Abgabedatum: 11.01.2016

Teichmann, Erik
Matrikelnummer: 761571

Simulated Annealing

1 Einführung

Im folgenden soll das „Traveling-Salesman“ Problem gelöst werden. Das Problem besteht darin, den kürzesten, geschlossenen Weg durch N Städte zu finden. Die numerische Lösung ist schwierig, da der naive Ansatz, alle möglichen Konfigurationen zu betrachten und dann die kürzeste auszuwählen nicht performant ist. Eine kurze Überlegung zeigt, dass die benötigte Zeit mit $N!$ ansteigt.
Stattdessen kann das Simulated-Annealing-Verfahren verwendet werden. Ähnlich zu einem Festkörper wird durch langsames Abkühlen ein energetisches Minimum gefunden. Dabei entspricht hier die Weglänge der Energie. Wichtig ist dabei, dass das System langsam genug abgekühlt wird. Im Festkörper führt das zu einer Kristallstruktur und hier zur Bestimmung der kürzesten Weglänge. Analog zum Festkörper wird auch hier das Abkühlen durch eine Boltzmann-Verteilung bestimmt.

2 Problemstellung

Die Aufgabe besteht darin, den kürzesten, geschlossenen Weg durch 76 deutsche Städte zu finden. Die Städte sind in der Form ihrer Längen- und Breitengrade gegeben. Der ideale Weg soll als Projektion dargestellt werden. Für das Simulated-Annealing sind zwei Flip-Arten vorgegeben:

(i) Zwei zufällige Städte werden vertauscht
(ii) Die Strecke zwischen zwei zufällig ausgewählten Städten wird umgedreht

Für beide Arten von Flips sollen die Wahrscheinlichkeitsdichten der Weglängen in Abhängigkeit des Cooling-Parameters bestimmt werden. Und letztendlich sollen die Ergebnisse in Abhängigkeit der Rechenzeit dargestellt werden.

3 Methodik

3.1 Simulated-Annealing

Zu Beginn des Simulated-Annealing gehen wir von einer zufälligen Konfiguration der Städte aus. Zuerst wird die Länge dieser Konfiguration bestimmt. Anschließend wird ein Flip durchgeführt, die Art des Flips ist dabei vorgegeben. Danach wird wieder die Länge bestimmt. Ist sie kleiner als vor dem Flip, wird die neue Konfiguration angenommen. Ist die neue Länge aber größer, als die vorherige, wird sie nicht gleich verworfen, stattdessen wird sie nach der Boltzmann-Verteilung angenommen. Das heißt die neue Konfiguration wird mit der Wahrscheinlichkeit p angenommen.

$$p = e^{-(L_{neu} - L_{alt})/T} \tag{1}$$

Stellt man sich das Verfahren als eine Bewegung auf dem Graphen vor, dann wird immer ein Schritt nach unten, also in Richtung des Minimums, gemacht. Dabei besteht die Gefahr, dass man in ein lokales Minimum fällt und sich dort nicht mehr heraus bewegt. Da aber auch Schritte

nach oben gemacht werden, kann so ein lokales Minimum wieder verlassen werden, um dann weiter in Richtung des globalen Minimums zu wandern. Solche Sprünge nach oben werden mit sinkender Temperatur immer unwahrscheinlicher, sodass nach genügend langer Zeit ein stabiler Zustand erreicht wird.
Nachdem eine bestimmte Anzahl dieser Flips ausgeführt wurde, hier $100N$ Flips, oder $10N$ erfolgreiche Fips, wird die Temperatur mit dem Cooling-Parameter verringert.

3.2 Simulation in einem C++-Programm

Zuerst werden die Koordinaten aus der Datei eingelesen. Dabei werden nur die Längen- und Breitengrade gespeichert. Im folgenden sind die absoluten Koordinaten aber uninteressant, nur der relative Abstand zwischen den Städten zählt. Es gibt zwei Möglichkeiten, entweder alle Distanzen werden nach jedem Flip neu berechnet, oder sie werden einmal berechnet und nach jedem Flip nur aufaddiert. Der zweite Weg scheint schneller zu sein, da für die Berechnung der Abstände immer wieder sin und cos berechnet werden müssen.
Jetzt müssen die Distanzen noch gespeichert werden. Am einfachsten ist es, sie in einem zweidimensionalen Array zu speichern, bei dem der Abstand zwischen den Städten i und j durch das Array-Element $[i][j]$ gegeben ist.
Um die nötigen Distanzen zu bestimmen wird die Haversine-Formel verwendet. Für die Längengrade φ und die Breitengrade λ ist der Abstand gegeben durch:

$$l = 2R \arcsin\left(\sqrt{\sin\left(\frac{\varphi_2 - \varphi_1}{2}\right)^2 + \cos(\varphi_1)\cos(\varphi_2)\sin\left(\frac{\lambda_2 - \lambda_1}{2}\right)^2}\right) \tag{2}$$

Damit kann die Länge und die Reihenfolge der besuchten Städte durch ein Folge der Zahlen 0 bis $N-1$ charakterisiert werden. Diese Folge nennen wir im Folgenden die Konfiguration.
Um das Simulated-Annealing durchzuführen wird die Konfiguration zufällig erstellt und die zugehörige Länge wird berechnet. Dann wird der vorgegeben Flips durchgeführt. Dazu müssen zwei zufällige Zahlen zwischen 0 und $N-1$ erzeugt werden. Für die erste Art von Flip werden die zwei Elemente an diesen Stellen vertauscht. Für die zweite Art wurde die Konvention gewählt, dass die Elemente dazwischen umgedreht werden, nicht aber die Elemente an den bestimmten Stellen. Würde man die Konvention so ändern, dass auch die Randelemente umgedreht werden, dann wäre der erste Flip eine Spezialfall des zweiten, und das soll verhindert werden, um die Eigenschaften der beiden Arten von Flips besser untersuchen zu können.
Dann wird das Simulated-Annealing wie vorhin beschrieben durchgeführt. Bei der Temperatur muss darauf geachtet werden, dass der Startwert weit über der durchschnittlichen Änderung nach einem Flip liegt.

