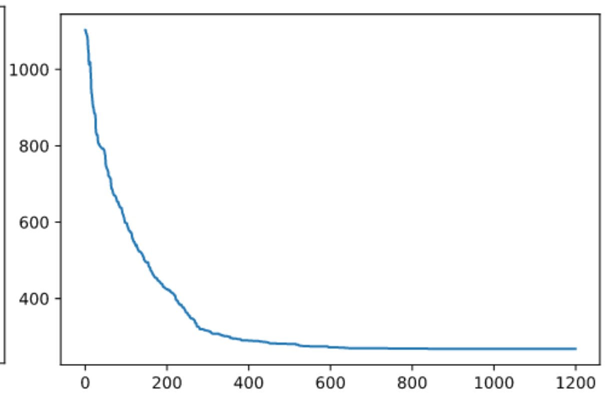


Figure 57: v=100

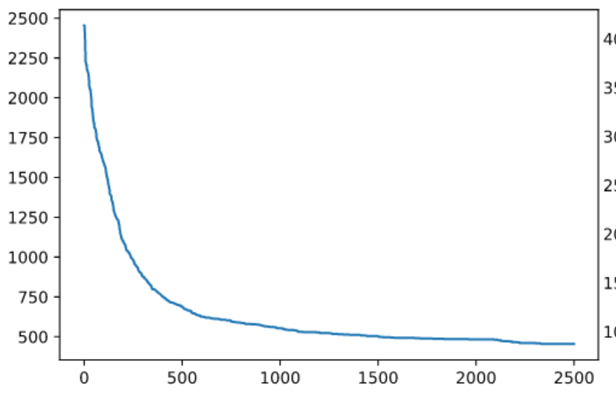


Figure 58: v=150

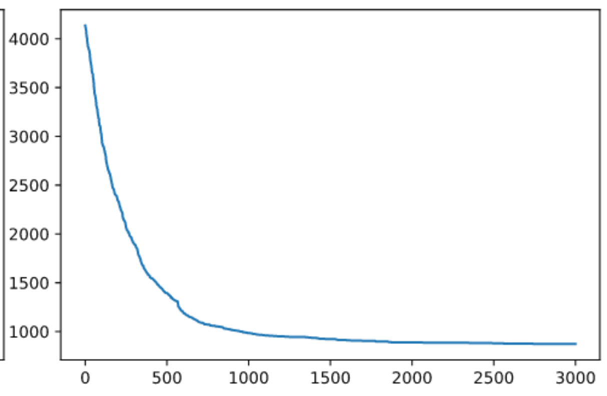


Figure 59: v=200

Figure 60: Isolated Results for Transition 2, Part 2, Simulated Annealing

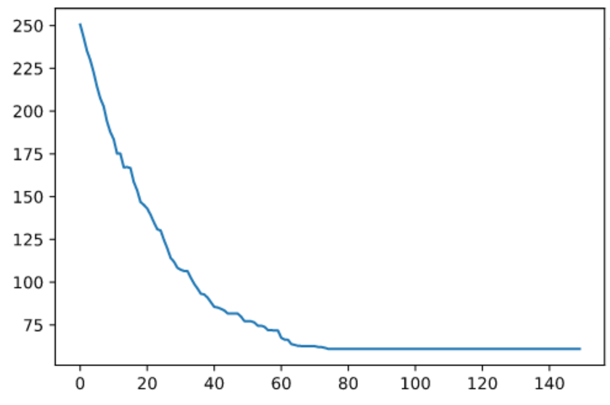


Figure 61: v=50

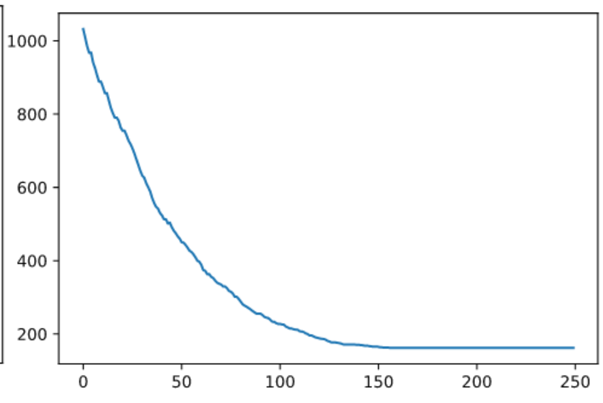


Figure 62: v=100

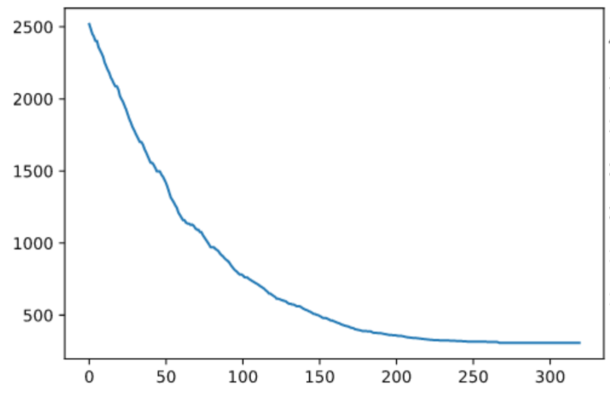


Figure 63: v=150

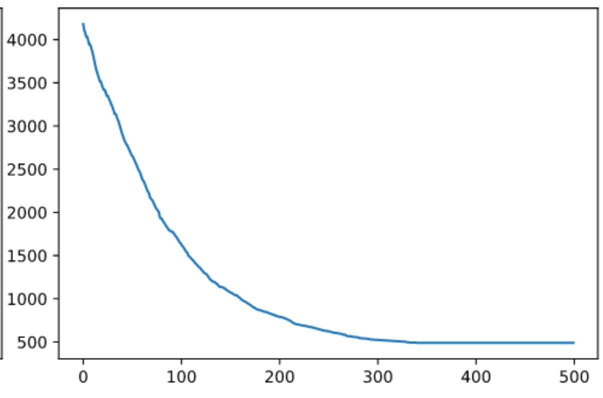
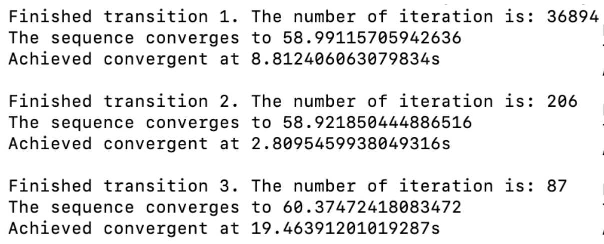


Figure 64: v=200

Figure 65: Isolated Results for Transition 3, Part 2, Simulated Annealing

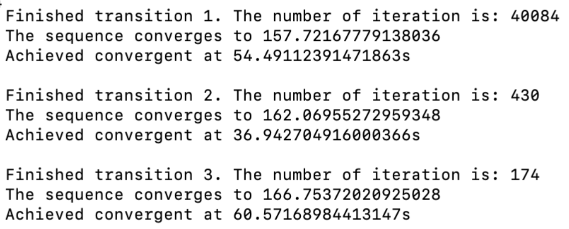


```
Finished transition 1. The number of iteration is: 36894
The sequence converges to 58.99115705942636
Achieved convergent at 8.812406063079834s

Finished transition 2. The number of iteration is: 206
The sequence converges to 58.921850444886516
Achieved convergent at 2.8095459938049316s

Finished transition 3. The number of iteration is: 87
The sequence converges to 60.37472418083472
Achieved convergent at 19.46391201019287s
```

Figure 66: v=50

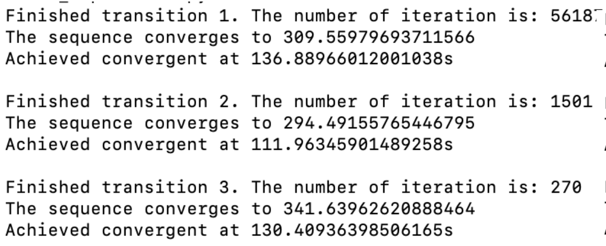


```
Finished transition 1. The number of iteration is: 40084
The sequence converges to 157.72167779138036
Achieved convergent at 54.49112391471863s

Finished transition 2. The number of iteration is: 430
The sequence converges to 162.06955272959348
Achieved convergent at 36.942704916000366s

Finished transition 3. The number of iteration is: 174
The sequence converges to 166.75372020925028
Achieved convergent at 60.57168984413147s
```

Figure 67: v=100

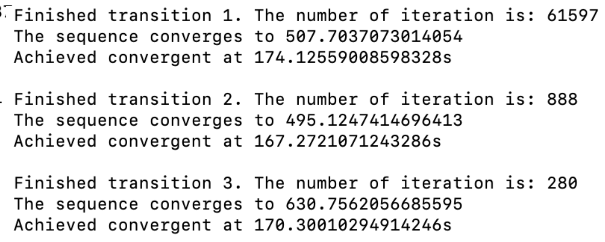


```
Finished transition 1. The number of iteration is: 5618
The sequence converges to 309.55979693711566
Achieved convergent at 136.88966012001038s

Finished transition 2. The number of iteration is: 1501
The sequence converges to 294.49155765446795
Achieved convergent at 111.96345901489258s

Finished transition 3. The number of iteration is: 270
The sequence converges to 341.63962620888464
Achieved convergent at 130.40936398506165s
```

