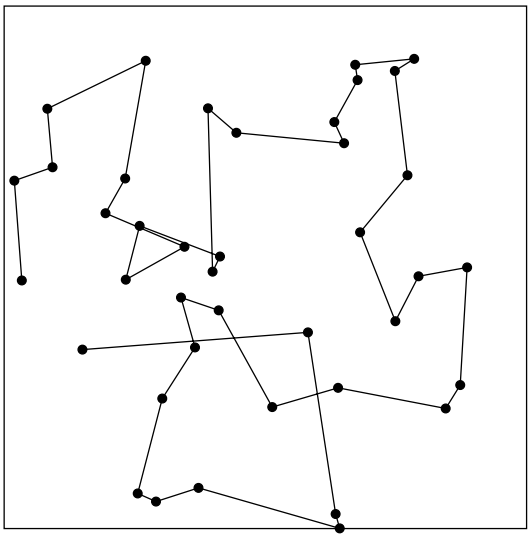
Karl-Franzens Universität Graz
Institut für Physik

Projekt für Computergrundkenntnisse und Programmieren

(Aufgabe 9.2)

Lösung des Travelling Salesman Problem durch Simulated Annealing



Andreas Windisch

Graz, am 21. Jänner 2009

Inhaltsverzeichnis

1 Motivation-Projektbeschreibung

1.1 Motivation

Zunächst will ich ein paar Worte über die Motivation zu diesem Projekt verlieren. Der Grund warum ich mich für dieses Problem entschieden habe ist ein sehr pragmatischer: In einem anderen Kurs den ich besuche (Computational Physics bei Professor Gattringer) behandeln wir Monte-Carlo-Simulationen im Allgemeinen, Gittersimulationen, insbesondere das Potts-Modell, im Speziellen.
Im Zuge der Diskussion der Anwendungsmöglichkeiten wurde auch kurz die Methode des 'Simulated Annealing' besprochen. (Der Begriff *Annealing* kommt aus der Stahlindustrie und bedeutet etwa 'Anlassen', 'Tempern'). Da dies also dem anderen Thema relativ nahe steht, und mir somit zu einem besseren Verständnis der Zusammenhänge verhilft, habe ich mich entschlossen dieses Thema aufzugreifen.
Ein weiterer Grund der sehr dafür spricht ist die Vielfalt dieses Problems: Die Problemstellung ist sehr einfach und kann von jedermann sofort verstanden werden. Die Lösung jedoch, wie so oft bei derlei Dingen, ist nicht leicht zu bekommen. Das Problem ist daher besonders reizvoll, und soll im Folgenden diskutiert werden. Die Umsetzung der Lösung in das C++ Programm wird im Detail besprochen. Der Sourcecode befindet sich schließlich im Anhang zur Gänze dargestellt.

1.2 Projektbeschreibung

Bei der Problemstellung wurde von folgendem Szenario ausgegangen:

- In Europa gibt es N Städte in denen Messen veranstaltet werden (alle Messen finden permanent statt).

- Diese N Städte sollen von unserem Handlungsreisenden, sagen wir ein Staubsaugervertreter, in einer Rundreise besucht werden.

- Der Staubsaugervertreter ist auf Effizienz bedacht und will aus Kosten- und Zeitersparnis die kürzeste Rundreise nehmen, wobei seine Start-/Ziel-Stadt fix vorgegeben ist.

- Wie stellt er das an?

2 Theoretischer Hintergrund

2.1 Monte-Carlo-Methode

In der statistischen Physik (und auch in vielen anderen Gebieten, wie z.B. bei dieser Problemstellung) haben Probleme oft die unangenehme Eigenschaft, dass ihre Mengen an Konfigurationen selbst auf winzigen Skalen von derart großer Kardinalität sind, dass jeder Versuch das Problem analytisch zu lösen von Vornherein ob der vielen Möglichkeiten zum Scheitern verurteilt ist.
Dennoch ist man an der Lösung solcher Probleme interessiert, da zumeist ebensolche Fragestellungn besonders reizvoll und interessant sind. Es gibt dennoch Möglichkeiten mit dieser Problematik fertig zu werden:
Sogenannte Monte-Carlo-Simulationen sind das Mittel der Wahl.
Der Name erzeugt bereits gewisse Assoziationen von rollenden Roulettkugeln, die wiederrum Betrachtungen von Wahrscheinlichkeit und Statistik induzieren. Diese Assoziationen sind absolut berechtigt: Denn man rückt der Problemstellung eben mit solchen Methoden zu Leibe, und kann nun statistische Aussagen über das zu untersuchende System treffen. Am besten sieht man die Wirkungsweise und Idee einer Sache anhand eines Beispiels. So will ich nun zuerst ein Beispiel einer Gittersimulation (das Ising-Modell), und im Anschluss das konkrete Beispiel des Travelling Salesman Problem ins Licht der Statistik rücken.

2.1.1 Beispiel: Das Ising-Modell

Das Ising-Modell beschreibt ein mikroskopisches Modell eines Magneten. Dabei wird auf einem d-dimensionalen Gitter Λ gearbeitet. An jedem Gitterpunkt $\vec{n}$ sitzt ein Spin S, wobei $S_{\vec{n}} \in \{-1, +1\}$ und $\vec{n} \in \Lambda$ ist.
Ohne hier auf die Details des Problems einzugehen soll nun gezeigt werden warum die Modellierung dieses Problems trotz winziger Größe Monte-Carlo-behandelt werden muss.
Dazu wollen wir uns folgendes vergegenwärtigen:
Sei unser Magnet ein mikroskopischer Würfel $L \times L \times L$. Die Anzahl möglicher Anordnungen von Spins auf diesem Gitterwürfel gibt bei V Gitterpunkten $\Rightarrow 2^V$ Möglichkeiten:

$$\begin{gathered} L = 100; \Rightarrow V = 100 \cdot 100 \cdot 100 = 10^6 = 1000000 \\ 2^V = 2^{10^6} = 2^{1000000} = 2^{10 \cdot 100000} = (2^{10})^{100000} = (1024)^{100000} \\ \simeq (10^3)^{100000} = 10^{300000} \end{gathered} \tag{1}$$

Sofort wird nun klar warum hier mit Monte-Carlo gearbeitet werden muss. Ähnlich verhält es sich mit unserem Travelling Salesman Problem.