3.3 Verwendete Parameter

Bei dem Simulated-Annealing gibt es einige Parameter, die variiert werden können.
Die Temperatur wurde bei 10000 gestartet. Damit ist sie etwa 10-mal so groß, wie eine typische Längenänderung nach einem Flip. Und das System wurde bis 1 abgekühlt, unabhängig von dem gewählten Cooling-Parameter.
Die Anzahl der Flips bis zum Verringern der Temperatur wurden zu $100N$ Flips, oder $10N$ erfolgreiche Flips gewählt. Diese Werte folgen den vorgeschlagenen Werten aus *Numerical Recipes*.
Um die Ergebnisse auswerten zu können, werden mehrere Messwerte für jeden Cooling-Parameter benötigt. Um auch noch Aussagen über die Statistik zu erhalten, müssen viele Messwerte berechnet werden. In der Simulation wurden für jeden Cooling-Parameter 100 Messwerte erzeugt.

Und als letzten Parameter gibt es noch die verwendeten Cooling-Parameter. Es wurden die Werte 0.5, 0.9, 0.95, 0.98, 0.99 und 0.995 verwendet. So kann eine große Breite von Cooling-Parametern abgedeckt werden, wobei der Schwerpunkt deutlich auf den Großen liegt.

4 Auswertung

4.1 Kürzester Weg

Der kürzeste gefundene Weg ist $l_{min} = 4292.14\,\text{km}$, er wurde mit Flip-Typ 2 und Cooling-Parameter 0.98 erreicht. Das erscheint interessant, da man das beste Ergebnis auch von dem größten Cooling-Parameter, also 0.995, erwarten würde. Aber nach genügend langer Laufzeit würde man auch dort diesen Weg finden. Der gefundene Weg ist in Abbildung 1 dargestellt.

4.2 Verteilung

Im weiteren Verlauf soll jetzt noch die Wahrscheinlichkeitsdichte für verschiedene Cooling-Parameter und Flip-Typen bestimmt werden. Für eine ideale Simulation wäre die Verteilung eine Delta-Verteilung bei l_{min}, diese Verteilung ist aber für die hier durchgeführten Simulationen nicht zu erwarten.
Als erste Näherung kann eine Gauss-Verteilung angenommen werden. Dabei muss aber darauf geachtet werden, dass Längen, die kleiner als l_{min} sind, nicht erreicht werden können. Die Gauss-Verteilung muss also bei l_{min} abgestumpft werden. Damit die Verteilung aber weiter normiert bleibt, muss die sie mit dem Flächeninhalt des abgeschnitten Stücks multipliziert werden. Dass ist die „truncated“ Gauss-Verteilung $\varphi \sim t\mathcal{N}(\mu, \sigma^2)$ und sie ist durch die Normal-Verteilung $\varphi_G \sim \mathcal{N}(1, 0)$ und ihre kumulative Verteilung Φ_G definiert. Hierbei muss allerdings beachtet werden, dass μ und σ den Größen der Gauss-Verteilung entsprechen und nicht dem Mittelwert und der Varianz der „truncated“ Gauss-Verteilung. Außerdem spielt hier immer wieder der Term $\dfrac{l_{min} - \mu}{\sigma}$ eine Rolle, deswegen wird er durch λ abgekürzt.

$$\varphi(x) = \frac{\frac{1}{\sigma}\varphi_G\left(\frac{x-\mu}{\sigma}\right)}{1 - \Phi_G(\lambda)} \tag{3}$$

Damit ergibt sich die kumulative Verteilung

$$\Phi(x) = \frac{\Phi_G\left(\frac{x-\mu}{\sigma}\right)}{1 - \Phi_G(\lambda)} \tag{4}$$

Der Mittelwert $\bar{\mu}$ und die Varianz $\bar{\sigma}^2$ der „truncated“ Gauss-Verteilung ist durch μ und σ vollkommen bestimmt.

$$\bar{\mu} = \mu + \sigma\frac{\varphi_G(\lambda)}{1 - \Phi_G(\lambda)} \tag{5}$$

$$\bar{\sigma}^2 = \sigma^2\left[1 + \frac{\lambda\varphi_G(\lambda)}{1 - \Phi_G(\lambda)} - \frac{\varphi_G(\lambda)^2}{(1 - \Phi_G(\lambda))^2}\right] \tag{6}$$

Um die Wahrscheinlichkeitsdichte aus gegebenen Messwerten zu bestimmen gibt es verschiedene Methoden. Hier wird die Dichte aus der kumulativen Verteilung bestimmt. Dazu wird die gemessene Verteilung gefittet und aus den Fit-Parametern ergibt sich direkt die Wahrscheinlichkeitsdichte. Die kumulativen Verteilungen sind in den Abbildungen 2 und 3 dargestellt.

In den Abbildungen sieht man schon deutlich, dass die Verteilungen bei Flip-Typ 2 dichter liegen, als bei Flip-Typ 1. Das heißt, bei kleinem Cooling-Parameter ist es sinnvoller den Flip-Typ 2 zu wählen. Die, sich ergebenden Wahrscheinlichkeitsdichten, sind in den Abbildungen 4 und 5 dargestellt.

	Flip-Typ 1		Flip-Typ 2	
Cooling-Parameter	$\bar{\mu}$	$\bar{\sigma}$	$\bar{\mu}$	$\bar{\sigma}$
0.5	5552.5	296.6	4698.1	156.6
0.9	4909.4	202.7	4580.6	126.4
0.95	4788.7	174.3	4572.7	143.1
0.98	4642.1	125.4	4534.7	114.9
0.99	4590.5	113.7	4567.4	136.9
0.995	4522.3	101.5	4459.7	100.6

