Foundations of Computing
vorlesungsbegleitenden Prüfungsaufgabe

Marco Möller
Marco.Moeller@macrolab.de

07.03.2006

Inhaltsverzeichnis

1 Einleitung **2**

2 Pakete, Klassen, Interfaces **2**

2.1 CommonInterfaces . . . 2
2.1.1 PrototypeSolution . . . 2
2.1.2 PrototypeIterator . . . 2
2.1.3 ConstructNeighborhood . . . 2
2.1.4 CalculateCostsByExplicit . . . 3
2.1.5 CalculateCostsByPrototype . . . 3
2.2 TSP . . . 3
2.2.1 Point . . . 3
2.2.2 Edge . . . 3
2.2.3 TspConstructNeighborhood . . . 4
2.3 TSPpermutation . . . 5
2.3.1 TspExplizitPermutation . . . 5
2.3.2 TspPrototypePermutation . . . 6
2.3.3 TspModificationPermutation . . . 8
2.3.4 TspIteratorPermutation . . . 9
2.3.5 TspCostPermutationExplicit . . . 12
2.3.6 TspCostPermutationPrototype . . . 13
2.4 TSPset . . . 13
2.4.1 TspExplizitSet . . . 13
2.4.2 TspPrototypeSet . . . 14
2.4.3 TspModificationSet . . . 15
2.4.4 TspIteratorSet . . . 15
2.4.5 TspCostSetExplizit . . . 19
2.4.6 TspCostSetPrototype . . . 19
2.5 Bipartition . . . 19
2.5.1 BipartitObject . . . 19
2.5.2 BipartitExplizit . . . 20
2.5.3 BipartitPrototypeCostIterativ . . . 20
2.5.4 BipartitPrototypeCostSaved . . . 21
2.5.5 BipartitModificationInformation . . . 22
2.5.6 BipartitConstructNeighborhood . . . 22
2.5.7 BipartitIterator . . . 23
2.5.8 BipartitCostExplicit . . . 25

2.6 Algorithms . . . 25
2.6.1 AlgoMeta . . . 25
2.6.2 AlgoCallBack . . . 28
2.6.3 ExplicitSolutionPair . . . 28
2.6.4 AlgoLocalSearch . . . 28
2.6.5 AlgoTabuSearch . . . 29
2.6.6 AlgoSimAnealing . . . 30
2.6.7 AlgoEvolutionStrategy . . . 31

3 GUI **33**

1 Einleitung

Hier dokumentiere ich die Bearbeitung der vorlesungsbegleitenden Prüfungsaufgabe im Fach "Foundations of Computing" bei Prof. Dr. Karsten Weihe im Wintersemester 2005/06 an der Technischen Universität Darmstadt.

Obwohl die Aufgabenstellungen sehr komplex sind, wollte ich das Programm durch möglichst wenig Klassen implementieren. Dazu habe ich die Klassen verallgemeinert, wodurch manche Strukturen sehr abstrakt wirken. Das hat jedoch den Vorteil, dass es auch mit vertretbaren Aufwand möglich ist, mein Programm zu erweitern, wie etwa um eine 5-Opt-Nachbarschaft.

2 Pakete, Klassen, Interfaces

Dieser Abschnitt beschreibt meinen Quellcode und die Ideen dahinter. Erläuterungen beziehen sich im Allgemeinen immer auf den darauf folgenden Quelltext.

2.1 CommonInterfaces

Das Paket `CommonInterfaces` enthält die in Aufgabe 1.a geforderten Interfaces.

2.1.1 PrototypeSolution

Das Interface `PrototypeSolution` entspricht dem Aufgabenteil 1.a.1.

```
public interface PrototypeSolution {
        Object OtherSolution();
        Object ModificationInformation();
        Object ThisSolution();
}
```

2.1.2 PrototypeIterator

Das Interface `PrototypeIterator` entspricht dem Aufgabenteil 1.a.2.

```
public interface PrototypeIterator {
        PrototypeSolution nextElement();
        boolean hasMoreElements();
}
```

2.1.3 ConstructNeighborhood

Das Interface `ConstructNeighborhood` entspricht dem Aufgabenteil 1.a.3.

```
public interface ConstructNeighborhood {
        PrototypeIterator constructNeighborhood(Object solution);
}
```