Figure 68: v=150



```
Finished transition 1. The number of iteration is: 61597
The sequence converges to 507.7037073014054
Achieved convergent at 174.12559008598328s

Finished transition 2. The number of iteration is: 888
The sequence converges to 495.1247414696413
Achieved convergent at 167.2721071243286s

Finished transition 3. The number of iteration is: 280
The sequence converges to 630.7562056685595
Achieved convergent at 170.30010294914246s
```

Figure 69: v=200

Figure 70: Numerical Results for Part 3, Simulated Annealing

Part 3 Now that the former experiments have shown the superiority of Transitions 2 and 3 in terms of iterations, we need further validation on the running time since the time required for one iteration varies a lot. In this part, we return to the **reversing** neighborhood. But this time, the x-axis will be changed from Iterations to Running Time. Literally, running time is the most crucial indicator of a good algorithm. We still apply the algorithms on the same set of graphs with vertices $50, 100, 150, 200$. However, this time we no longer need to isolate the results of transitions 2 and 3. Figures 70 and 75 show the results.

Using the same set of hyper-parameters as Part 2, the Markov Chains have approximated the global minimums successfully. Referring to the plots, we can see that Transition 3 has weak competitiveness because it converges the slowest in terms of running time even if it takes the fewest number of iterations. But for Transition 2, the green curve (for Transition 2) in every plot decreases faster than the blue curve (for Transition 1), which indicates the efficiency in Transition 2. By checking the sequence of cost, the evident superiority of Transition 2 has been shown by both the smaller convergent point and the tremendous advantage in running time. In general, Transition 2 is better than Transition 1.

Part 4 Formerly, I have shown that no matter how I adjust the hyper-parameters, the performance of the Markov Chain is unsatisfactory. Thus, it is meaningless to use **swapping** as the definition of neighborhood. However, I still do experiments for this part in order to compare the superiority of the three transition mechanisms when the neighborhood definition is no longer "reversing". The objective of this part is to check whether the comparative advantage of different transition mechanisms varies among

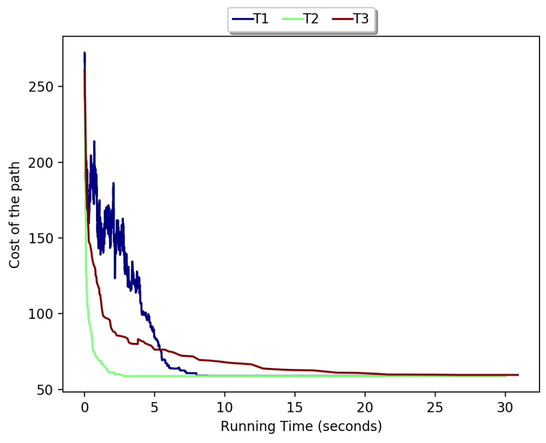


Figure 71: v=50

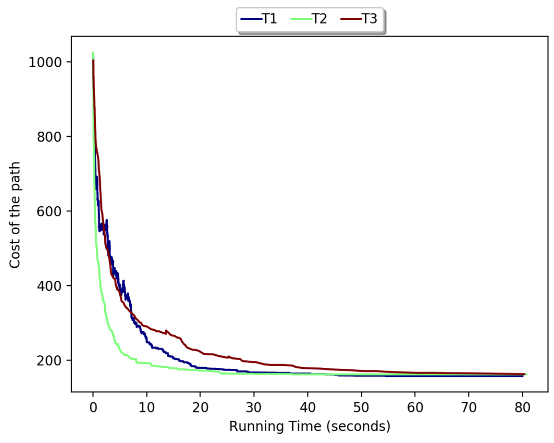


Figure 72: v=100

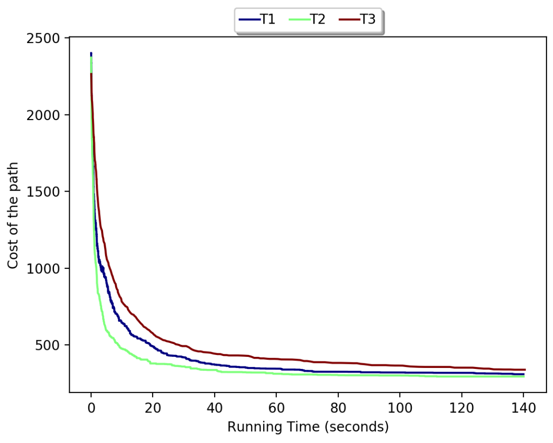


Figure 73: v=150

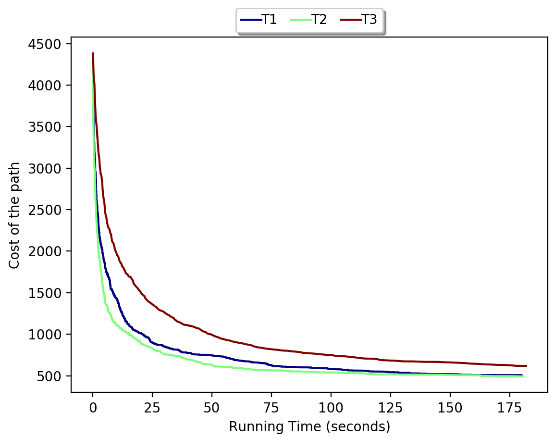
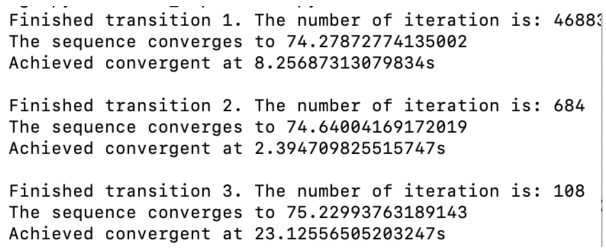


Figure 74: v=200

Figure 75: Combined Results for Part 3, Simulated Annealing

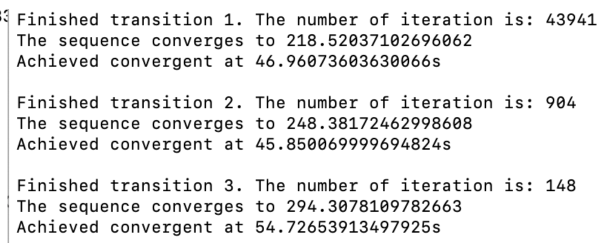


```
Finished transition 1. The number of iteration is: 4688
The sequence converges to 74.27872774135002
Achieved convergent at 8.25687313079834s

Finished transition 2. The number of iteration is: 684
The sequence converges to 74.64004169172019
Achieved convergent at 2.394709825515747s

Finished transition 3. The number of iteration is: 108
The sequence converges to 75.22993763189143
Achieved convergent at 23.12556505203247s
```

Figure 76: v=50

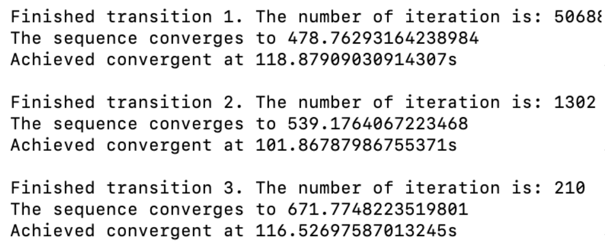


```
Finished transition 1. The number of iteration is: 43941
The sequence converges to 218.52037102696062
Achieved convergent at 46.96073603630066s

Finished transition 2. The number of iteration is: 904
The sequence converges to 248.38172462998608
Achieved convergent at 45.850069999694824s

Finished transition 3. The number of iteration is: 148
The sequence converges to 294.3078109782663
Achieved convergent at 54.72653913497925s
```

Figure 77: v=100

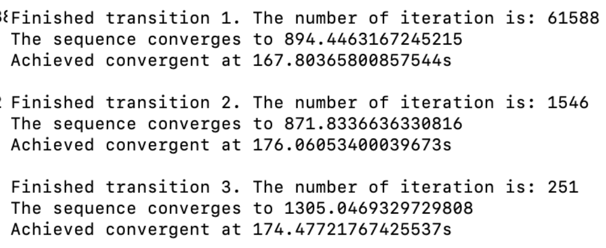


```
Finished transition 1. The number of iteration is: 5068
The sequence converges to 478.76293164238984
Achieved convergent at 118.87909030914307s

Finished transition 2. The number of iteration is: 1302
The sequence converges to 539.1764067223468
Achieved convergent at 101.86787986755371s

Finished transition 3. The number of iteration is: 210
The sequence converges to 671.7748223519801
Achieved convergent at 116.52697587013245s
```

Figure 78: v=150



```
Finished transition 1. The number of iteration is: 61588
The sequence converges to 894.4463167245215
Achieved convergent at 167.80365800857544s

Finished transition 2. The number of iteration is: 1546
The sequence converges to 871.8336636330816
Achieved convergent at 176.06053400039673s

Finished transition 3. The number of iteration is: 251
The sequence converges to 1305.0469329729808
Achieved convergent at 174.47721767425537s
```

Figure 79: v=200

Figure 80: Numerical Results for Part 4, Simulated Annealing

different neighborhood definitions. Figures 80 and 85 show the results.