2.1.2 Beispiel: Travelling Salesman Problem

Beim Travelling Salesman Problem wächst die Anzahl der Möglichkeiten mit steigender Städteanzahl fakultativ. Das bedeutet, dass es für eine Rundreise mit vorgegebener Start-/Ziel-Stadt bei N zu besuchenden Städten $(N-1)!$ Möglichkeiten gibt.
Nehmen wir also an wir haben insgesamt 15 Städte:

$$N = 15 \Rightarrow (N-1)! = (14)! = 14! = 87178291200 \tag{2}$$

50! ist bereits

$$50! = 3.04140932\ldots \times 10^{64} \tag{3}$$

und 70!

$$70! = 1.19785717\ldots \times 10^{100} \tag{4}$$

Dies bedeutet dass hier durch Ausprobieren von Wegen keine Chance besteht zu einer vernünftigen Lösung zu gelangen. Also wendet man sich wieder der Monte-Carlo-Methode zu.

2.2 Arbeitsweise der Monte-Carlo-Simulation

2.2.1 Markov-Ketten

Die Menge aller Konfigurationen S wird nun durch eine sehr viel kleinere Menge an Konfigurationen $S^{(t)}$ ersetzt. Wir bewegen uns nun schrittweise durch den Konfigurationsraum. Durch einen Zufallsprozess wird aus der aktuellen Konfiguration eine Angebotskonfiguration $S^{(t+1)}$ erzeugt. Für den übergang zu der neuen Konfiguration gibt es eine bestimmte Wahrscheinlichkeit:

$$W(S^{(t+1)} = S' | S^{(t)} = S) = W(S \to S') \tag{5}$$

Natürlich muss es sich dabei um eine Wahrscheinlichkeit handeln, dh. es muss gelten:

$$1 \geq W(S \to S') \geq 0 \tag{6}$$

$$\sum_{S'} W(S \to S') = 1 \tag{7}$$

2.2.2 Formalisierung des Travelling Salesman Problem

Um überhaupt dem Travelling Salesman Problem mit Monte-Carlo-Methoden der statistischen Physik zu begegnen muss dieses dem Lösungsapparat formal zugängig gemacht werden. Dazu definieren wir ein kanonisches Ensemble, also ein System in einem Wärmebad mit konstanter Temperatur. Nun mag ein Einwand was denn ein Handlungsreisender mit einem Wärmebad gemein hat durchaus begründet sein, doch erschließt sich diese Einsicht unmittelbar nach der folgenden Diskussion.
Wir gehen von folgender Wahrscheinlichkeit für eine Konfiguration S aus:

$$P[S] = \frac{1}{Z} \cdot e^{-\frac{1}{k_B \cdot T} \cdot L[S]} \tag{8}$$

$$\begin{aligned} k_B = 1.3806505(24) \cdot 10^{-23} J/K \quad &\ldots \quad \text{Boltzmannkonstante} \\ T \quad &\ldots \quad \text{Temperatur} \\ L[S] \quad &\ldots \quad \text{Kostenfunktion} \end{aligned}$$

Wir haben nun eine Wahrscheinlichkeit eingeführt, wir müssen also überprüfen ob sie auch die Axiome erfüllt:

$$0 \leq P[S] \leq 1 \quad \checkmark \tag{9}$$

$$\sum_S P[S] = \frac{1}{Z} \underbrace{\sum_S e^{-\frac{1}{k_B \cdot T} \cdot L[S]}}_{Z} = \frac{1}{Z} \cdot Z = 1 \quad \checkmark \tag{10}$$

Der Ansatz scheint also nicht Unvernünftig zu sein. Wir definieren ferner die Kostenfunktion $L[S]$ als die Länge einer Konfiguration, also als die Länge der Rundreise der Liste der Konfiguration. Damit haben kürzere Konfigurationen eine höhere Wahrscheinlichkeit, was unserem Anliegen, der Suche nach dem kürzesten Weg, zuträglich ist.

Ohne Beweis nehmen wir an, dass diese Wahrscheinlichkeit die Detailed-Balance-Gleichung erfüllt, dh. dass der Markov-Prozess in die gewünschte Verteilung läuft.
Nun müssen wir noch wissen, wie ein möglicher Algorithmus zur Lösung des Problems aussehen kann. Dies führt zum sogenannten Metropolis-Algorithmus.

2.3 Metropolis-Algorithmus

Wie vorher definiert sei

$$P[S] = \frac{1}{Z} \cdot e^{-\frac{1}{k_B \cdot T} \cdot L[S]} \tag{11}$$

Der Algorithmus arbeitet auf folgende Weise:

1. Erzeuge aus $S^{(t)}$ eine Angebotskonfiguration $\tilde{S}$

2. Berechne $\rho = \frac{P[\tilde{S}]}{P[S^{(t)}]} \Rightarrow \rho \in [0, \infty)$

3. Ziehe eine Zufallszahl $r \in [0, 1]$ gleichverteilt und akzeptiere $\tilde{S}$ als $S^{(t+1)}$ wenn $r < \rho$

4. Gehe zu 1

Wieder ohne Beweis: Der Metropolis-Algorithmus erfüllt die Detailed-Balance-Gleichung.
Dieser Algorithmus wird nun immer wieder abgearbeitet, und dabei wird die Temperatur langsam heruntergefahren, bis der Prozess schließlich an einer Konfiguration hängen bleibt, deren Liste der Liste mit der kürzesten Rundreise sehr nahe ist. Die zu erzeugende Angebotskonfiguration muss 'in der Nähe' der aktuellen Konfiguration sein, dh. die Markov-Schritte durch den Konfigurationsraum dürfen nicht zu groß werden. Auf die Erstellung der Angebotskonfiguration wird im nächsten Kapitel eingegangen. Mit dem Wissen das nun vorliegt sollte es gelingen dieses Problem in ein Programm umzusetzen.

3 Umsetzung des Problems in ein C++ Programm

3.1 Technisches

Dieses Programm wurde in C++ geschrieben. Dabei kam zum Einsatz:

- OS: Debian/GNU
- compiler: g++ version 4.0.3
- Graphik: `xmgrace`
- Editor: `vi`
- Dokumenterstellung: LaTeX

3.2 Aufteilung des Programms, Konzept

Zum Konzept des Programmes sei folgendes gesagt. Um einen gewissen Wirklichkeitsanspruch der Problemstellung, respektive der Lösungen zu wahren wurde das Problem auf ein Quadrat im $\mathbb{R}^2$ beschränkt dessen Kantenlänge so gewählt wurde, dass die eingeschlossene Fläche ungefähr der Fläche Europas ($A_{Europa} \sim 10180000 km^2$) entspricht. Die verwendete Längeneinheit ist also km. Dies dient ausschließlich der Anschauung, da km ein gutes Maß der alltäglichen Erfahrung im Bezug auf Reisen darstellt.