Betrachtet man die Mittelwerte zeigt sich, dass die des Flip-Typ 2 immer kleiner sind, als die des Flip-Typ 1. Außerdem ist meistens die Varianz kleiner. Damit scheint, unabhängig von der Rechenzeit, der Flip-Typ 2 zu besseren Ergebnissen zu führen.
Das zeigt sich auch, wenn man die mittlere Weglänge in Abhängigkeit des Cooling-Parameters untersucht. Diese sind in der Abbildung 6 dargestellt. Hier kann man wieder den Vorteil des Flip-Typ 2 bei kleinen Cooling-Parametern erkennen. Die Punkte konnten durch eine Funktion der Form $f(x) = a\sqrt{1-x}+b$ gefittet werden. Das heißt im Grenzfall des Cooling-Parameters gegen 1 ist die mittlere Weglänge= b. Interessanterweise ist aber b bei beiden Flip-Typen deutlich größer als l_{min}, sogar um etwa 200 km. Das könnte entweder an der zu geringen Zahl der Wiederholungen liegen, oder daran, dass das Gesetz für Cooling-Parameter nahe 1 nicht mehr gilt. Oder aber, dass für diese Flip-Typen eine ideale Delta-Verteilung nicht erreicht werden kann, unabhängig von dem Cooling-Parameter. Zuletzt sollte noch kurz darauf eingegangen werden, dass b bei Flip-Typ 2 größer ist, als bei Flip-Typ 1. Das liegt vermutlich an dem Aussreißer bei dem Cooling-Parameter 0.99. Würde man diesen Wert herausnehmen, wären die beiden Schnittpunkte bei 1 etwa gleich groß.

4.3 Rechenzeit

Bis jetzt ist bekannt, dass der Flip-Typ 2 zu kürzeren Weglängen führt. Aber wie hoch ist der Preis in der Rechenzeit? Heuristisch ist der Flip-Typ 2 eine drastische Veränderung der Konfiguration. Das heißt, die Weglänge nach dem Flip ist nicht so stark von der Konfiguration vor dem Flip abhängig, wie bei einer kleinen Veränderung des Flip-Typs 1. Damit kann man auch von einer ungünstigen Lage aus direkt in Richtung eines Minimums springen. Gleichzeitig kann der Sprung auch weit weg von einem Minimum führen, und das herunterrollen in das Minimum wird unwahrscheinlicher. Das heißt aber, dass auch mehr Flips durchgeführt werden müssen, als beim Flip-Typ 1. Damit erhöht sich also auch die Rechenzeit. Das sieht man in der Abbildung 7.
Für kleine Cooling-Parameter unterscheiden sich die Zeiten nicht. Das sieht man auch an der gefitteten Funktion $g(x) = a\tan(bx)$. Dabei unterscheiden sich die beiden a kaum. Für große Cooling-Parameter nahe bei 1 sieht man aber, dass die Differenz der durchschnittliche Rechenzeiten immer größer wird. Der Flip-Typ 2 wird immer aufwendiger. Das zeigt sich auch an der gefitteten Funktion. Die Funktion $\tan(bx)$ geht für $x \to 1$ gegen ∞, und das sogar sehr schnell. Damit macht sich der kleine Unterschied zwischen den beiden b stark bemerkbar, obwohl er erst

in der 4. Nachkommastelle ist.

4.4 Ergebnisse

Das Ziel der Simulation war es, den kürzesten Weg durch die angegebene Städte zu finden. Aber gleichzeitig soll auch die Methode des Simulated Annealing untersucht werden. Der kürzeste, gefundene Weg beträgt 4292.14 km. Anhand der Daten hat sich gezeigt, dass die Rechenzeit um den kürzesten Weg zu finden mit dem Cooling-Parameter ansteigt, dieser Anstieg ist nicht linear, sondern scheint einem $\tan(x)$-Gesetz zu folgen.
Gleichzeitig nimmt mit steigendem Cooling-Parameter aber auch die mittlere Weglänge ab. Diese folgt einem $\sqrt{1-x}$-Gesetz. Das kann man allgemein zusammenfassen zu: Ein kürzerer Weg braucht eine längere Rechenzeit.
Über das genaue Ergebnis entscheidet der Flip-Typ maßgeblich. Allgemein hat sich gezeigt, dass die Ergebnisse des Flip-Typ 1 etwas länger sind, dafür aber auch eine kürzere Rechenzeit benötigen. Für einen kleinen Cooling-Parameter bedeutet das einen großer Nachteil. Hier ist die zeitliche Differenz zwischen den beiden Flip-Typen sehr klein, aber die Längendifferenz ist vergleichsweise groß. Ist aber der Cooling-Parameter sehr groß, also nahe bei 1, dann wird der Flip-Typ 1 effizienter. Die Ergebnisse sind nur wenige Kilometer länger als mit dem Flip-Typ 2, die benötigte Zeit ist aber deutlich kürzer. Für den Cooling-Parameter 0.995 ist die Rechenzeit nur etwa 75% so groß, wie die des Flip-Typ 2, der Mittelwert des berechneten Weges ist aber nur 70 km ($\approx 1.5\%$) länger.
In Konklusion kann man sagen: Wenn die Kürze des Weges nur zweitrangig ist, kann ein kleiner Cooling-Parameter mit Flip-Typ 2 gewählt werden. Die Ergebnisse liegen im Durschnitt 10% über dem kürzesten Weg. Ist aber jeder Kilometer entscheidend sollte ein sehr großer Cooling-Parameter gewählt werden. Spielt die Zeit keine Rolle, können mit dem Flip-Typ 2 die besten Ergebnisse erreicht werden, aber am zeitlich effizientesten ist hier der Flip-Typ 1.