Even though the green curve (Transition 2) decreases faster than the blue curve (Transition 1), the numerical results show that Transition 1 performs better than Transition 2 in terms of both convergent point and running time under the ”swapping” neighborhood. Therefore, we can conclude that the comparative advantage of different transition mechanisms varies among different neighborhood definitions.

2.4 Conclusions

According to the experiment results, we can summarize the following conclusions. I also provide my intuitive interpretation for each conclusion.

1. Number of iterations required before convergence: Transition 1 >> Transition 2 > Transition 3; Running time per iteration: Transition 3 > Transition 2 >> Transition 1.

   Interpretation: Probability measures have been added to Transitions 2 and 3, especially Transition 3 in which we allocate the weights to candidate states based on their costs. Thus, it takes much longer time for one single iteration in Transitions 2 and 3. However, since ”more preference” is given to the neighboring states with smaller costs, the cost of current state descents faster. Thus, the efficiency of the two alternative transitions is higher.

2. Running time required before convergence: Transition 3 > Transition 1 > Transition 2.

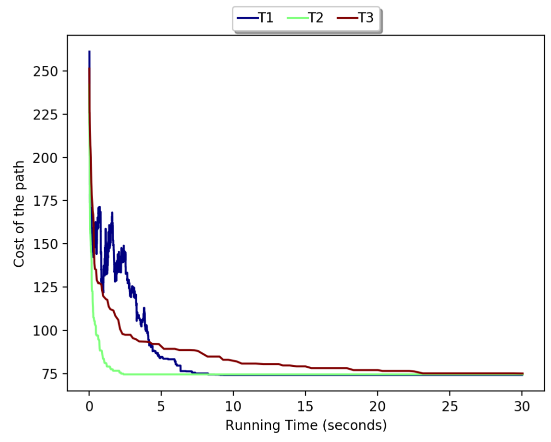


Figure 81: v=50

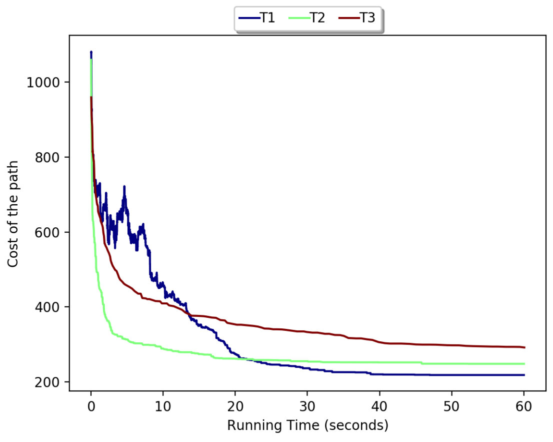


Figure 82: v=100

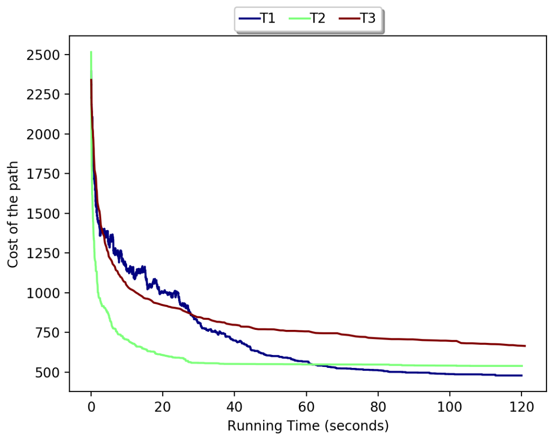


Figure 83: v=150

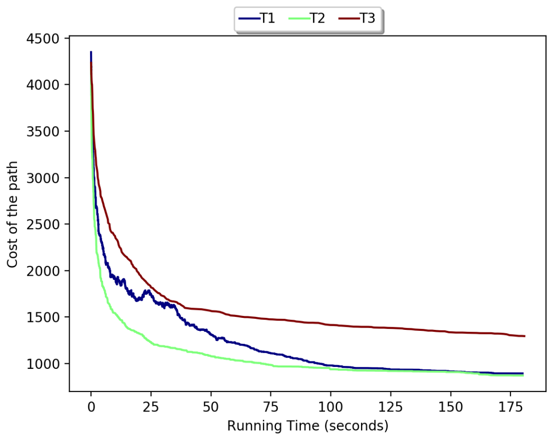


Figure 84: v=200

Figure 85: Combined Results for Part 4, Simulated Annealing

Interpretation: Since each single iteration of transition 3 takes much longer time than the others, its advantage in number of iterations cannot make up for its significant shortcoming in running time for a single iteration. But for transition 2, the reduction in number of iterations has successfully "hedged" its shortcoming in running time for a single iteration. Thus, transition 2 has the best overall performance.

3. In terms of neighborhood definition, "reverse" is better than "swap".

   Interpretation: In a permutation, we denote the connection between two cities as a "bond". When reversing a sub-sequence, the cost of the entire path will be changed by new arrangements of 2 bonds. But for swapping, the cost will be changed by new arrangement of 4 bonds. Thus, we make a guesstimating here. Changing more bonds in a permutation makes the Markov Chain harder to get to a new state with a smaller cost because of the hedging effect. (i.e. The length of some bonds may decrease while the others increase.)

4. When using the "reverse" neighborhood, Transition 3 performs the worst while Transition 2 performs the best.

   Interpretation: Transition 3 is trapped by its huge computing complexity caused by the candidate permutation generation and negative softmax weight allocation. As for Transition 2, since we put more weight on those neighboring states with smaller costs, the cost decreases faster than Transition 1. Meanwhile, the positive probability of switching into a state with larger cost keeps the chain from falling into the local mini-ma.

3 Improving Naive Bayes Classifier by Comonotonicity

3.1 Background

In probability theory, Bayes theorem computes the probability of an event based on some prior knowledge which is related to that event [20]. The Bayes formula: $Pr(\theta|X) = Pr(X|\theta)\frac{Pr(\theta)}{Pr(X)}$ has provided a strong basis for Bayesian inference. When we use X to represent the data and use θ to represent the parameters of a model, this provides some insights for classification. In this section, I will first discuss about the Naive Bayes classifier, a prevalent classifier based on Bayesian inference. Then, I will propose and test two alternative approaches based on Comonotonicity and Clustering in order to improve the performance of Naive Bayes. The results have shown the potential of Bayesian inference in image recognition.

3.2 Ideology of Bayesian Classifier

Before entering into the introduction of each method, we take a look at the Bayes theorem. In Bayes theorem, we wish to compute the conditional probability of an event A given that another event B has happened. The formula $Pr(A|B) = \frac{Pr(A,B)}{Pr(B)} = \frac{Pr(B|A)*Pr(A)}{Pr(B)}$ can transfer the calculation of the joint probability of events A and B to the posterior probability of event B given A and the prior probability of A.

When it comes to the classification tasks in machine learning, suppose the input X is n-dimensional, i.e. the features are $X_0 \quad X_{n-1}$. Given the input data $(x_0, x_1, ..., x_{n-1})$, we can make the classification by computing the posterior probability of each possible class Y_i and select the one with the largest posterior probability. Thus, according to the Bayes theorem, we have the following formula:

$$Pr(Y_i|X_0 = x_0, X_1 = x_1, ..., X_{n-1} = x_{n-1}) = \frac{Pr(X_0 = x_0, X_1 = x_1, ..., X_{n-1} = x_{n-1}|Y_i) * Pr(Y_i)}{Pr(X_0 = x_0, X_1 = x_1, ..., X_{n-1} = x_{n-1})}$$

Since for each class, the denominator remains the same, only $Pr(X_0 = x_0, X_1 = x_1, ..., X_{n-1} = x_{n-1}|Y_i)$ and $Pr(Y_i)$ affect the result. Indeed, the most crucial part is to compute the joint conditional probability based on the training set. All of the methods introduced later differs only in the way of computing this joint conditional probability.