3.2.1 Aufteilung

Das Problem wurde in zwei Programmteile aufgespalten:

1. Erstellung der Städte und Bereitstellung einer Datei mit Städtenummern und ihren Koordinaten →`citymapcreator.cpp`
2. Monte-Carlo-Programm zur Berechnung der Route, Bereitstellung der Start- und Lösungskonfiguration zur weiteren (graphischen) Auswertung→`freezer.cpp`

Beiden Programmen ist eine Klasse `City` gemein, die sie sich durch Einbindung der Datei `city.h` zur Verfügung halten.

Um nun zu einer Lösung zu gelangen sind folgende Schritte vom User erforderlich:

1. Eine Landkarten-Datei `map.dat` ist durch Aufruf des Programmes `citymapcreator.run` zu erstellen
2. Das Programm `freezer.run` ist auszuführen.

Nach Ausführung dieser beiden Schritte liegen zwei Dateien vor: Die Datei `start.dat` enthält die Startkonfiguration, die Datei `solution.dat` die Lösungskonfiguration. Die Dateien können nun von einem Graphikprogramm (z.B. `xmgrace, gnuplot...`) zur weiteren Auswertung ausgelesen werden. Jede der Dateien besteht aus N Zeilen, wobei N die Anzahl der Städte ist, und 3 Spalten. In jeder Zeile befindet sich die Information zu genau einer Stadt, und zwar so, dass in der ersten Spalte die Nummer der Stadt steht, in der zweiten Spalte die x- und in der dritten die y-Koordinate.

Ein Ausschnitt sieht so aus:

```
.       .         .
.       .         .
.       .         .
.       .         .
23   1203.4    756.3
12   865.2     302.0
27   1592.6   2058.7
.       .         .
.       .         .
.       .         .
.       .         .
```

3.2.2 Konzept-Die Klasse `City`

Für eine solche Problemmstellung bietet sich natürlich eine objektorientierte Implementierung an. Dazu wurde die Klasse `City` kreiert, deren Aufbau so aussieht: Ein Objekt der Klasse `City` besitzt drei Werte:

- `n`: Datentyp `int`, ist die fortlaufende Nummer der Stadt, Wertebereich $1, 2, \ldots, N$
- `x`: Datentyp `double`, ist die x-Position der Stadt, Wertebereich $x \in [0, 3200]$
- `y`: Datentyp `double`, ist die y-Position der Stadt, Wertebereich $y \in [0, 3200]$

Darüber hinaus wurde der Operator '-' so überladen, dass er, wenn mit zwei Objekten aufgerufen, den Abstand dieser Objekte zurückgibt. Dies ist eine sehr brauchbare Sache, z.B. wenn man die Länge einer Rundreise berechnen will. Weiters stellt die Klasse noch eine Methode für die Ausgabe am Bildschirm (`print`), sowie die Überladung des Operators '<<' für die Ausgabe in einen Stream bereit.

3.2.3 Konzept-Das Programm `citymapcreator.cpp`

Dieses Programm ist relativ kurz und einfach, es soll hier dennoch kurz behandelt werden. Dem Programm wird ein `N` vorgegeben, (also eine Sädteanzahl), sowie ein `L` und ein Wert `ACCEPT`.
Die Werte müssen vor dem Kompilieren festgelegt werden. `L` steht für die Kantenlänge des Quadrats in km (in unserem Europa-Fall 3200), und `ACCEPT` für den Mindestabstand zwischen den Städten. Dieser Wert wurde für alle Rechnungen auf $50km$ gesetzt. Der Grund für diesen Parameter ist der, dass es einfach unrealistisch ist dass zwei Städte 'in einem Punkt zusammenfallen'. Deswegen wurde, aus Gründen der Anschauung, dieser zusätzliche Parameter eingeführt. Er hat keinen weiteren Einfluss auf die Berechnung.

Die Arbeitsweise des Programms ist durch die Zufallsgeneration von x- und y-Koordinaten für jede Stadt von $n = 1, \ldots, N$ gegeben. Dies wird durch eine einfache `for`-Schleife erreicht. Für jede neu generierte Stadt wird der Abstand zu allen zuvor generierten Städten gemessen und überprüft, ob dieser Abstand größer ist als `ACCEPT`. Ist dies nicht der Fall, so wird der Schleifenzähler einfach um eins zurückgesetzt und die Abstandsschleife mit `break` verlassen, was zur Folge hat dass eine neue Stadt zum aktuellen `n` generiert wird. Erst wenn alle Abstände über dem Wert `ACCEPT` liegen wird die neue Stadt akzeptiert. Dieser Umstand ist Namensgebend für den Parameter `ACCEPT`.

Liegen letztlich `N` Städte vor so werden diese in die bereits zuvor erwähnte Datei `map.dat` ausgegeben. In dieser Datei stehen in den ersten Zeilen auch noch die Werte `L, N` sowie `ACCEPT`, die dann vom anderen Programm übernommen werden.

Der Anfang der Datei `map.dat` sieht so aus:

```
L.............. 3200
N.............. 20
ACCEPT......... 50
0 3022.19 1499.06
1 415.088 2030.83
2 2446.86 2125.87
3 3166.8 3162.93
.   .       .
.   .       .
.   .       .
.   .       .
```

Dies entspricht auch zugleich der Startkonfiguration: Die zufällig generierten Städte weisen am Anfang die Ordnung der fortlaufenden Nummerierungen auf, dh. die fortlaufende Nummer ihrer Aufnahme in die Liste der zulässigen Städte. Jeder Zeile entspricht ein Objekt welches genau die in der Zeile angegebenen Parameter besitzt. Die Umordnung der Zeilen in dieser Liste bedeutet eine Änderung der Konfiguration des Systems.

3.2.4 Konzept-Das Programm `freezer.cpp`

Dieser Programmteil ist der wesentlich kompliziertere, und bedarf einer eingehenderen Betrachtung.
Zunächst werden aus der Datei `map.dat` die Werte für `L,N` und `ACCEPT` eingelesen und auch gleich wieder auf den Bildschirm ausgegeben, so dass sich der User überzeugen kann welche Parameter der aktuelle Run besitzt, bzw. ob diese korrekt Eingelesen wurden. Für den restlichen Teil des Programms will ich noch kurz ein paar Begriffe und Vorgehensweisen diskutieren:

Die Kühlfunktion: Wie schon in Abschnitt 2.3 erwähnt muss bei immerwiederkehrender Berechnung der Metropolis-Schritte die Temperatur langsam abgesenkt werden. Wie in der Stahlindustrie beim Tempern kann man dies auf unterschiedliche Art und Weise tun. Diese Art und Weise findet sich in einer Funktion wieder, die ich Kühlfunktion genannt habe. Im einfachsten Falle ist die Funktion eine Geradengleichung der Form

$$y = d - k \cdot x, \quad k > 0 \tag{12}$$

dh. eine abfallende Gerade. Besser wäre es aber die Temperatur 'pendelnd' abzusenken, dh. sie immer wieder ein klein wenig anheben und anschließend wieder absenken. Mit einer solchen Funktion wurde ebenfalls experimentiert. Diese ist vom Typ einer Geradengleichung plus einer Sinus-Funktion:

$$y = d - k \cdot x + \sin a \cdot x, \quad k, a > 0 \tag{13}$$

Der Einfachheit halber wurde im weiteren aber ausschließlich die Gerade mit negativer Steigung verwendet.
Die relevanten Parameter sind also die Steigung, bzw. der Startwert d. Die Funktion wurde in eine C++-Funktion namens `double fx(double x)` geschrieben. Sie gibt an, dass die Temperatur eine Funktion der Zeit x ist. Wird sie mit einem bestimmten Zeitpunkt aufgerufen, so gibt sie die entsprechende Temperatur zurück.
WICHTIG: Die 'Temperatur' in diesem Programm ist ein Parameter in den die Boltzmannkonstante und die aktuelle eigentliche Temperatur eingehen, sie darf also nicht mit der Temperatur im eigentlichen Sinne verwechselt werden.

Das Erstellen einer Angebotskonfiguration: Wir haben dies zwar schon angesprochen, jedoch noch nicht ausführlich genug diskutiert. Die Frage die sich einem stellen sollte lautet: Wie erstelle ich eine Angebotskonfiguration die nahe an der aktuellen Konfiguration liegt?
Dabei wird wie folgt vorgegangen: (Die Vorgehensweise habe ich dem Vorschlag aus der Vorlesung von Professor Gattring übernommen)

1. Vertausche zwei benachbarte Einträge in der Liste:
   $\ldots, 3, 25, 12, 76, 33, \ldots \rightarrow \ldots, 3, 25, 76, 12, 33, \ldots$

2. Schneide Subliste aus und drehe diese um:
   $\ldots, 16, 3, 25, 76, 12, 33, 11, \ldots \rightarrow \ldots, 16, 33, 12, 76, 25, 3, 15, \ldots$

3. Schneide Subliste aus, füge sie woanders ein:
   $\ldots, 16, 33, 12, 76, 25, 3, 15 \ldots \rightarrow \ldots, 3, 15, 12, 76, 25, 16, 33, \ldots$

Während des Schreibens des Programms `freezer.cpp` wurde ein Programmfragment `offer.cpp` ausgegliedert, welches aus einer Startkonfiguration eine Angebostkonfiguration erstellt und die Listen vor und nach jedem Schritt am Bildschirm ausgibt. Das Programm wurde zur Kontrolle der Arbeitsweise des Angebots-Algorithmus verwendet und eignet sich nun sehr gut zur Anschauung:

Hier der Output von `offer.run`, dem Kompilat von `offer.cpp`

```
reading parameters...
L.....................3200
N.....................20
ACCEPT................50
Tstart................300
0 3022.19 1499.06
1 415.088 2030.83
2 2446.86 2125.87
3 3166.8 3162.93
4 1716.63 1177.71
5 2123.54 578.963 //Liste geladen
6 641.571 2966.93
7 1662.04 734.516
8 2553.53 2362.46
9 694.316 483.041
10 1911.45 1983.29
11 1987.57 2073.61
12 693.166 2147.82
13 2375.53 2718.09
14 2756.76 770.775
15 2269.84 2123.69
16 954.525 2578.94
17 1504.81 2684.93
18 2647.85 201.388
19 1379.1 1471.61
Staedte importiert...
```

```
Schritt 1:
0 3022.19 1499.06
1 415.088 2030.83
2 2446.86 2125.87
3 3166.8 3162.93
4 1716.63 1177.71
5 2123.54 578.963
6 641.571 2966.93
7 1662.04 734.516
8 2553.53 2362.46
9 694.316 483.041
10 1911.45 1983.29
11 1987.57 2073.61
12 693.166 2147.82
13 2375.53 2718.09
14 2756.76 770.775
15 2269.84 2123.69
17 1504.81 2684.93 //16 und 17 wurden vertauscht
16 954.525 2578.94
18 2647.85 201.388
19 1379.1 1471.61
```

```
Schritt 2:
0 3022.19 1499.06
1 415.088 2030.83
2 2446.86 2125.87
3 3166.8 3162.93
4 1716.63 1177.71
5 2123.54 578.963
6 641.571 2966.93
7 1662.04 734.516
8 2553.53 2362.46
9 694.316 483.041
10 1911.45 1983.29
15 2269.84 2123.69
14 2756.76 770.775
13 2375.53 2718.09 //11 bis 15 ausgeschnitten und umgedreht
12 693.166 2147.82
11 1987.57 2073.61
17 1504.81 2684.93
16 954.525 2578.94
18 2647.85 201.388
19 1379.1 1471.61
```

```
Schritt 3:
0 3022.19 1499.06
15 2269.84 2123.69
14 2756.76 770.775
13 2375.53 2718.09
12 693.166 2147.82
11 1987.57 2073.61
6 641.571 2966.93
7 1662.04 734.516
8 2553.53 2362.46
9 694.316 483.041 //Elemente {1-5} mit Elementen {11-15} vertauscht
10 1911.45 1983.29
1 415.088 2030.83
2 2446.86 2125.87
3 3166.8 3162.93
4 1716.63 1177.71
5 2123.54 578.963
17 1504.81 2684.93
16 954.525 2578.94
18 2647.85 201.388
19 1379.1 1471.61
Offer ist 33268.2 km lang und aktuelle Liste ist 33779 km lang.
```

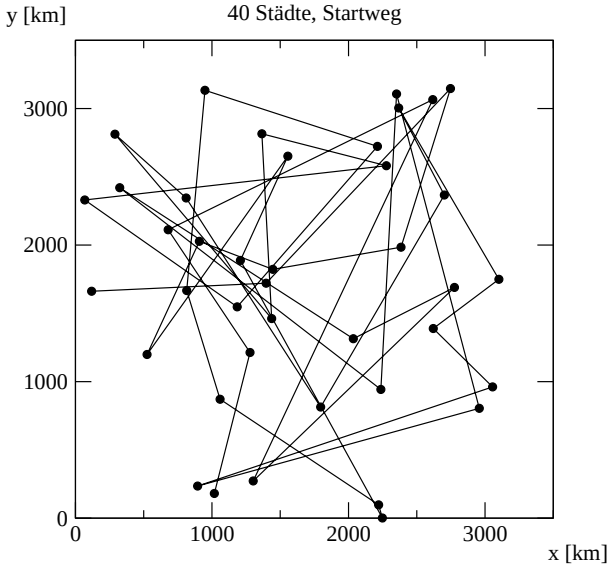
Die letzte Ausgabe zeigt an, dass die Angebotsliste (also die nach Schritt 3) kürzer ist als die aktuelle Liste. Zu beachten ist, dass jede gültige Liste immer die Stadt mit Nummer $n = 0$ an erster Stelle haben muss (Ausgangs-/Zielpunkt), bzw. dass jede Stadt nur genau einmal vorkommt. Um diese Manipulationen an der Liste in den Griff zu bekommen, muss man geschickt auf Intervallüberlappungen und dergleichen Rücksicht nehmen. Es soll hier nicht näher darauf eingegangen, der Vollständigkeit halber jedoch darauf hingewiesen werden. Im Detail kann man die Vorgehensweise am ausführlich kommentierten Sourcecode nachvollziehen.