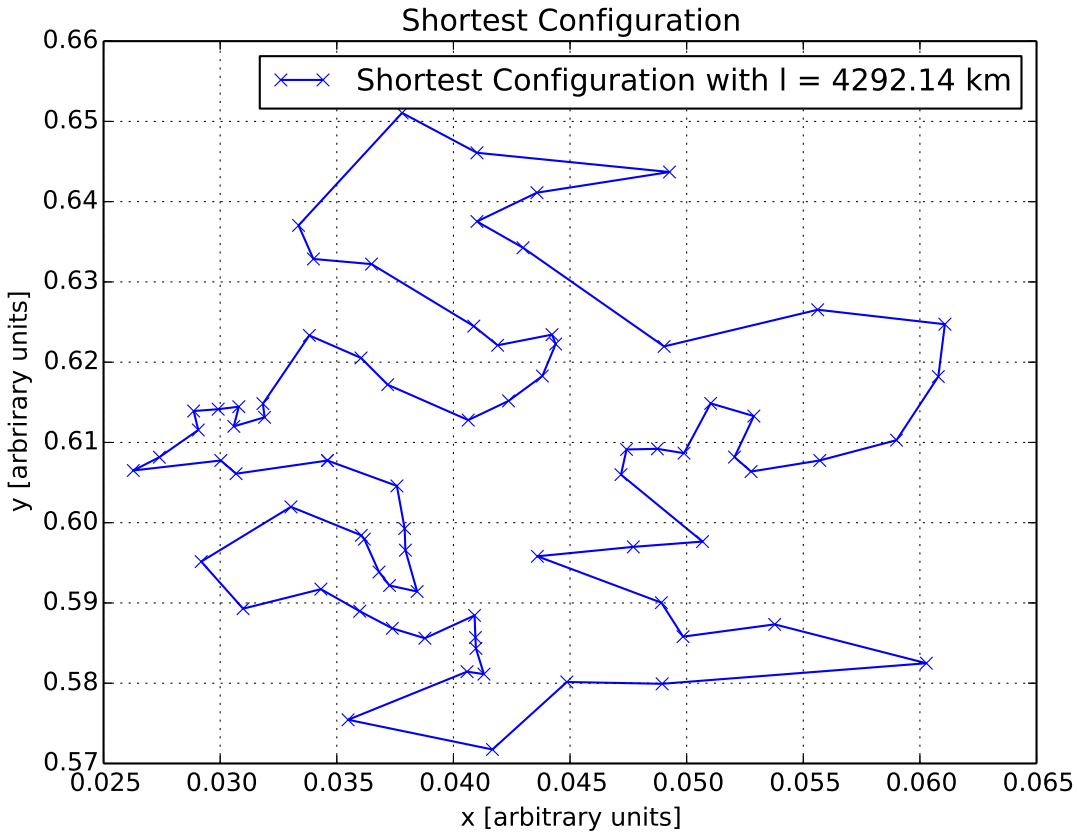

Abbildung 1: Kürzester, gefundener Weg mit Flip-Typ 2 und Cooling-Parameter 0.98 bei 76 deutschen Städten