3.3 Methodologies

In general, the features can be roughly divided into 3 types: *continuously distributed*, *discretely distributed but rank-able*, *discretely distributed but unrankable*. For the discrete but unrankable features such as gender, nationality, race, etc., we just treat them as mutually independent with the other features. My amendments mainly deal with the other two types of features. Basically, there are two branches which are **aggregation** and **clustering**. For aggregation, we just compute the conditional joint probability of all continuous features and discrete but rank-able features. For clustering, the computation is restricted within clusters of features rather than all features. Firstly, I compute the correlation coefficient matrix and take the absolute value element-wisely. Next, I use 1 minus the absolute value of each entry in the matrix to get a distance matrix D in which the higher correlation between two features X_i and X_j (no matter whether positive correlation or negative correlation), the smaller D_{ij} would be. Then I use agglomerative nesting clustering (AGNES) method to do hierarchical clustering. The algorithm starts by treating each feature as a singleton cluster. During each iteration, the algorithm finds one pair of clusters having the shortest distance, and merge them into a new one. For any two clusters, the distance between them equals to the shortest distance between the features in these two clusters [11]. I set a minimum correlation min_corr so that the clustering algorithm will stop if the distance between any two clusters is larger than $1 - min\_corr$.

3.3.1 Naive Bayes Classifier

In Naive Bayes, the "naive" assumption is the mutual independence of all the features. In this way, the joint conditional probability equals to the product of conditional probabilities of each X_i. Thus, regardless of the dependence structure of input features, we use $\prod_{i=0}^{n-1} Pr(X_i = x_i|Y_i)$ to replace $Pr(X_0 = x_0, X_1 = x_1, ..., X_{n-1} = x_{n-1}|Y_i)$ [4]. If a feature X_i is continuous-valued, $Pr(X_i = x_i|Y_i)$ is usually computed based on Gaussian distribution with a mean μ and variance σ : $Pr(X_i = x_i|Y_i) = N(X_i = x_i|\mu_{Y_i}, \sigma_{Y_i}) = \frac{1}{\sqrt{2\pi}\sigma_{Y_i}} e^{\frac{(x_i - \mu_{Y_i})^2}{2\sigma^2}}$.

Since Naive Bayesian classification required each conditional probability to be non-zero, otherwise, the product of posterior probabilities may become zero, we should also employ Laplacian correction to avoid the zero probability problem. It can be achieved by adding 1 to the number of occurrences to each case.

In general, although the Naive Bayes classifier ignores the dependence structure of input features, it indeed performs well on many data-sets. Intuitively speaking, this might be due to the nature of classification tasks. We just need to get accurate ranking of posterior probabilities on each class rather than computing the exact posterior probability. Thus, if the features share relatively weak dependencies and the number of features is small, Naive Bayes can be robust. However, things becomes different when it comes to strong dependencies and high dimensional data.

3.3.2 Comonotonic Classifier

I propose an approach to estimate the conditional joint density based on comonotonicity in Actuarial Science. There are two sub-methods. The first one is to treat all continuous or discrete but rank-able features as comonotonic. The other one is to conduct this method only within clusters.

In probability theory, comonotonicity refers to perfect positive dependence among the components of a random vector. Mathematically, a set $S \subset \mathbf{R^n}$ is comonotonic if $\forall \mathbf{X}$ and $\mathbf{Y} \in S$, either $\mathbf{X} \geq \mathbf{Y}$ or $\mathbf{X} \leq \mathbf{Y}$ [6]. In other words, a comonotonic set S is a totally ordered set.

Comonotonicity is widely used in risk measures. Before delving into the probability measure, we recall the method to sample from a certain probability distribution. Suppose a random variable X has cumulative distribution function F_X, since $F_X \quad U(0,1)$, the way to sample from X's distribution is to uniformly generate a random number u from 0 to 1 and the sample value $x = F_X^{-1}(u)$. Then in the case of comonotonicity, suppose $X_1, \ldots, X_n$ are comonotonic, we sample by uniformly generating a random number u from 0 to 1 so that $x_i = F_{X_i}^{-1}(u) \quad \forall i \in \{1, \ldots, n\}$. In this sense, not all combinations of values for $X_1, \ldots, X_n$ are feasible since we generate all of them using the same u.

Theoretically, for data-sets with a large feature space n, assume for each feature there are m possible values on average. Then, a fully general machine learning algorithm which attempts to learn all possible dependencies among the features has to deal with m^n possible combinations of all features, and thus needs $O(m^n)$ many data. But for Naive Bayes, due to the "naive" assumption on mutual independence, it only needs $O(mn)$ many data. Meanwhile, for comonotonicity, within a group of k comonotonic features, the feature value combination is $O(km)$ and for all features the feature value combination is $O((km)^{\frac{n}{k}})$. For aggregated comonotonic classifier, $k = 1$. Therefore, in terms of the size of data-sets, Naive Bayes ¡ Comonotonic classifier < general machine learning algorithms.

Probability Measure There are indeed various approaches to estimate the joint density given the marginal ones. For example, in quantitative finance, Copulas are widely used to model and minimize tail risk by defining a multivariate cumulative distribution function for which the marginal distribution follows $U(0,1)$ [15]. According to what we have discussed before, the sampling method in comonotonicity requires the features to be discretely distributed. Otherwise, it is trivial to use random numbers to determine the probability mass of each specific feature value combination. Therefore, we need to discretize the continuous features. For discrete but rank-able variables, $Pr(X_i = x_i)$ can be represented by an interval $(max\{0, \sum_{k=0}^{k=x_i-1} Pr(X_i = k)\}, \sum_{k=0}^{k=x_i} Pr(X_i = k)]$. However, for continuous features, the method for discretization is of great importance for the performance of the model, I will provide more details in the experiment part of this report.

In terms of marginal distribution, when we draw a sample from the distribution of X_i, we just need to uniformly generate a random number u between 0 and 1 and get $X_i = x_i$ if and only if u is in this interval. In this way, we can transform the calculation of probability mass into finding the intersection of n intervals. Intuitively, when evaluating the joint probability of dependent variables, multiplication of marginal probabilities will make the joint probability smaller. However, when it comes to the intersection of intervals, as long as the comonotonicity condition holds, such error can be reduced.

Aggregated Comonotonic Classifier Without the loss of generality, we assume $X_0, \ldots, X_{m-1}$ are continuous or discrete but rank-able features. In this approach, we treat all of them as comonotonic features. This is actually a very naive assumption and it is very likely that the intersection of n intervals is empty, which triggers the Laplacian correction and exacerbates the performance of the model.

Clustered Comonotonic Classifier In this approach, we use the AGNES method to do the clustering. We should use the original data rather than discretized data for the computation of the correlation coefficient matrix. The discretization is always done after clustering. For the rank-able variables, we compute the probability mass of each cluster. Since the features within a cluster are not necessarily positively correlated. We need to take the negative of some feature values in order to make the correlation be positive. Within the cluster, say, we have variables $X_0, \ldots, X_5$, we assume that they are comonotonic. We treat X_0 as the basic variable and for all other variables X_i's, if $corr(X_0, X_i) < 0$, we take the negative of X_i in order to keep a positive correlation with the basic variable. Otherwise, keep X_i unchanged. Suppose in an observation in the test set, $X_i = x_i$ for i between 0 and 5, we compute 6 intervals by the formula: $(max\{0, \sum_{k=0}^{k=x_i-1} Pr(X_i = k)\}, \sum_{k=0}^{k=x_i} Pr(X_i = k)]$, and get the intersection. The length of intersection is the probability mass of this cluster conditional to a specific class.

Applying the method above to every observation in the test set, we can get the probability distribution of each test data over all possible classes. In this case, we choose the largest one to be the predicted class.

3.4 Experiments

The selection of data-set is nontrivial for the verification of a new algorithm. Since the objective of this research is to improve Naive Bayes classifier, at least on some specific data-sets, the performance of Naive Bayes should not be good on these data-sets. Otherwise, no significant improvement can be made anymore. Referring to the methodology of Naive Bayes, it ignores the dependence structure of features. Based on intuition, we may consider data-sets with strong dependency among features. For each data-set in this section, I apply Naive Bayes Classifier, Aggregated Comonotonic Classifier, Clustered Comonotonic Classifier separately. For the binary classification task, I use ROC Curve and AUC value to evaluate the performance.

Before describing the experiments, I provide the details for discretization first.

3.4.1 Discretization

Transforming a continuous feature into a discrete feature is of vital importance for comonotonic classifier. Traditionally, we can discretize the value into equal-width bins based on the range or discretize into equal-sized buckets based on rank or sample quantiles. Both of them requires the specification of number of categories, which is nontrivial as well. I have tried both of these discretization methods and also tried to optimize the combination of number of classes for all continuous features. Unfortunately, these trials are unsuccessful and I will describe them later in the subsection of Discussion.

In practice, I discretize the continuous features by mean and standard deviation. For each continuous feature and each class in the training set, the bins should be stored for the sake of data discretization for continuous features in the test set. Here I discretize each continuous variable given the class number into 8 categories. The bins are: $(-\infty, mean–3 * std]$, $(mean–3 * std, mean–2std]$, $(mean–2 * std, mean–std]$, $(mean–std, mean]$, $(mean, mean + std]$, $(mean + std, mean + 2 * std]$, $(mean + 2 * std, mean + 3 * std]$, $(mean + 3 * std, +\infty)$. Then for each rank-able variable X_i, we store the value of $Pr(X_j = k|Y = Y_j)$ for any possible value k. Note that we should use Laplacian correction to avoid zero probability.