2.1.4 CalculateCostsByExplicit

Das Interface `CalculateCostsByExplicit` gehört zum Aufgabenteil 1.a.4.

```
public interface CalculateCostsByExplicit {
        double calculateCostsByExplicit(Object solution);
}
```

2.1.5 CalculateCostsByPrototype

Das Interface `CalculateCostsByPrototype` gehört zum Aufgabenteil 1.a.4.

```
public interface CalculateCostsByPrototype {
        double calculateCostsByPrototype(PrototypeSolution solutionPrototype);
}
```

2.2 TSP

Das Paket `TSP` enthält Klassen, die sowohl von der Mengen- als auch von der Permutationsimplementierung des TSP benötigt werden.

2.2.1 Point

Eine Ecke des TSP-Graphen wird durch ein Objekt des Typs `Point` repräsentiert. In meiner Implementierung gilt $x, y \in [0, 1]$.

```
public class Point {
        private double mX;
        private double mY;
        public Point(double x, double y){
                mX = x;
                mY = y;
        }
        public double getX() {
                return mX;
        }
        public double getY() {
                return mY;
        }
        public boolean equals(Object obj) {
                if (obj instanceof Point) {
                        Point other = (Point) obj;
                        return (other.mX == mX) && (other.mY == mY);
                }
                return false;
        }
        public int hashCode() {
                return (int) ((mY + mX) * Integer.MAX_VALUE);
        }
}
```

2.2.2 Edge

Die Klasse `Edge` repräsentiert eine Kante des TSP-Graphen.

```
public class Edge {
        private Point mPointA;
        private Point mPointB;
        private double mCost;
        public Edge(){}
```

```
        public Edge(Point A, Point B){
                mPointA = A;
                mPointB = B;
                double x = mPointA.getX() - mPointB.getX();
                double y = mPointA.getY() - mPointB.getY();
```

Um die Kostenberechnung von TSP-Lösungen zu beschleunigen, wird die (euklidische) Länge der Kante direkt beim Erzeugen berechnet.

```
                mCost = Math.sqrt( x*x + y*y );
        }
        public double getCost(){
                return mCost;
        }
        public Point getPointA() {
                return mPointA;
        }
        public Point getPointB() {
                return mPointB;
        }
        public int hashCode() {
                return mPointA.hashCode() + mPointB.hashCode();
        }
```

Ber der Überprüfung von zwei Kanten auf Äquivalenz kommt es nicht auf die Reihenfolge ihrer Ecken an (TSP-Lösungen sind ungerichtete Graphen).

```
        public boolean equals(Object obj) {
                if (obj instanceof Edge) {
                        Edge other = (Edge) obj;
                        return ( other.mPointA.equals(mPointA) &&
                                        other.mPointB.equals(mPointB) ) ||
                                ( other.mPointA.equals(mPointB) &&
                                  other.mPointB.equals(mPointA) );
                }
                return false;
        }
}
```

2.2.3 TspConstructNeighborhood

Um von TSP-Lösungen Nachbarschaftsiteratoren zu erzeugen, lässt sich die Klasse `TspConstructNeighborhood` benutzen.

```
public class TspConstructNeighborhood implements ConstructNeighborhood {
```

Dieser Nachbarschaftskonstruktor benötigt Informationen über den Typ der Nachbarschaft (k-Opt) und über die Anzahl der Knoten im TSP.

```
        int mKOpt, mGraphSize;
        public TspConstructNeighborhood(int kOpt, int graphSize){
                mKOpt = kOpt;
                mGraphSize = graphSize;
        }
        public int getKopt(){
                return mKOpt;
        }
```

Dieser Konstruktor ist universell programmiert. Er erkennt, ob es sich um die Mengen- oder die Permutationsimplementierung des TSP handelt und gibt den passenden Iterator zurück.

```
        public PrototypeIterator constructNeighborhood(Object solution) {
                if (solution instanceof TspPrototypePermutation ||
                    solution instanceof TspExplizitPermutation) {
                        return new TspIteratorPermutation(mKOpt,
                                                          mGraphSize,solution);
                }
                if (solution instanceof TspPrototypeSet ||
                    solution instanceof TspExplizitSet) {
                        return new TspIteratorSet(mKOpt,mGraphSize,solution);
                }
                return null;
        }
}
```

2.3 TSPpermutation

Das Paket `TSPpermutation` enthält alle Klassen, die zur Permutationsimplementierung des TSP gehören.