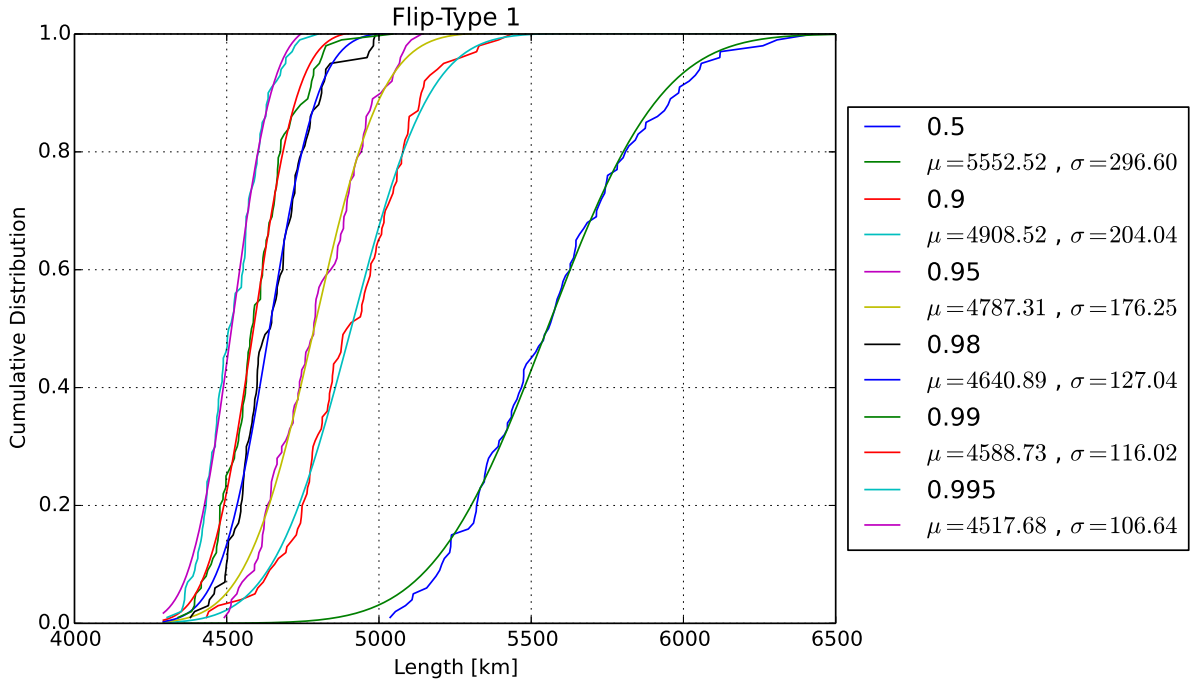

Abbildung 2: Kumulative „truncated“ Gauss-Verteilung für Flip-Typ 1

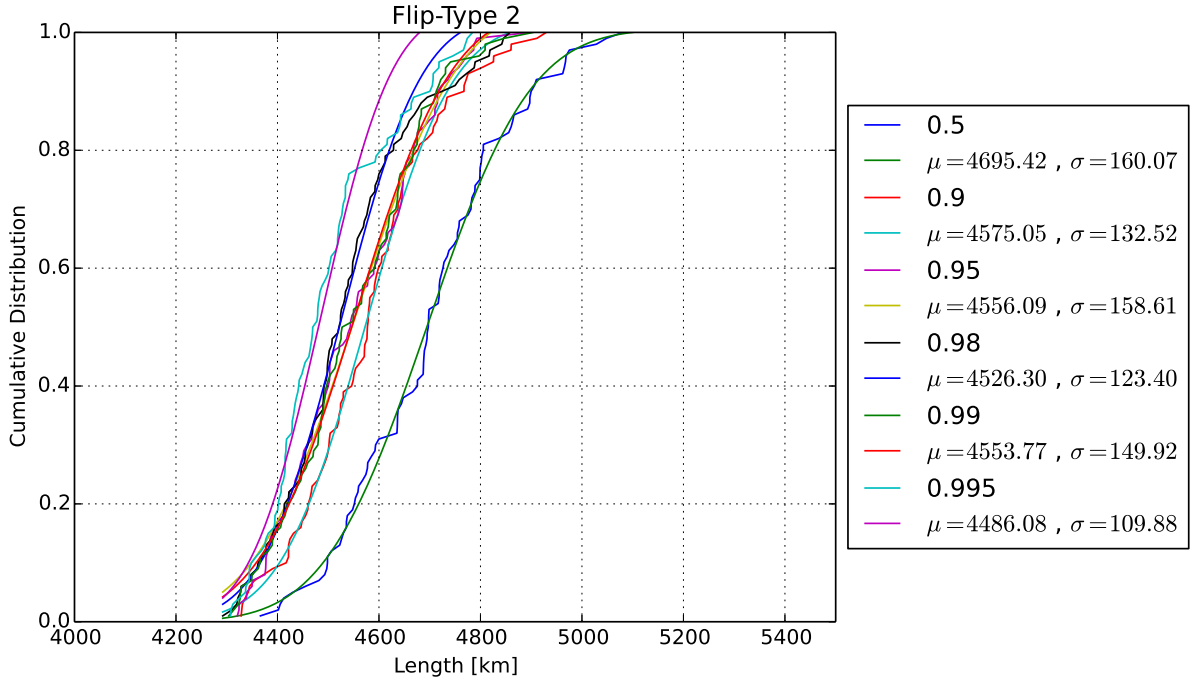

Abbildung 3: Kumulative „truncated“ Gauss-Verteilung für Flip-Typ 2

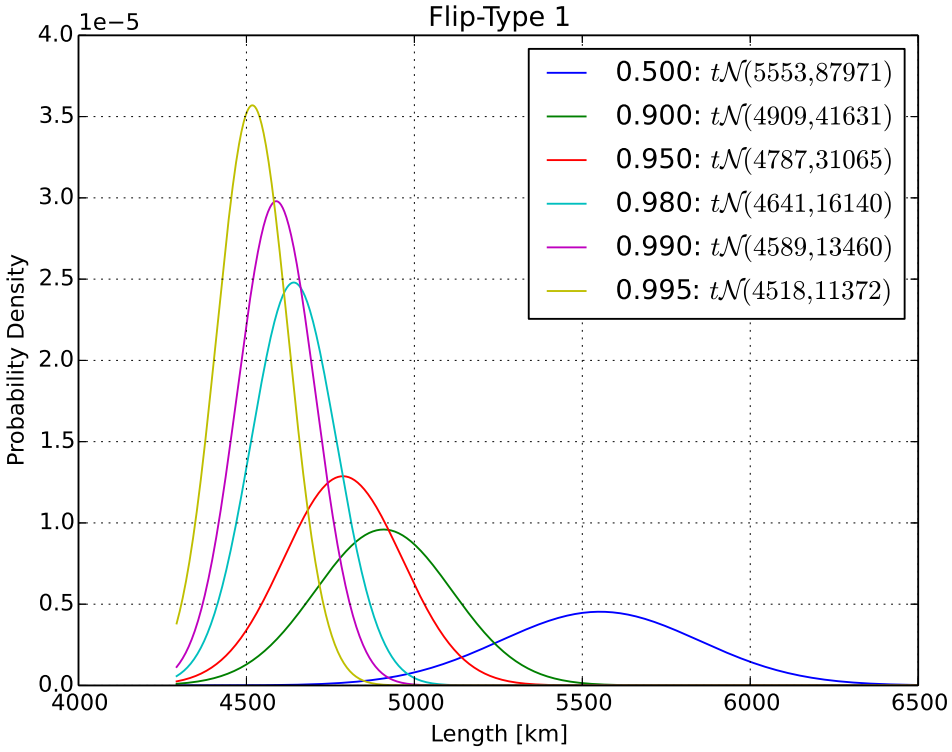

Abbildung 4: „truncated“ Gauss-Verteilung für Flip-Typ 1