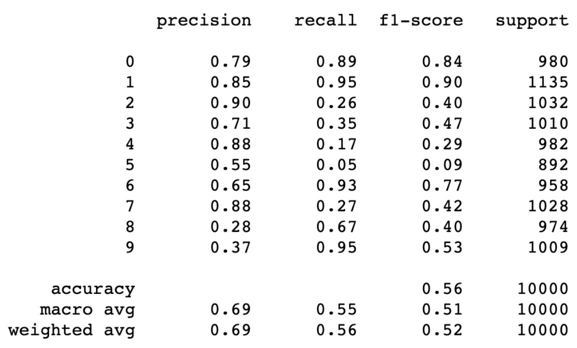
3.4.2 Overall Performance on MNIST Data-set

The MNIST data-set consists of handwritten digits from 0 to 9 in which $60,000$ are for training and $10,000$ are for testing. All of the digits in MNIST have been normalized in size and are centered in a $28 * 28$ image [3]. Inspired by the application of logistic regression on MNIST data-set, I discovered the potential of comonotonic classifier on image processing. The MNIST has 10 classes. When we train a logistic regression model on MNIST data-set, basically, we should reshape each image of size 28 * 28 in which each entry is real-valued to a vector of size 1 * 784. Then we randomly initialize a matrix of size 784 * 10 and use stochastic gradient descent to optimize the cross entropy error function. Using the idea from this, I also reshape each image in MNIST data-set and conduct discretization to each of the 784 features. The results are shown in Figure 89.

Analysis of Results From the results, we can see that the clustered comonotonic classifier demonstrates significant improvement in precision, recall, F1-score and accuracy rate compared with the Naive Bayes classifier. However, The aggregated comonotonic classifier performed extremely poor when dealing with high dimensional data-set. Literally, we can attribute the poor performance of aggregated comonotonic classifier to the naive assumption of comonotonicity among all rank-able features, which leads to empty intersection of intervals and triggered the Laplacian correction and exacerbated the result.

As for Naive Bayes, we can see that the precision and recall is not stable on some classes. For example, the precision on classes 5, 8, 9 and the recall on classes 2, 3, 4, 5, 7 are even lower than 0.6. But for clustered comonotonic classifier, the precision and recall on each class is at least 0.7 and the testing accuracy significantly surpasses that of Naive Bayes.

Our data processing method for the MNIST data-set is to reshape the images, or in other words, $2d$ arrays to $1d$ arrays. Although such reshaping operation commits spacial information loss from the original image, the accuracy is over 80%. In the next section, I will apply the three algorithms on a low dimensional data-set and compare the performances again.

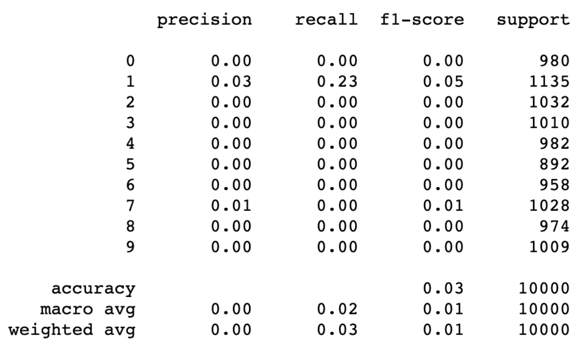


```
              precision    recall  f1-score   support

           0       0.79      0.89      0.84       980
           1       0.85      0.95      0.90      1135
           2       0.90      0.26      0.40      1032
           3       0.71      0.35      0.47      1010
           4       0.88      0.17      0.29       982
           5       0.55      0.05      0.09       892
           6       0.65      0.93      0.77       958
           7       0.88      0.27      0.42      1028
           8       0.28      0.67      0.40       974
           9       0.37      0.95      0.53      1009

    accuracy                           0.56     10000
   macro avg       0.69      0.55      0.51     10000
weighted avg       0.69      0.56      0.52     10000
```

Figure 86: Naive Bayes

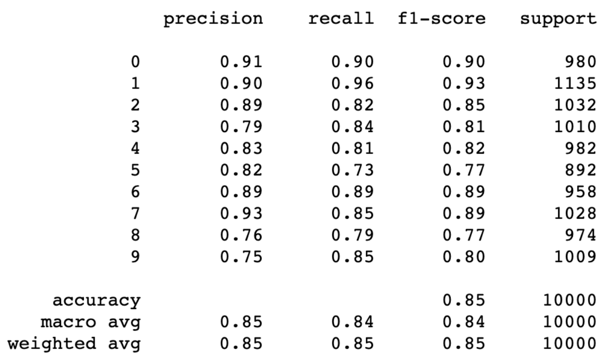


```
              precision    recall  f1-score   support

           0       0.00      0.00      0.00       980
           1       0.03      0.23      0.05      1135
           2       0.00      0.00      0.00      1032
           3       0.00      0.00      0.00      1010
           4       0.00      0.00      0.00       982
           5       0.00      0.00      0.00       892
           6       0.00      0.00      0.00       958
           7       0.01      0.00      0.01      1028
           8       0.00      0.00      0.00       974
           9       0.00      0.00      0.00      1009

    accuracy                           0.03     10000
   macro avg       0.00      0.02      0.01     10000
weighted avg       0.00      0.03      0.01     10000
```

Figure 87: Aggregated Comonotonic



```
              precision    recall  f1-score   support

           0       0.91      0.90      0.90       980
           1       0.90      0.96      0.93      1135
           2       0.89      0.82      0.85      1032
           3       0.79      0.84      0.81      1010
           4       0.83      0.81      0.82       982
           5       0.82      0.73      0.77       892
           6       0.89      0.89      0.89       958
           7       0.93      0.85      0.89      1028
           8       0.76      0.79      0.77       974
           9       0.75      0.85      0.80      1009

    accuracy                           0.85     10000
   macro avg       0.85      0.84      0.84     10000
weighted avg       0.85      0.85      0.85     10000
```

Figure 88: Clustered Comonotonic

Figure 89: Overall Performance MNIST, Comonotonicty

3.4.3 Overall Performance on Adult Census Income Data-set

Now that we have tested the three classifiers on an "artificially made" high-dimensional data-set. Next, we turn to a data-set in which the features are highly correlated and apply the three algorithms on the Adult Census Income data-set. The data-set was extracted from the Machine Learning Repository of University of California, Irvine. [19]. It contains around 32,000 rows with both numerical and categorical features. They are **age**, **workclass**, **fnlwgt**, **education**, **education.num**, **marital.status**, **occupation**, **relationship**, **race**, **sex**, **capital.gain**, **capital.loss**, **hours.per.week**, **native.country**. The objective is to predict whether an adult has income over \$$50k$ or not based on the 14 features. Intuitively, we may find that there would be strong correlation between **education** and **education.num**. Indeed, in the original data-set, **education** is a feature of text value specifying the actual highest level of institution that this person has ever taken while **education.num** is a natural number ranking the level of institution in the **education** column. In order to increase the level of dependence among the features while not just simply replacing one feature with another, I did some data transformation. Referring to the distribution of **education.num** (as shown in Figure 90), most of the people are between level 9 and level 14. Thus, I encode the feature **education** according to the value in **education.num**. The correspondence is: $\leq 9 \rightarrow 0$, $10 \rightarrow 1$, $11\&12 \rightarrow 2$, $\geq 13 \rightarrow 3$. In this way, the two columns are highly correlated but they are not strictly equal.

For the description of the feature **fnlwgt**, they are described as follows: The weight of the current Census (CPS) records is controlled, and the number of non-institutional civilians in the United States can be freely measured. These are arranged for us by the Civilian Department of the Census Bureau every month. We use 3 sets of controls.

1. The estimated single cell population for each state population is 16+.
2. Control Hispanic origins by age and gender.
3. Conduct race, age and gender control. [18]

We use all three sets of controls in the weighted program and perform "traversal" 6 times through them to finally return to all controls used. The term "estimate" refers to the total population derived from the CPS by creating a "weighted count" with specific population socioeconomic characteristics. We assign similar weights to those people whose demographic characteristics are similar. Regarding this statement, an important warning needs to be remembered. That is because the CPS sample is actually a collection of 51 state samples, each sample has its own selection probability, so this statement is only applicable within the state.

Now after some pre-processing, the continuous features are **age**, **fnlwgt**, **education.num**, **capital.gain**, **capital.loss**, **hours.per.week**. The unrankable features are **workclass**, **marital.status**, **occupation**, **relationship**, **race**, **sex**, **native.country**. The discrete but rank-able feature is **education**. I apply the three models to this pre-processed data-set.