Nachdem jetzt diese beiden Punkte besprochen wurden wenden wir uns wieder dem Programm zu:
Nach dem Einlesen der `map.dat` generiert `freezer.cpp` ein Array mit einer exakten Kopie der Städte wie sie in der Datei vorgefunden wurden. Nach einigen kurzen Berechnungen zur Kühlfunktion (im wesentlichen die Nullstelle der Kühlfunktion die für den Modulo-Schleifenzähler der Statusanzeige benötigt wird) wird die erste Angebotskonfiguration nach obigen Schema erstellt. Es kommt dann der Metropolis-Algorithmus zum tragen, der entscheidet, ob eine neue Konfiguration akzeptiert wird oder nicht. Um diese Entscheidung treffen zu können benötigt er Kenntnis der Länge der Angebots-, sowie der Momentankonfiguration. Dazu gibt es die Funktion `double length(vector<City> z)`. Diese Funktion summiert einfach alle Abstände aufeinanderfolgender Städte (plus den Abstand der Rückreise zum Ausgangspunkt) und gibt diese als `double` zurück.

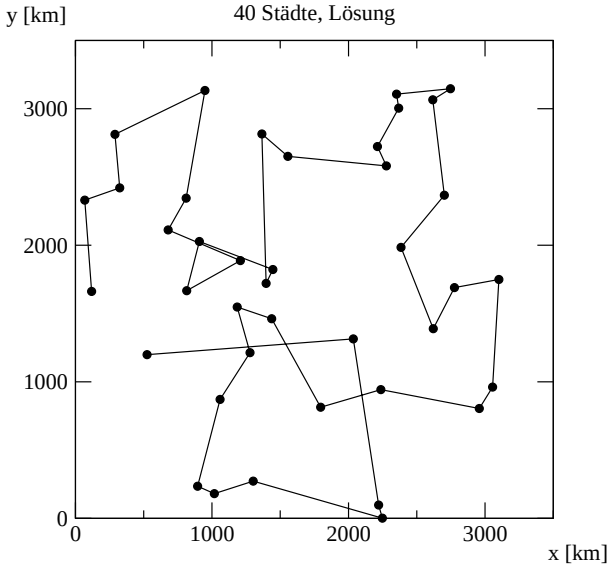
Die zweite relevante Funktion für die Berechnung von ρ ist `double probability(vector<City> z, double len)`. Sie berechnet ρ wie im Metropolisalgorithmus. An dieser Stelle sei noch erwähnt, dass das Einstellen der Kühlfunktionsparameter erheblichen Einfluß auf die Berechenbarkeit der Wahrscheinlichkeit hat.

4 Auswertung

Eine Startkonfiguration sieht i.A. so aus:

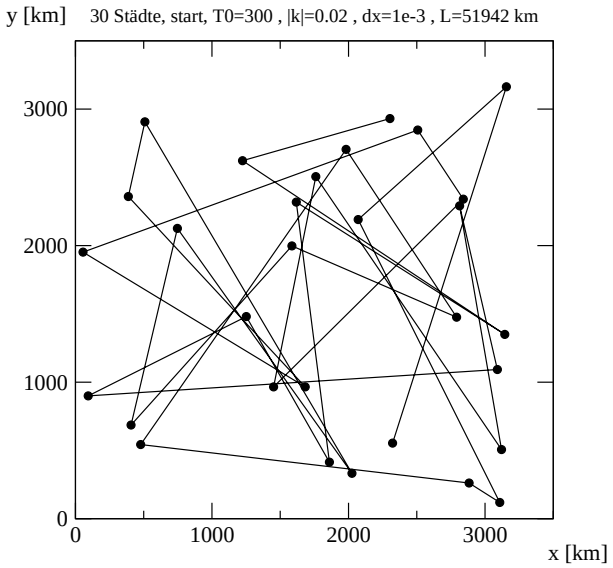


Nach der Abkühlung:

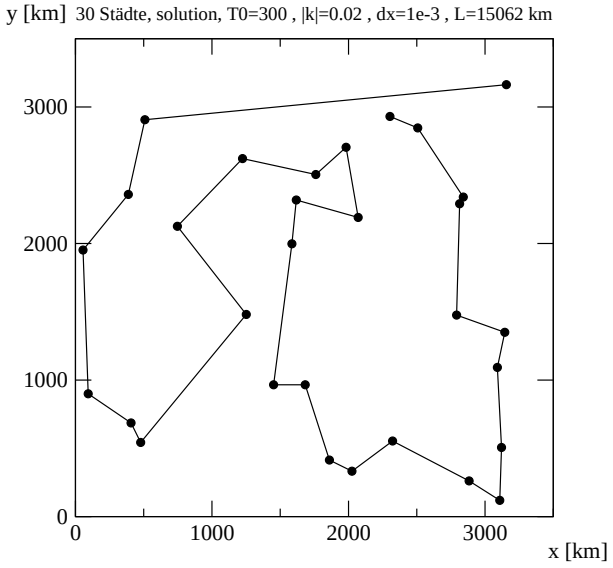


Wir finden also eine deutliche Entwirrung, in diesem Fall noch nicht ganz optimal, vermutlich wegen schlecht gewählter Parameter der Kühlfunktion.
Ein Beispiel für eine gute Wahl der Parameter:

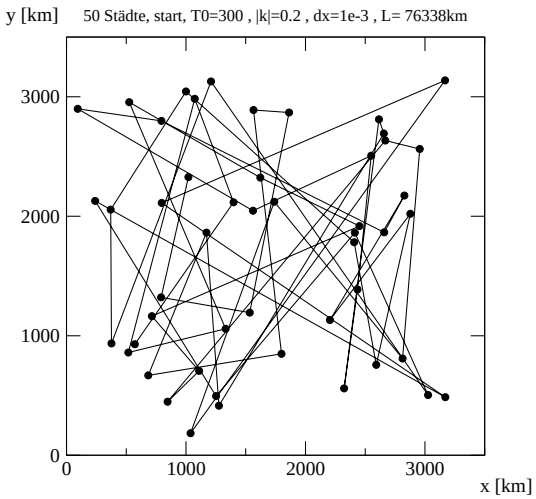
Startkonfiguration:

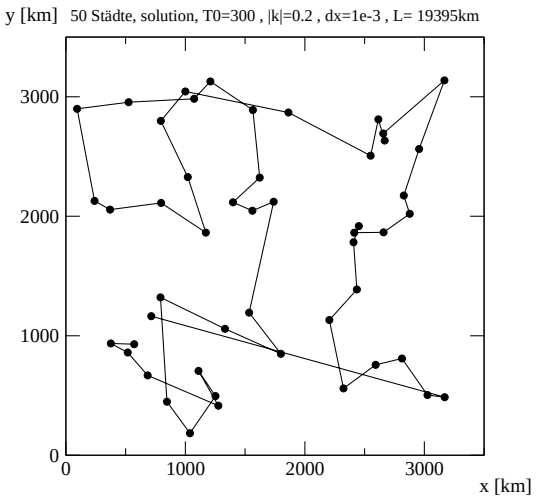


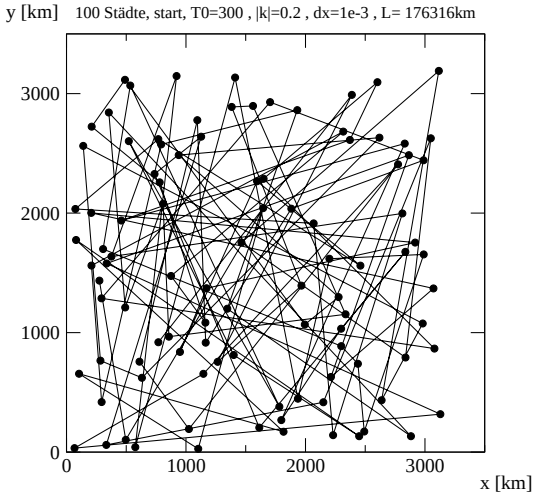
Nach der Abkühlung:

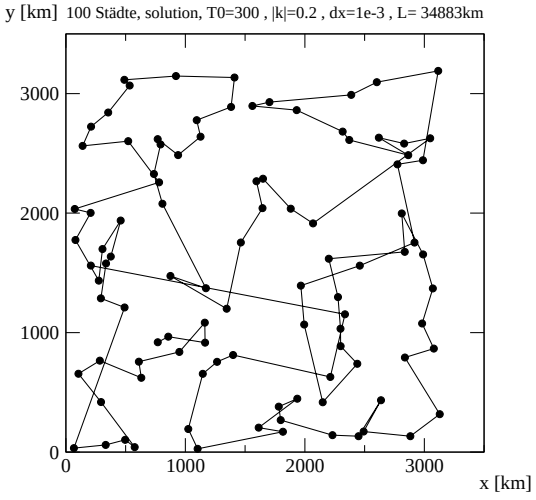


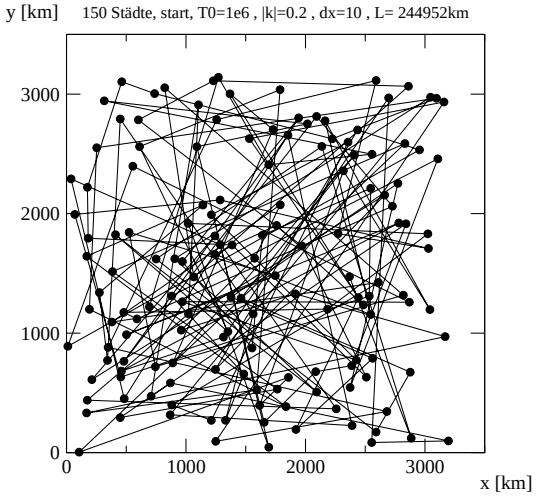
Weiter Runs mit etwas größeren Ns:

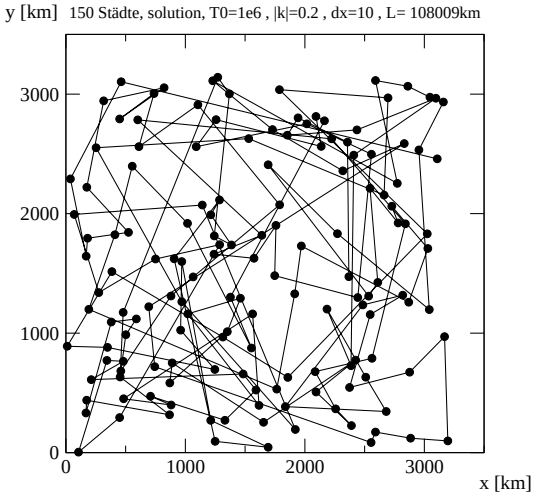












5 Appendix A: city.h

```
//city.h

#ifndef CITY_H
#define CITY_H

#include <iostream.h>
#include <math.h>

using namespace std;

class City{

        public:
                //Konstruktoren
                City() : n(0), x(0), y(0){}
                City( const City& c) : n(c.n), x(c.x), y(c.y) {}
                City( int number, double xpos, double ypos) : n(number), x(xpos), y(ypos) {}

                //Dekonstruktor
                ~City() {}

                //Operator fuer Abstandsmessung
                double operator- (const City& c) const;

                //generiere City-Koordinaten
                void create(int number, double xpos, double ypos);

                //Ausgabe
                void print() const
                {
                        cout << n << " " << x << " " << y;
                }

                friend ostream& operator<< (ostream&, const City&);
        private:
                int n;
                double x;
                double y;

        };

//Initialisieren der City
void City::create(int number, double xpos, double ypos)
{
        n=number; x=xpos; y=ypos;
}

//Abstandsmessung
double City::operator- (const City& c) const
{
return sqrt((x-c.x)*(x-c.x)+(y-c.y)*(y-c.y));
}

//Ausgabe
ostream& operator<< (ostream& os, const City& c)
{
        return os << c.n << " " << c.x << " " << c.y;
}

#endif
```