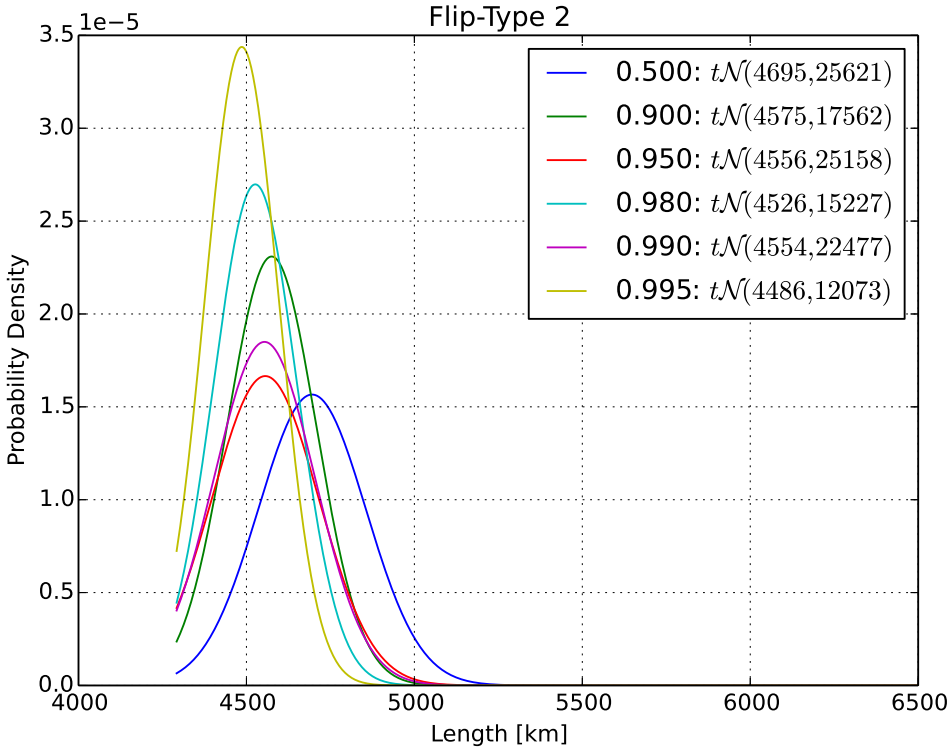

Abbildung 5: „truncated“ Gauss-Verteilung für Flip-Typ 2

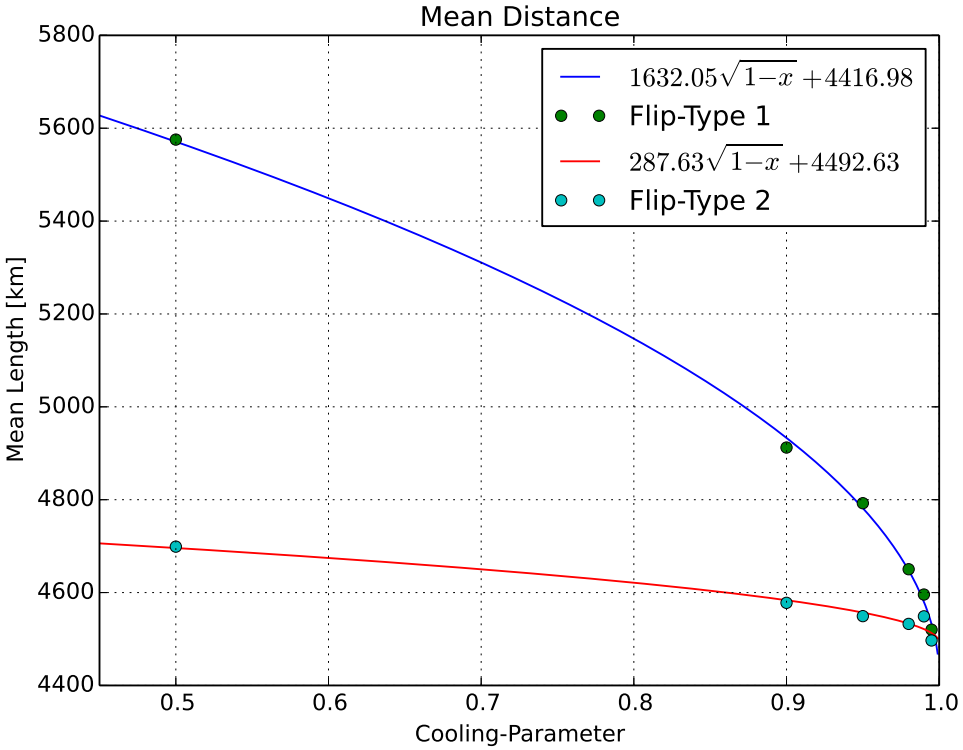

Abbildung 6: Mittlere Weglänge für beide Flip-Typen in Abhängigkeit des Cooling-Parameters

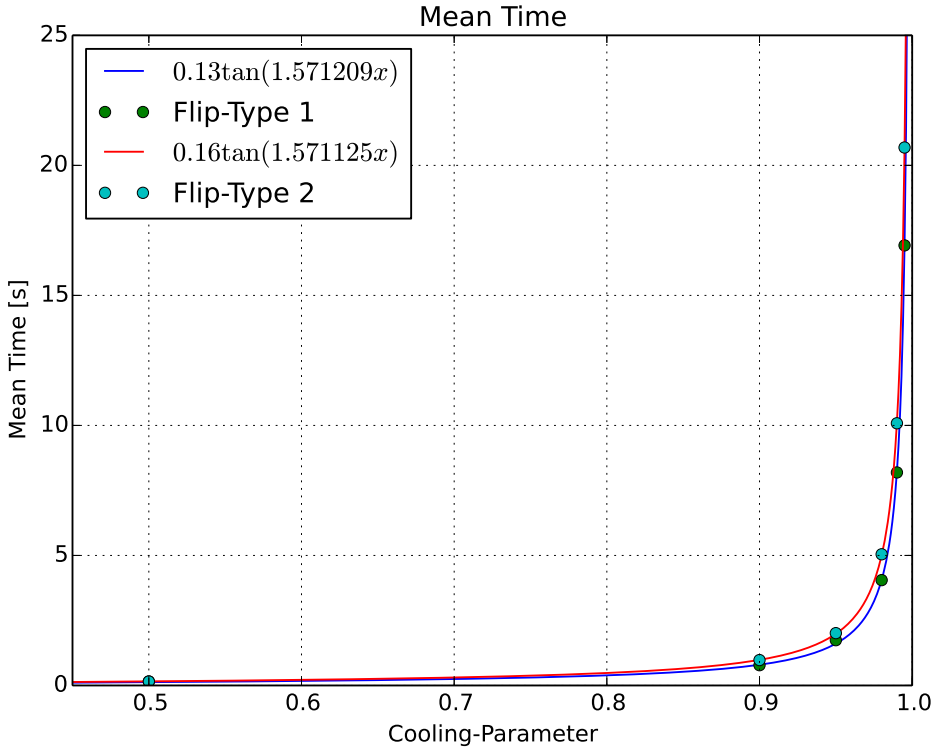

Abbildung 7: Mittlere Zeit für beide Flip-Typen in Abhängigkeit des Cooling-Parameters