The results are as shown in Figure 94:

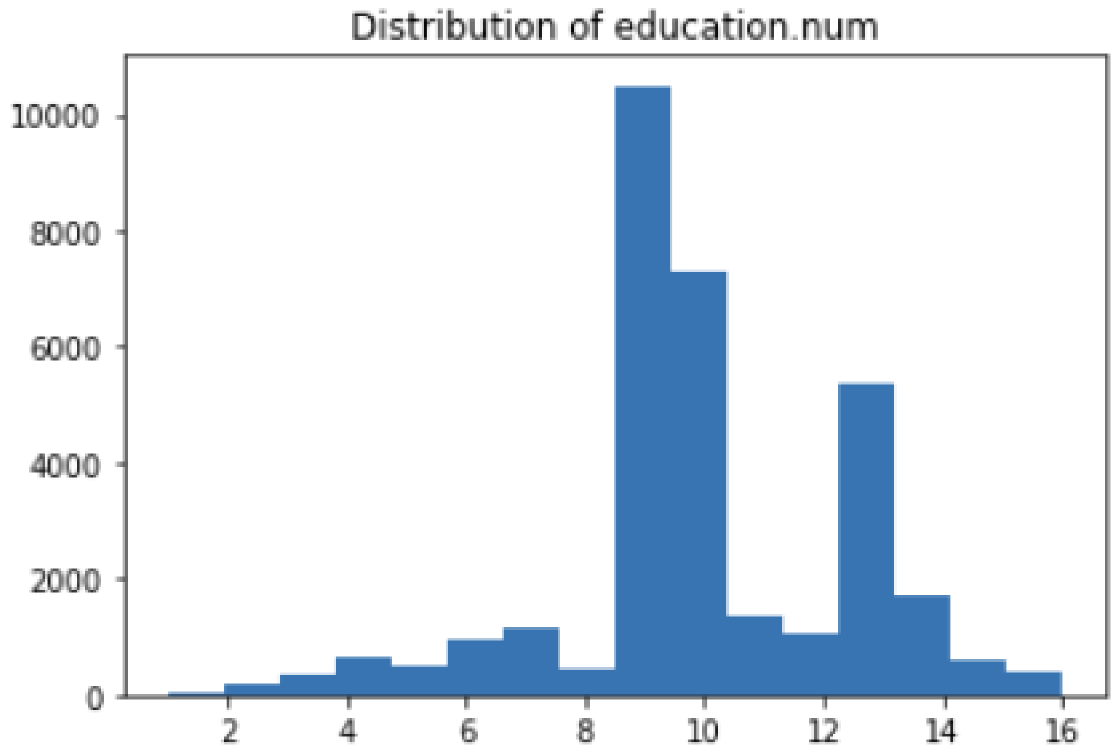


Figure 90: Distribution of education.num

```
              precision    recall  f1-score   support

           0       0.80      0.95      0.87      4623
           1       0.66      0.29      0.41      1521

    accuracy                           0.79      6144
   macro avg       0.73      0.62      0.64      6144
weighted avg       0.77      0.79      0.76      6144
```

Figure 91: Naive Bayes

```
              precision    recall  f1-score   support

           0       0.89      0.77      0.83      4623
           1       0.51      0.72      0.60      1521

    accuracy                           0.76      6144
   macro avg       0.70      0.75      0.71      6144
weighted avg       0.80      0.76      0.77      6144
```

Figure 92: Aggregated Comonotonic

```
              precision    recall  f1-score   support

           0       0.91      0.85      0.88      4623
           1       0.63      0.74      0.68      1521

    accuracy                           0.83      6144
   macro avg       0.77      0.80      0.78      6144
weighted avg       0.84      0.83      0.83      6144
```

Figure 93: Clustered Comonotonic

Figure 94: Overall Performance MNIST, Comonotonicty

AUC-ROC Curve When it comes to the evaluation of a binary classification model, we can count on **AUC-ROC Curve** where AUC stands for Area Under the Curve and ROC stands for Receiver Operating Characteristics [10].

An ROC Curve is a probability curve which shows the performance of a classification model at different classification thresholds. A classification threshold is also called a decision threshold, which outputs ”positive” if a value (between 0 and 1) is above the threshold. The $x-axis$ of the curve is the False Positive Rate (FPR) while the $y-axis$ is the True Positive Rate (TPR). TPR is a synonym of recall, sensitivity or probability of detection. The formula is $TPR = \frac{TP}{TP+FN}$. FPR is also called probability of false alarm [21]. The formula is $FPR = \frac{FP}{FP+TN}$. Before plotting the ROC curve, we train the model and compute the probability distribution over negative (class 0) and positive (class 1) for each item in the test set. Then we pass in the labels for the test set, i.e., y_{test} which is a vector of 0 and 1, and the probability of ”positive” for each item. When reducing the classification threshold, more items will be classified as positive. In this way, both FP and TP increase. Therefore, the ROC curve starts from $(0,0)$ and ends at $(1,1)$. We also set the baseline which is the line segment connecting $(0,0)$ and $(1,1)$. This can be achieved by a random classifier. The higher the ROC curve is above the baseline, the better the model's performance.

When it comes to AUC, as the name suggests, it uses numerical methods to measure the area underneath the ROC curve. It provides a measure of aggregate performance over all classification thresholds. We can interpret it as the probability that the model ranks a random positive item higher than a random negative item. Thus, an excellent model has AUC near to 1. On the contrary, if a model has AUC close to 0, then it just runs in the opposite direction since it predicts 0s and 1s reciprocally. The worst case is when the AUC is 0.5 and the ROC curve approximately coincides with the baseline, indicating that the model has no class separation capacity at all.

Since the Adult Census Income is a binary classification task, I use AUC-ROC Curve to check the performance of the models. Using the same set of hyper-parameters, I get the following Figure 95:

The AUC values for the three models are: 0.851 for Naive Bayes, 0.897 for Clustered Comonotonic, 0.808 for Aggregated Comonotonic.

3.5 Discussion for Discretization

In the experiments above, I use the mean and standard deviation to discretize the data into 8 categories and it works fine for the features which approximately follow normal distribution. However, we may come up with some questions. What if we discretize each continuous feature into different number of bins? To what extent would the discretization method influence the model's performance? Of course, applying grid search to test the accuracy of all possible combinations of number of bins will definitely find the optimal one. However, it is of $O(k^m)$ where k is the maximum number of bins and m is the number of continuous features. Thus, grid search is not applicable for high dimensional data-sets. Another good method is random search and it has been verified by empirical

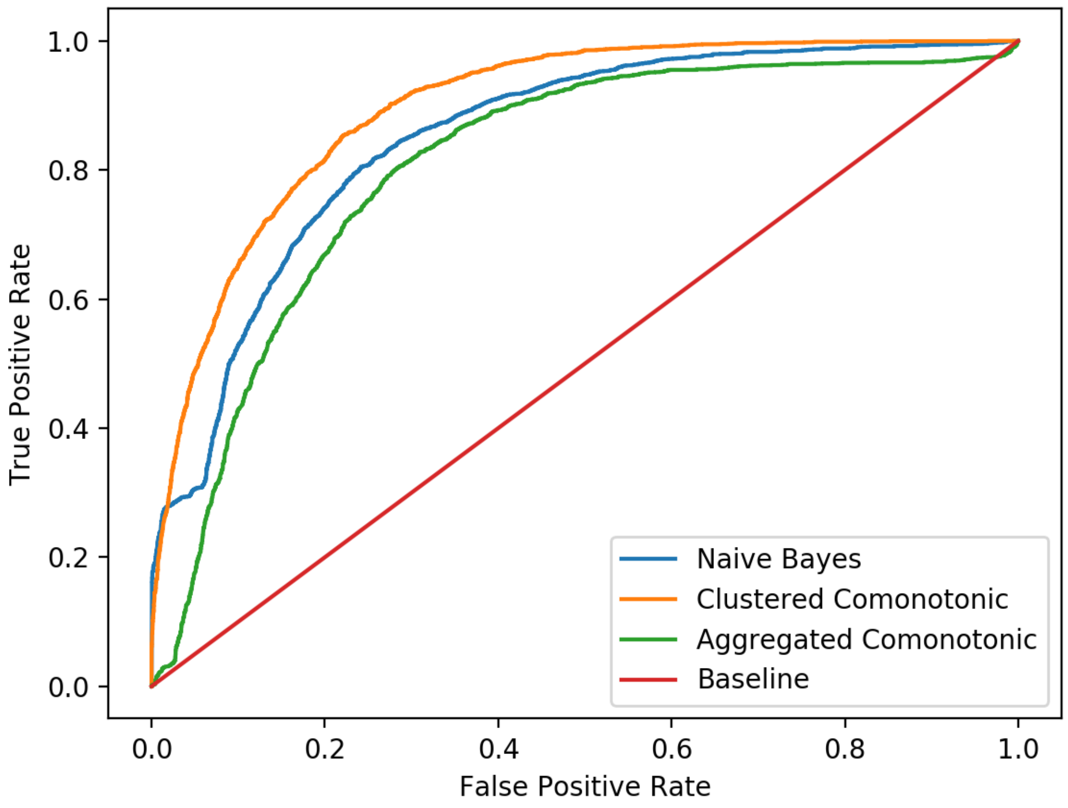


Figure 95: ROC-AUC Curve

and theoretical studies to be more efficient for hyper-parameter tuning than trials on grid search [14]. In this part, I make some "directional" modifications to random search which incorporate the ideology of Simulated Annealing algorithm to find an optimal combination of the number of buckets for all continuous features. For simplicity, I conduct two sets of experiments. In the first set, the bins are of equal *length*, just as what we do when creating a histogram. In the second set, the bins are of equal *size*, or in other words, quantile-based discretization. In each set, assume each continuous feature X_i can be discretized into i_k categories where i_k is a natural number between 2 and 10. I initialise them randomly. For the sake of reducing the demand for computing power, I use the traditional method of transition in each iteration. In other words, in the ith iteration, suppose the accuracy in the previous iteration is acc_{i-1}, we change the number of bins for one continuous feature, and then apply clustered comonotonic model to the preprocessed data and compute a new accuracy acc_i on the test set. If $acc_i > acc_{i-1}$, then just switch to this new combination of bins. Otherwise, the transition probability is $\exp(-\frac{acc_{i-1}-acc_i}{T})$ where T is a temperature and it should be gradually lowered during the process. Here are the results of the application of Simulated Annealing algorithm on the discretization optimisation task.