6 Appendix B: citymapcreator.cpp

```
#include <iostream.h>
#include <math.h>
#include <fstream.h>
#include <time.h>
#include "city.h"

//GLOBALE KONSTANTEN

int L = 3200;//Kantenlaenge des Quadrates
int N = 20;//Anzahl der Staedte
int ACCEPT = 50; //Mindestabstand zwischen Staedten

int main(void)
{
cout << "\n\n";
cout << "Dieses Programm generiert eine Zufallsanordnung von 500
        Staedten (Punkten) auf einer\n";
cout << "quadratischen Flaeche. Die Kantenlaenge des Quadrats
        L=3200. Dann entspricht\n";
cout << "die Flaeche des Quadrates ungefaehr der Flaeche Europas
        (F(Europa)=10180000 km^2)\n";
cout << "in Quadratkilometern. Dies soll zur Veranschaulichung des
        Problemes beitragen.\n";
cout << "Die Daten sollen in weiterer Folge dazu verwendet werden,
        um das Optimierungsproblem\n";
cout << "der kurzesten Rundreise (Travelling Salesman Problem) daran
        zu erproben.\n";
cout << "Die Generation der Orte erfolgt zufaellig, allerdings mit der
        Einschraenkung, dass\n";
cout << "die Entfernung jeder neu generierten Stadt zu jeder bereits
        akzeptierten Stadt\n";
cout << "mindestens "<< ACCEPT <<" km betraegt. Dies soll den Realismus
        des Modells wahren und dient \n";
cout << "wieder nur der Anschauung. Die Entfernungen sind Luftlinie und
        beruecksichtigen keine \n";
cout << "Erdkruemmung. Die Daten werden in eine Datei (map.dat)
        geschrieben und so dem zweiten\n";
cout << "Programmteil freezer.cpp, der dann die Optimierung vornimmt,
         bereitgestellt.\n";

srand(time (NULL));

double xpos = 0;
double ypos = 0;
int aux = 0;

City c[N];//Viele Staedte

ofstream fout("map.dat"); //Ausgabedatei
fout << "L..............." << L << "\n";
fout << "N..............." << N << "\n";
fout << "ACCEPT.........." << ACCEPT << "\n";
cout << "\n\n\n";
cout << "L..............." << L << "\n";
cout << "N..............." << N << "\n";
cout << "ACCEPT.........." << ACCEPT << "\n";



for (int i=0; i<N; i++)
{
        c[i].create(i,(float)(L)*rand()/(RAND_MAX +1.),
                        float(L)*rand()/(RAND_MAX+1.));

        for (int j=0; j<i; j++)
```

```
        {
                if( (c[i] - c[j]) < ACCEPT)
                {
                        //cout << "ACHTUNG!!!: " << (c[i] - c[j]) << " ";
                        //c[i].print();
                        //cout << "\n";
                        i--;
                        break;
                }



        //c[i].print();
        //cout << " mit Abstand zu c" << j <<" "  << (c[i] - c[j]) << "\n";
        }


}

for (int q=0; q<N; q++)
{
        //c[q].print();
        //cout << "\n";
        fout << c[q] << "\n";
}

cout << "\n\n\n";
cout << "Es wurden " << N << " Staedte mit Abstand " << ACCEPT
                        << " km generiert...\n\n\n";

return 0;
}
```