5 Code

```
//
//  main.cpp
//  Simulated_Annealing
//
//  Created by Erik Teichmann on 04/12/2015.
//

#include <iostream>
#include <fstream>
#include <vector>
#include <random> // std::mt19937_64
#include <chrono> // std::chrono
#include <string> // std::to_string
#include "simulated_annealing.hpp"
#include "file_operations.hpp"

typedef std::vector<std::vector<double>> coord_matrix;

int main() {

  // Parameters
  std::vector<double> cooling_parameter = {0.995, 0.99, 0.98, 0.95, 0.9, 0.5};
  int number_repetitions = 100;

  // Input File
  std::ifstream coordinates_file ("../../Tabelle_deutsche_Staedte.txt");

  long long seed = std::chrono::system_clock::now().time_since_epoch().count();
  std::mt19937_64 generator(seed);

  coord_matrix coordinates = GetCoordinatesFromFile(coordinates_file);
  coord_matrix distances = DistanceBetweenCities(coordinates);
  std::vector<int> configuration;
  for (int i = 0; i < coordinates.size(); i++){
    configuration.push_back(i);
  }

  // Iterate over both types of flips
  for (int flip_type = 1; flip_type <= 2; flip_type ++) {
    // Iterate over all cooling Parameters
    for (int i = 0; i < cooling_parameter.size(); i++){
      // Get multiple results for statistical analysis
      for (int j = 0; j < number_repetitions; j ++) {
        RandomizeConfiguration(generator, configuration);
        double length = CalculateLength(configuration, distances);
        // We want to know, how long each run takes
        std::chrono::system_clock::time_point start =
                                              std::chrono::system_clock::now();
        for (double temp = 10000; temp > 1; temp = temp*cooling_parameter[i]){
          SimulatedAnnealingStep(temp, flip_type, distances, generator, length,
                                 configuration);
        }
        std::chrono::system_clock::time_point end =
                                              std::chrono::system_clock::now();
        double duration =std::chrono::duration_cast
                               <std::chrono::microseconds>(end-start).count();
        std::string file_name = "../../Data/" + std::to_string(flip_type) +
```

```
                                "_" + std::to_string(cooling_parameter[i]) +
                                ".dat";
        std::ofstream data_file (file_name.c_str(), std::ofstream::app);
        WriteToFile(duration, length, configuration, coordinates,
                    data_file);
        data_file.close();
        // Just to make sure everything is running smoothly
        std::cout << "Flip-Type = " << flip_type << ", Cooling-Parameter = " <<
                     cooling_parameter[i] << ", Iteration = " << j <<
                     ", Length = " << length << std::endl;
      }
    }
  }

  return 0;
}
```

```
//
//  circle_functions.hpp
//  Simulated_Annealing
//
//  Created by Erik Teichmann on 04/12/2015.
//

#ifndef circle_functions_hpp
#define circle_functions_hpp

#include <iostream>
#include <cmath>
#include <vector>
#include <algorithm> // std::min
#include <assert.h> // assert

typedef std::vector<std::vector<double>> coord_matrix;

std::vector<double> ArcToRad(std::vector<double> &arc);
double DistanceLatLon(const std::vector<double> &coord1,
                      const std::vector<double> &coord2);

#endif /* circle_functions_hpp */

//
//  circle_functions.cpp
//  Simulated_Annealing
//
//  Created by Erik Teichmann on 04/12/2015.
//
#include "circle_functions.hpp"

/* Converts an array consisting of [degree, arcmin, arcsec] to Radians */
std::vector<double> ArcToRad(std::vector<double> &arc){
  assert(arc.size() == 3);
  double rad = (arc[0] + arc[1]/60 + arc[2]/3600)*M_PI/180;
  arc.resize(1);
  arc[0] = rad;
  return arc;
}

/* Calculates the distance between two points given by latitude and longitude
 by using the haversine algorithm */
double DistanceLatLon(const std::vector<double> &coord1,
                      const std::vector<double> &coord2){
  // Make sure all coordinates are in the right format
  assert(coord1.size() == 2 && coord2.size() == 2);
  assert(0 <= coord1[0] <= 2*M_PI && 0 <= coord1[1] <= 2*M_PI &&
         0 <= coord2[0] <= 2*M_PI && 0 <= coord2[1] <= 2*M_PI);
  double r_earth = 6371; // Radius of the Earth in km
  double hav_d_over_r = sin((coord2[0] - coord1[0])/2)
                        *sin((coord2[0] - coord1[0])/2)
                        + cos(coord1[0])*cos(coord2[0])
                          *sin((coord2[1] - coord1[1])/2)
                          *sin((coord2[1] - coord1[1])/2);
  // Make sure, that there weren't any rounding errors, that would break the
  // asin function
  return 2*r_earth*asin(std::min(1. ,sqrt(hav_d_over_r)));
}
```

```
//
//  file_operations.hpp
//  Simulated_Annealing
//
//  Created by Erik Teichmann on 07/12/2015.
//

#ifndef file_operations_hpp
#define file_operations_hpp

#include <iostream>
#include <fstream>
#include <assert.h> //assert
#include "circle_functions.hpp"
#include <sstream> //std::istringstream
#include <vector>

typedef std::vector<std::vector<double>> coord_matrix;

coord_matrix GetCoordinatesFromFile(std::ifstream &coordinates_file);
void WriteToFile(const double &running_time, const double &length,
                 const std::vector<int> &configuration,
                 const coord_matrix &coordinates, std::ofstream &data_file);

#endif /* file_operations_hpp */

//
//  file_operations.cpp
//  Simulated_Annealing
//
//  Created by Erik Teichmann on 07/12/2015.
//

#include "file_operations.hpp"

/* Takes a vector of strings of the form
 {cityname, lat_deg, lat_min, lat_s, N, lon_deg, lon_min, lon_s, O}
 and returns the latitude and longitude converted to radians*/
std::vector<double> StringToDouble(
    const std::vector<std::string> &coordinates_string){
  // Make sure the string has the correct format, since it would result in empty
  // coordinates
  assert(coordinates_string.size() == 9);
  std::vector<double> lat {std::stod(coordinates_string[1]),
    std::stod(coordinates_string[2]),
    std::stod(coordinates_string[3])};
  std::vector<double> lon {std::stod(coordinates_string[5]),
    std::stod(coordinates_string[6]),
    std::stod(coordinates_string[7])};
  return std::vector<double> {ArcToRad(lat)[0], ArcToRad(lon)[0]};
}

/* Takes a string of the form
 cityname lat_deg lat_min lat_s N lon_deg lon_min lon_s O
 and returns the latitude and longitude converted to radians*/
std::vector<double> GetCoordinatesFromString(const std::string &line){
  // Make sure there wasn't an empty line somewhere, since it would result in
  // empty coordinates and thus a faulty simulation
  assert(line.find_first_not_of(' ') != std::string::npos);
```

```
  std::stringstream strstream(line);
  std::string buf;
  std::vector<std::string> coordinates_string;
  while (strstream >> buf) {
    coordinates_string.push_back(buf);
  }
  return StringToDouble(coordinates_string);
}

/* Reads a csv-file of the form
 cityname lat_deg lat_min lat_s N lon_deg lon_min lon_s O
 and returns a 2d-vector of latitude and longitude of the form