3.5.1 Bins of Equal Length

I use the function *pandas.cut*() in Python to discretize the data into bins of equal size. The hyper-parameters are: *maximum iteration* : 250; *initial temperature* : 100; *annealing schedule* : 10. I reduce the temperature to 1/10 after every 10 iterations. The overall accuracy and class-wise accuracies are shown in Figure 99.

3.5.2 Bins of Equal Size

I use the function *pandas.qcut*() in Python to discretize the data based on quantiles. The hyper-parameters are: *maximum iteration* : 200; *initial temperature* : 100; *annealing schedule* : 5. I reduce the temperature to 1/10 of the original once every 5 iterations.

The plot of overall accuracy and class-wise accuracies are shown in Figure 103.

3.5.3 Analysis of Results

In terms of overall accuracy, the plots show that no matter which type of discretization method we choose, the overall accuracy oscillated in the first tens of iterations, then it gradually goes up. However, changing the combination of numbers of bins seems insignificant for the optimisation of overall accuracy since the range is even lower than 10%. But at least, we have verified the validity of simulated annealing algorithm in discrete optimisation once more. When it comes to the class-wise accuracy, the performance of boosting on class 0 is better than that on class 1.

The results shown above have provided a further indication on the lower importance of finding the most optimal combination of number of bins for continuous features compared with the importance of modifying the methodology itself. Therefore, even

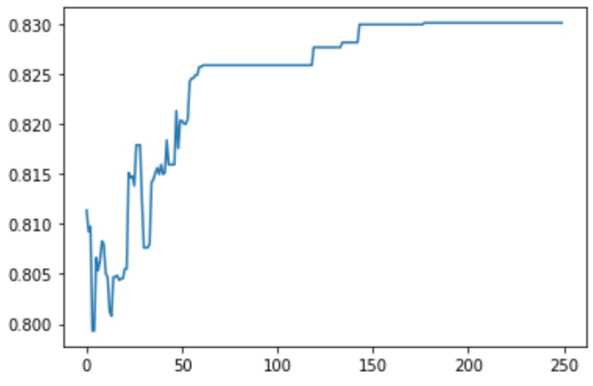


Figure 96: Overall Accuracy, Equal Bin

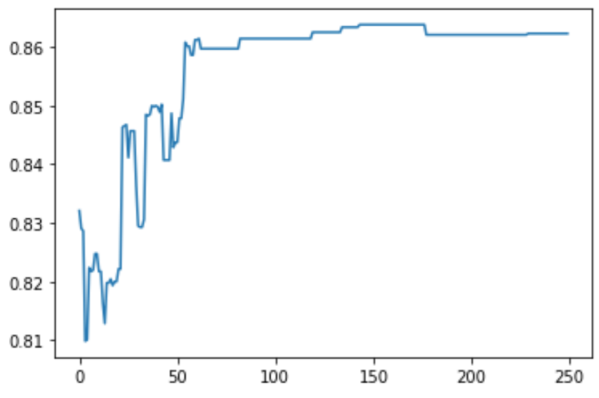


Figure 97: Class 0 Accuracy, Equal Bin

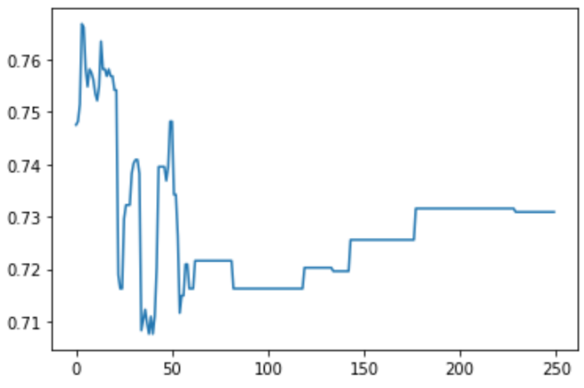


Figure 98: Class 1 Accuracy, Equal Bin

Figure 99: Equal Bin Discretization, Comonotonicty

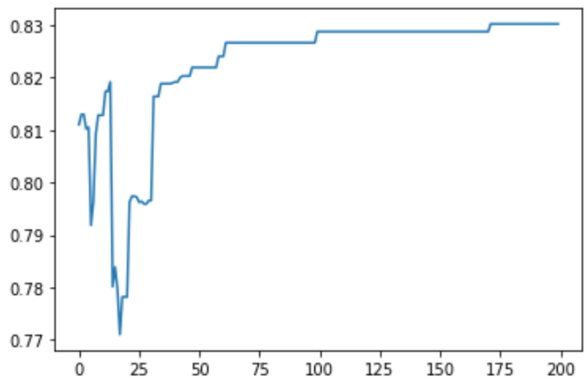


Figure 100: Overall Accuracy, Equal Size

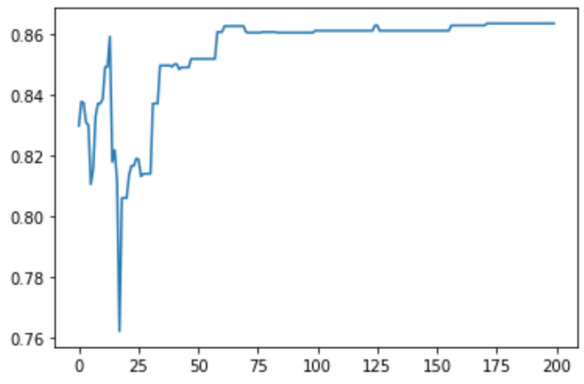


Figure 101: Class 0 Accuracy, Equal Size

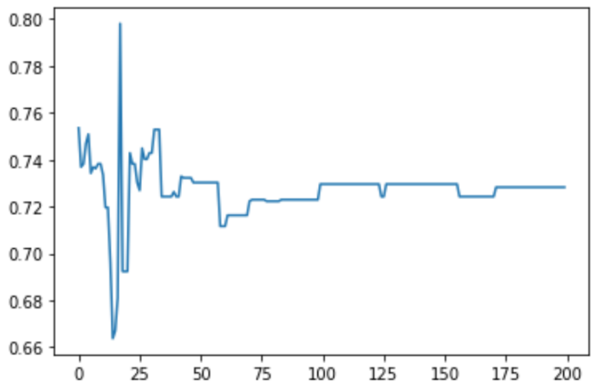


Figure 102: Class 1 Accuracy, Equal Size

Figure 103: Equal Size Discretization, Comonotonicty

though the application of simulated annealing is an unsuccessful approach in performance boosting, it confirms that the naive approach to discretize based on mean and standard deviation is harmless to the final result.

3.6 Conclusion

Even though the naive assumption in Naive Bayes ignores the dependency among features, it simplifies the learning process greatly without too much loss in accuracy in many circumstances. Empirical studies of Naive Bayes classifier attribute the success to the unnecessary relationship between optimality in zero-one loss and the quality of model fitting to the actual probability distribution [5]. In this research, I mainly discussed about the ideology of Naive Bayes classifier and proposed using the comonotonicity method to replace the "naive" assumption of features' mutually independence. In this way, we need to discretize the continuous features so that the marginal density for each feature can be represented as an interval. (i.e. $Pr(X_i = x_i)$ can be represented by $(max\{0, \sum_{k=0}^{k=x_i-1} Pr(X_i = k)\}, \sum_{k=0}^{k=x_i} Pr(X_i = k)]$.) The necessity of discretization does not imply a significant performance boosting when trying to optimise the combination of the number of bins for each continuous feature. In practical applications, I just discretize them into 8 bins using the mean and standard deviation. Then we can transform the computation of joint density into finding the intersection of multiple intervals. Under this ideology, there are two affiliated approaches, namely Aggregated Comonotonic and Clustered Comonotonic. The former one conceives another naive assumption of mutually comonotonic among all features except those discrete and unrankable ones such as gender, race, religion etc. The latter one becomes more reasonable since it just treats those highly correlated features as comonotonic. I use agglomerative nested clustering algorithm to cluster the rank-able features where the distance between any two features is $1 - |corr(X_i, X_j)|$. After clustering, the features within a cluster are treated as comonotonic while different clusters are treated as mutually independent.