7 Appendix C: freezer.cpp

```
#include <iostream.h>
#include <math.h>
#include <time.h>
#include <fstream.h>
#include <vector.h>
#include "city.h"
using namespace std;

//Globale Variable
int L;                          //L"aenge des quadrats in km
double K=0.02;                  //Betrag der Steigung der Kuehlfunktion
double A=25;                    //Amplitude des Sinus in der Kuehlfunktion
double I=100;                   //Periodizitaetsintervall d. Sinus d. KFkt
int N=10;                       //Anzahl der Staedte
int ACCEPT;                     //Mindestabstand der Staedte
double T0 = 300;                //Starttemperatur
double T = 0;                   //Temperatur
const double dx = 0.001;        //Evolvierungsinkrement fuer Kuehlzeit x
double NS = T0/K;               //Nullstelle der Kuehlgeraden
float CL = NS/(10*dx);          //Counterlimit in float
unsigned long limit = (unsigned long)(CL);//auf Integer gerundetes Counterlimit

//Funktionsprototypen
double fx(double x);            //Kuehlfunktion
double length(vector<City> z);  //Berechnet die Laenge einer Rundreise
double probability(double len, double T); //Berechnet die
                                          //Wahrscheinlichkeit einer Liste


int main(void)
{

srand(time(NULL));              //Initialisieren des RANDGEN

double xpos=0;                  //x-Position einer Stadt
double ypos=0;                  //y-Position einer Stadt
double rho=0;                   //Wahrscheinlichkeitsverhaeltnis
double r=0;                     //Akzeptanzwahrscheinlichkeit
string dummy;                   //Dummystring zum Einlesen
string verbose[10];             //Ausgabestring waehrend Kuehlphase
double len = 0;                 //Laenge einer Rundreise
double aux=0;                   //Hilffloat
double x=0;                     //Zeit
int counter=0;                  //Zaehler
int h,h1=0;                     //Hilfsinteger
vector<City> d;                 //Hilfsvektor
vector<City> e;                 //Hilfsvektor

cout << "\n\n\n";
cout << "reading parameters:\n\n";
ifstream infile ("map.dat");            //Einlesen der von citymapcreator.run
ofstream fout ("solution.dat");         //Vorbereiten der Loesungsdatei
ofstream gout ("start.dat");            //Vorbereiten der Startkonfigdatei
infile >> dummy >> L;                   //generierten Daten und ausschreiben
infile >> dummy >> N;                   //derselben zur Kontrolle
infile >> dummy >> ACCEPT;

cout << "L....................." << L << "\n";
cout << "N....................." << N << "\n";
cout << "ACCEPT..............." << ACCEPT << "\n";
cout << "Tstart..............." << T << "\n";
cout << "\n\n";
cout << "Kuehlfunktionsparameter:\n\n";
cout << "Steigung d. Geraden....." << -1*K << "\n";
cout << "T0....................." << T0 << "\n";
```

```
cout << "Nullstelle ..... " << NS << "\n";
cout << "dx ..... " << dx << "\n";
cout << "Counterlimit ..... " << limit << "\n\n\n";

City c[N];              //erstelle Staedte
City offer[N];          //Kopie der Konfiguration fuer Angebotskonfiguration
vector<City> angebot;   //Nimmt fertiges Angebot auf
vector<City> momentan;  //Nimmt aktuelle Konfiguration auf
vector<City> loesung;   //Loesungsvektor
vector<City> start;     //Startkonfiguration


for (int i=0; i<N; i++)         //reproduziere Landkarte
{
infile >> dummy >> xpos >> ypos;
c[i].create(i,xpos,ypos);
start.push_back(c[i]);          //speichere Startkonfiguration
}
cout << "Reproduziere Landkarte:\n\n";
cout << "Es wurden " << N << " Staedte importiert...\n";
//alle Staedte eingelesen
cout << "\n\n";

verbose[0] = "Es wird kuehl.....(000% done)\n";   //Laden des? Ausgabearrays
verbose[1] = "Es wird kalt.....(010% done)\n";    //Ausgabearrays fuer
verbose[2] = "Es wird frostig.....(020% done)\n"; //die Kuehlphase
verbose[3] = "Es wird eisig.....(030% done)\n";
verbose[4] = "Es wird eiskalt.....(040% done)\n";
verbose[5] = "Es wird bitterkalt.....(050% done)\n";
verbose[6] = "Es wird arktisch.....(060% done)\n";
verbose[7] = "Es wird kosmisch.....(070% done)\n";
verbose[8] = "Es wird Brown'sch.....(080% done)\n";
verbose[9] = "Es wird absolut.....(090% done)\n";



for(x=0; x<(NS)-1; x+=dx)
{
T=fx(x);                                //Temperatur aktualisieren
loesung.clear();                //Loesche alte potentielle Loesung

//Generiere Angebotsliste:
//0.) Kopiere Startliste auf eine Angebotsliste

for(int n=0; n<N; n++)
{
        offer[n] = c[n];
}

//1.) Vertausche zwei Nachbarn: ziehe Zufallszahl und vertausche mit linkem Nachbar.
//    Dabei muss der Ausgangspunkt in der Liste invariant bleiben.

h=rand()%(N-2) + 2;             //Zufallszahl zwischen 2 und N-1
d.push_back(offer[h]);          //herausnehmen der Stadt an Stelle h
d.push_back(offer[h-1]);        //herausnehmen linker Nachbar von h
offer[h]=d[1];                  //Austausch 1
offer[h-1]=d[0];                //Austausch 2
d.clear();                      //Loeschen des Hilfsvekors



//2.) Schneide eine Subliste aus und drehe diese um. Dabei darf wieder kein Ueberlauf
//    ueber den Listenrand vorkommen, da der Ausgangspunkt invariant bleiben muss
```

```
h=rand()%(N-5) + 5; //Zufallszahl zwischen 5 und N-1

for(int l=0; l<5; l++)
{
        d.push_back(offer[h-l]);        //Kopiere 4 Eintraege links von h und h selbst
}

for(int k=0; k<5; k++)
{
        offer[h-k]=d[abs(k-4)];         //Kopiere in verkehrter Reihenfolge
}

d.clear();                              //Loeschen des Hilfsvektors




//3.) Schneide eine Subliste aus und fuege diese an einer anderen Stelle wieder an.
//    Wieder wird die Aktion auf einen Bereich begrenzt der die 0 invariant laesst
e.clear();                                      //Hilfsvektor loeschen

h  =  rand()%(N-5) + 5;                         //Zufallszahl zwischen  5 und N-1
h1 =  rand()%(N-5) + 5;                         //zweite Zufallszahl zw 5 und N-1
                                                //Schraenke Wertebereich fuer h1 ein:


if(((h-5) < 5) && (h1 < (h+5)))                 //Links kein Platz mehr
{
        h1=rand()%(N-h-5) + h + 5;
}


if(((h+5) > (N-1)) && (h1 > (h-5)))             //Rechts kein Platz mehr
{
        h1=rand()%(h-9) + 5;
}


if(((h-5) >=  5) && ((h+5) <= (N-1)))           //Links und rechts genuegend Platz, aber
{                                               //h1 im verbotenen Bereich
        while(((h-4)<=h1) && ((h+4)>= h1))      //KEINE Intervalluebersch neidung!!!!!
        {
                h1 = rand()%(N-5) + 5;          //waehle neue Zufallszahl
        }
}


for(int j=0; j<5; j++)
{
        d.push_back(offer[h-j]);        //Kopiere erstes Intervall
        e.push_back(offer[h1-j]);       //Kopiere zweites Intervall
}

for(int o=0; o-5; o++)
{
        offer[h-o]  = e[o];             //Einsetzten der anderen Liste
        offer[h1-o] = d[o];             //Einsetzten der zweiten Liste
}

d.clear();                              //Loeschen des Hilfsvektors
e.clear();                              //Loeschen des Hilfsvektors

//Manipulationen der Angebotsliste abgeschlossen
```

```
angebot.clear();                        //Vorbereiten fuer Angebotsaufnahme
momentan.clear();                       //Vorbereiten fuer Ist-Aufnahme

for(int q=0; q<N; q++)
{
        angebot.push_back(offer[q]);            //Angebotsaufnahme
        momentan.push_back(c[q]);               //Ist-Aufnahme
        loesung.push_back(c[q]);                //Potentielle Loesung

}

//Berechne nun rho als Verhaeltnis der Wahrscheinlichkeiten der Listen:

rho = (probability(length(angebot),T))/(probability(length(momentan),T));
r = rand() / (RAND_MAX + 1.); //ziehe Zufallszahl zwischen 0 und 1
//cout << "r ist " << r << " und rho ist " << rho << "\n";
if( r <= rho)           //Akzeptiere Angebot
{
loesung.clear();
for(int s=0; s<N; s++)
{
        c[s] = offer[s];
        loesung.push_back(offer[s]);
}
}


if((counter%(limit) == 0)) cout << verbose[counter/(limit)]; //Zaehler der Schleife
counter++;
}


//Ausgeben der Loesungsliste
cout << "Eingefroren!.....(100% done)\n\n\n";
cout << "Die Loesungsliste hat eine Laenge von " << length(loesung) << " km.\n";
cout << "Die Liste der Startkonfiguration war " << length(start) << " km lang.\n";
cout << "Dies bedeutet eine Verkuerzung der Route um "
     << length(start)-length(loesung) << " km.\n\n\n";
cout << "Die Reihenfolge der Staedte wurde zur weiteren Verwendung in die Datei\n";
cout << "'solution.dat' geschrieben.\n\n";
cout << "Ich wuensche eine angenehme Reise.\n\n\n";

for(int g=0; g<loesung.size(); g++)
{
        fout<<loesung[g]<<"\n";
        gout<<start[g]<<"\n";
}


return 0;
}









//Funktionen


double fx(double x)
{
        //return (T0 - K*x+A*sin(M_PI*x/I)); //Kuehlfunktion, 0 bei x=1500
        return(T0-K*x);                         //Kuehlfunktion
```

```
}



double length(vector<City> z)
{
double l=0;      //setze Laenge auf 0
double aux =0;  //setze Hilfsfloat auf 0

for(int i=0; i<z.size()-2; i++)          //Berechnung der Laenge einer Rundreise
{
        aux = z[i]-z[i+1];
        l = aux + l;
}

aux = z[0]-z[z.size()-1];                //Rueckreise zum Ausgangspunkt
l = l + aux;
z.clear();
return l;
}



double probability(double len, double T)
{
        return(exp(-1/(T)*len));         //Wahrscheinlichkeit einer best. Liste
}
```