 {{lat1, lon1}, {lat2, lon2}, ...}*/
coord_matrix GetCoordinatesFromFile(std::ifstream &coordinates_file){
  assert(coordinates_file.is_open());
  std::string line;
  coord_matrix coordinates;
  while (std::getline(coordinates_file, line)){
    coordinates.push_back(GetCoordinatesFromString(line));
  }
  return coordinates;
}

/* Writes the results to file */
void WriteToFile(const double &running_time, const double &length,
                 const std::vector<int> &configuration,
                 const coord_matrix &coordinates, std::ofstream &data_file){
  data_file << running_time << " " << length ;
  for (int i = 0; i < coordinates.size(); i++) {
    data_file << " " << coordinates[configuration[i]][0] << " " <<
                 coordinates[configuration[i]][1];
  }
  data_file << std::endl;
}
```

```
//
//  simulated_annealing.hpp
//  Simulated_Annealing
//
//  Created by Erik Teichmann on 07/12/2015.
//

#ifndef simulated_annealing_hpp
#define simulated_annealing_hpp

#include <iostream>
#include <random> // std::mt19937_64
#include <algorithm> // std::shuffle, std::iter_swap
#include <assert.h> // assert
#include <vector>
#include <cmath>
#include "circle_functions.hpp"

typedef std::vector<std::vector<double>> coord_matrix;

coord_matrix DistanceBetweenCities(const coord_matrix &coordinates);
void RandomizeConfiguration(std::mt19937_64 &generator,
                            std::vector<int> &configuration);
double CalculateLength(const std::vector<int> &configuration,
                       const coord_matrix &distances);
void SimulatedAnnealingStep(const double &temperature, const int &flip_type,
                            const coord_matrix &distances,
                            std::mt19937_64 &generator, double &length,
                            std::vector<int> &configuration);

#endif /* simulated_annealing_hpp */

//
//  simulated_annealing.cpp
//  Simulated_Annealing
//
//  Created by Erik Teichmann on 07/12/2015.
//

#include "simulated_annealing.hpp"

/* Calculates the distance between all citys */
coord_matrix DistanceBetweenCities(const coord_matrix &coordinates){
  // We don't want to calculate with no citys
  assert(!coordinates.empty());
  coord_matrix distances(coordinates.size());
  for (int i = 0; i < coordinates.size(); i++) {
    for(int j = 0; j < coordinates.size(); j++){
      distances[i].push_back(DistanceLatLon(coordinates[i], coordinates[j]));
    }
  }
  return distances;
}

/* Randomizes a given array of integers */
void RandomizeConfiguration(std::mt19937_64 &generator,
                            std::vector<int> &configuration){
  std::shuffle(configuration.begin(), configuration.end(), generator);
}
```

```
/* Calculates the length of the path, given by the configuration */
double CalculateLength(const std::vector<int> &configuration,
                       const coord_matrix &distances){
  double length = 0;
  for (int i = 0; i < configuration.size()-1; i++) {
    length += distances[configuration[i]][configuration[i+1]];
  }
  length += distances[configuration[configuration.size()-1]][configuration[0]];
  return length;
}

/* Choose two random cities from the configuration and return them, ordered by
 size */
std::vector<int> ChooseTwoRandomCities(std::mt19937_64 &generator,
                                       const std::vector<int> &configuration){
  std::uniform_int_distribution<> dis(0, int(configuration.size()-1));
  int city_1 = dis(generator);
  int city_2 = dis(generator);
  // We don't want to swap the city with itself
  while(city_2 == city_1){
    city_2 = dis(generator);
  }
  if (city_1 < city_2) {
    return std::vector<int> {city_1, city_2};
  }
  return std::vector<int> {city_2, city_1};
}

/* Flips two randomly chosen citys in the configuration */
std::vector<int> FlipType1(std::mt19937_64 &generator,
                           const std::vector<int> &configuration){
  std::vector<int> new_configuration = configuration;
  std::vector<int> cities = ChooseTwoRandomCities(generator, new_configuration);
  std::iter_swap(new_configuration.begin()+cities[0],
                 new_configuration.begin()+cities[1]);
  return new_configuration;
}

/* Reverse the sequence between two randomly chosen cities */
std::vector<int> FlipType2(std::mt19937_64 &generator,
                           const std::vector<int> &configuration){
  std::vector<int> new_configuration = configuration;
  std::vector<int> cities = ChooseTwoRandomCities(generator, new_configuration);
  // We don't want to want a flip, where nothing changes, so between the two
  // cities mus lie more than one other city
  while (cities[1] == cities[0] + 2) {
    cities = ChooseTwoRandomCities(generator, new_configuration);
  }
  // We need to know, how many cities need to be swapped, excluding the
  // randomly chosen
  int length_swap_sequence = cities[1] - cities[0] - 2;
  for(int i = 0; i <= length_swap_sequence/2.; i++){
    std::iter_swap(new_configuration.begin() + cities[0] + 1 + i,
                   new_configuration.begin() + cities[1] - 1 - i);
  }
  return new_configuration;
}
```

```
/* Uses the Boltzman-distirbution, to decide, whether to switch the
 configuration, or to keep the old one */
bool DecideToSwitchConfigurations(const double &l1, const double &l2,
                                  const double &temperature,
                                  std::mt19937_64 &generator){
  // If the new way is shorter, we always want to take it
  if (l1 > l2) {
    return true;
  }
  std::uniform_real_distribution<double> uniform_distribution(0.0,1.0);
  double random_number = uniform_distribution(generator);
  double probability = exp(-(l2-l1)/temperature);
  return random_number < probability;
}

std::vector<int> ChooseFlipType(const int &flip_type,
                                const std::vector<int> &configuration,
                                std::mt19937_64 &generator){
  assert(flip_type == 1 || flip_type == 2);
  if (flip_type == 1){
    return FlipType1(generator, configuration);
  }
  else{
    return FlipType2(generator, configuration);
  }
}

/* Make one step in the Simulated Annealing with constant temperature*/
void SimulatedAnnealingStep(const double &temperature, const int &flip_type,
                            const coord_matrix &distances,
                            std::mt19937_64 &generator, double &length,
                            std::vector<int> &configuration){
  std::vector<int> new_configuration = configuration;
  double new_length = length;
  int succesful_flips = 0;
  // We stop the step, if there were either 100N steps, or 10N succesful steps
  for (int i = 0; i < 100*configuration.size() &&
       succesful_flips <= 10*configuration.size(); i ++) {
    new_configuration =  ChooseFlipType(flip_type, configuration, generator);
    new_length = CalculateLength(new_configuration, distances);
    if (DecideToSwitchConfigurations(length, new_length, temperature,
                                     generator)){
      configuration = new_configuration;
      length = new_length;
      succesful_flips += 1;
    }
  }
}
```