The three methods are first tested on the MNIST data-set. The results of Aggregated Comonotonic revealed huge flaws when dealing with high dimensional data-sets. This might be due to the naive assumption of mutually comonotonicity among all rank-able features, which caused a large possibility to encounter an empty intersection of intervals. When empty intersection really happens, the probability is computed using Laplacian correction, which is invariant among any set of feature values with empty intersections. However, the performance of Clustered Comonotonic showed significant improvement to the Naive Bayes Classifier. This is reasonable since we have reshaped each image of size $28 * 28$ into a vector with size $1 * 784$, so there should be a probability distribution of each pixel to contain information of the handwritten digits. We can even regard the clustering process as a searching process to find those pixels with larger probabilities to contain information. The success of Clustered Comonotonic on MNIST indicates the potential of this method in image recognition.

I also tested the three methods on the Adult Census Income data-set. Both the overall accuracies and AUC-ROC curves showed the ranking of superiority to be: Clustered Comonotonic $>$ Naive Bayes $>$ Aggregated Comonotonic. This test is a verification of the advantage of Clustered Comonotonic on low dimensional data-sets in which some features are highly dependent.

Bayesian statistics is worthwhile for classification model designing. Although Naive Bayes has been widely used in many applications such as spam filtering, multi-class prediction etc., it is meaningful to think of alternative methods to replace the naive assumption of mutually independence. Empirical evidences have shown that the Clustered Comonotonic method is a good way to solve this issue.

4 Conclusion

In this project, we have delved deeply into three topics: the application of Gibbs Sampler on the Vertex Coloring Problem; the application of Simulated Annealing algorithm on the Travelling Salesman Problem; the application of Comonotonicity on the improvement of Naive Bayes classifier. I have made some discoveries in each of the topic. Some are discoveries of features or intrinsic mechanisms of a process while the other are novel improvements to the existing methodologies.

As for the Vertex Coloring Problem, previous scholars mainly focused on the way to minimize the number of colors used in the graph. However, my attention lie on the features about vertices and edges when running the Gibbs sampler. When it comes to the vertices, my focus is on the average number of vertices which occupy a same certain color. In order to reduce the random error, I compute the Manhattan distance between the vector containing the actual average number of vertices occupying each color and the vector of expectation, which is indeed a vector of length q and each entry is $\frac{n}{q}$. I find that when increasing q or reducing the connectivity of the graph, the total number of configurations increases, and the Manhattan distance between the two vectors converges faster. When it comes to the edges, my work is more like a verification of the correctness of MCMC. The results have shown that for any edge type, the average occurrences converges to $\frac{e}{qC2}$ when running the Gibbs sampler. Moreover, the speed of convergence is almost the same for each type.

As for the application of Simulated Annealing algorithm on the Travelling Salesman Problem, I proposed two alternative transition mechanisms in order to improve the performance. In the traditional approach, we randomly generate a neighbor of the current permutation and then transit directly if the new cost is lower. Otherwise, the transition probability is equal to $\exp(-\frac{f(s_j)-f(s_i)}{T})$. In the first alternative approach, I assign a larger weight on those neighboring permutations with lower costs. If I get a new permutation of lower cost, then transit directly. Otherwise, the transition probability is exactly the same as the traditional approach. In the second alternative approach, I base on the idea in the first one and assign more weights on those permutations which reduce the cost in a larger scale. The weight assignment is achieved by the softmax function. The estimation of performance is based on both the number of iterations required and the running time required before convergence. After hyper-parameter tuning, I find that the second alternative approach requires the fewest number of iterations while the traditional one requires much more than the other two. However, the price of the alternative approaches, especially the second one, is huge since it takes quite a long time to complete a single iteration. Thus, comparing the running time is also necessary. The plots of running time suggest that the first alternative approach converges fastest while the second alternative approach the slowest. Therefore, these evidences is a solid verification of the better performance of the first alternative approach. At least, it is a practical way to give more preference to permutations with smaller costs.

As for the application of Comonotonicity to improve the performance of Naive Bayes classifier, I incorporated both the ideology of comonotonicity and clustering. The rank-able features are clustered based on the correlation coefficient matrix. The distance between any two features is defined as $1 - |corr(X_i, X_j)|$. I treat the features inside the same cluster as comonotonic and their joint posterior density is the size of the intersection of intervals for each feature's marginal posterior density. This classifier is called clustered comonotonic classifier. I apply this classifier on both the MNIST data-set and the Adult Census Income data-set. The performance on MNIST shows significant improvement to the Gaussian Naive Bayes method while on the Adult Census Income data-set there is also some improvement but not that large. Thus, we conclude that clustered comonotonic is a useful way to replace the naive assumption of mutual independence in Gaussian Naive Bayes.

Randomized algorithms have been playing an important role in computational statistics. The ability in simulation makes it beyond the deterministic algorithms. There must be more possibilities await!

Acknowledgements

Before the ending of this thesis, I want to express my gratitude to my supervisor, Dr. K.C. Cheung, for his patient guidance in the whole project. All the innovations mentioned in this report were jointly made by Dr. Cheung and me.

I have implemented all the algorithms in the three topics in this report by Python from scratch. The code are available from my GitHub at:

1. Gibbs Sampler on Vertex Coloring Problem: `https://github.com/jamespengcheng/MCMC-of-Final-Year-Project-HKU`

2. Simulated Annealing in Travelling Salesman Problem: `https://github.com/jamespengcheng/Simulated-Annealing-in-Traveling-Salesman-Problem`

3. Comonotonicity in Improving Naive Bayes Classifier: `https://github.com/jamespengcheng/Improvement-to-Naive-Bayes-Classifier`

References

1. Van Laarhoven, P. J. & Aarts, E. H. in *Simulated annealing: Theory and applications* 7–15 (Springer, 1987).
2. Casella, G. & George, E. I. Explaining the Gibbs sampler. *The American Statistician* **46,** 167–174 (1992).
3. LeCun, Y., Cortes, C. & Burges, C. J. The MNIST database of handwritten digits, 1998. *URL http://yann. lecun. com/exdb/mnist* **10,** 34 (1998).
4. Hand, D. J. & Yu, K. Idiot's Bayes—not so stupid after all? *International statistical review* **69,** 385–398 (2001).
5. Rish, I. *et al. An empirical study of the naive Bayes classifier* in *IJCAI 2001 workshop on empirical methods in artificial intelligence* **3** (2001), 41–46.
6. Dhaene, J., Denuit, M., Goovaerts, M. J., Kaas, R. & Vyncke, D. The concept of comonotonicity in actuarial science and finance: theory. *Insurance: Mathematics and Economics* **31,** 3–33 (2002).
7. Häggström, O. *et al. Finite Markov chains and algorithmic applications* (Cambridge University Press, 2002).
8. Mulder, S. A. & Wunsch II, D. C. Million city traveling salesman problem solution by divide and conquer clustering with adaptive resonance neural networks. *Neural Networks* **16,** 827–832 (2003).
9. Carlo, C. M. Markov chain monte carlo and gibbs sampling. *Lecture notes for EEB* **581** (2004).
10. Davis, J. & Goadrich, M. *The relationship between Precision-Recall and ROC curves* in *Proceedings of the 23rd international conference on Machine learning* (2006), 233–240.
11. Kaufman, L. & Rousseeuw, P. J. *Finding groups in data: an introduction to cluster analysis* (John Wiley & Sons, 2009).
12. Motwani, R. & Raghavan, P. *Randomized algorithms* (Chapman & Hall/CRC, 2010).
13. Davydov, A. Y. A Probabilistic Attack on NP-complete Problems. *arXiv preprint arXiv:1107.0098* (2011).
14. Bergstra, J. & Bengio, Y. Random search for hyper-parameter optimization. *Journal of machine learning research* **13,** 281–305 (2012).
15. Low, R. K. Y., Faff, R. & Aas, K. Enhancing mean–variance portfolio selection by modeling distributional asymmetries. *Journal of Economics and Business* **85,** 49–72 (2016).
16. Rubinstein, R. Y. & Kroese, D. P. *Simulation and the Monte Carlo method* (John Wiley & Sons, 2016).

17. Ruder, S. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).

18. Uci. *Adult Census Income* Oct. 2016. `https://www.kaggle.com/uciml/adult-census-income`.

19. Dua, D. & Graff, C. *UCI Machine Learning Repository* 2017. `http://archive.ics.uci.edu/ml`.

20. Papy, J. Stanford Encyclopedia of Philosophy (Spring 2019 Edition) (2019).

21. *Detector Performance Analysis Using ROC Curves - MATLAB & Simulink* `https://www.mathworks.com/help/phased/examples/detector-performance-analysis-using-roc-curves.html`. (Accessed on 05/02/2020).