2.3.1 TspExplizitPermutation

`TspExplizitPermutation` repräsentiert explizite Lösungen des TSP. Hierzu wird `ArrayList<Point>` überladen. Somit sind bereits alle wesentlichen Funktionen zum effizienten Abspeichern der Daten vorhanden.

```
public class TspExplizitPermutation extends ArrayList<Point> {
        private static final long serialVersionUID = 1L;
```

Es gibt die Möglichkeit, neue Klassen mit einer Zufallsrundtour zu füllen. Außerdem lassen sich bestehende Rundtouren durch zufällige ersetzen.

```
        public void newRandomize(){
                newRandomize(size());
        }
        public void newRandomize(int n){
                clear();
                for(int i = 0;i<n;i++){
                        Point pnt = new Point(Math.random(), Math.random());
                        add(pnt);
                }
        }
        public TspExplizitPermutation(int n){
                super(n);
                newRandomize(n);
        }
        public TspExplizitPermutation() {}
```

Eine interessante Funktion in dieser Klasse ist `equals(...)`. Da es beim TSP weder auf den Startpunkt der Rundtour noch auf den Umlaufsinn ankommt, sind bestimmte Permutationen von Ecken als äquivalent anzusehen. Dies wird z.B. für die Tabusuche benötigt, in der neue Lösungen immer mit der Aktuellen verglichen werden müssen.

```
        public boolean equals(Object o) {
                TspExplizitPermutation other;
                if (o instanceof TspPrototypePermutation) {
                        other = ((TspPrototypePermutation) o).ThisSolution();
                } else {
                        if (o instanceof TspExplizitPermutation) {
                                other = (TspExplizitPermutation) o;
                        } else {
                                return false;
                        }
                }
                if (other.size() != size()) return false;
```

Hierzu wird zunächst nach einer gemeinsamen Ecke beider Graphen gesucht.

```
                for (int i = 0; i< size();i++){
                        if (get(i).equals(other.get(0))) {
```

Nun muss geprüft werden, ob der Umlaufsinn der beiden Rundtouren übereinstimmt oder entgegengesetzt ist. Das ist eine Voraussetzung um festzustellen, ob alle weiteren Ecken übereinstimmen.

```
                                int next = i+1;
                                if (next == size()) next = 0;
                                int prev = i-1;
                                if (prev == -1) prev =size()-1;
                                if (get(next).equals(other.get(1))){
                                        for (int j=1;j < size();j++){
                                                i++;
                                                if (i==size()) i=0;
                                                if (!get(i).equals(other.get(j)))
                                                    return false;
                                        }
                                        return true;
                                }
                                if (get(prev).equals(other.get(1))){
                                        for (int j=1;j < size();j++){
                                                i--;
                                                if (i==-1) i=size()-1;
                                                if (!get(i).equals(other.get(j)))
                                                    return false;
                                        }
                                        return true;
                                }
                                return false;
                        }
                }
                return false;
        }
}
```

2.3.2 TspPrototypePermutation

Die Klasse `TspPrototypePermutation` ist die Implementierung des `PrototypeSolution`-Interface für den TSP mit Permutationen.

```
public class TspPrototypePermutation implements PrototypeSolution{
        private Object mOtherSolution;
        private TspModificationPermutation mModificationInformation;
        public Object OtherSolution(){
                return mOtherSolution;
        }
        public TspModificationPermutation ModificationInformation() {
                return mModificationInformation;
        }
```

Hier ist die aufwendigste Funktion die explizite Konstruktion dieser Lösung. Da ich von einer allgemeinen k-Opt Nachbarschaft ausgehe, ist dieser Code *sehr* allgemein!

```
        public TspExplizitPermutation ThisSolution() {
```

Zunächst muss geprüft werden, ob die andere Lösung bereits explizit vorliegt. Ist dies nicht der Fall, wird sie vorweg konstruiert.

```
TspExplizitPermutation otherSolution = null;
if (mOtherSolution instanceof TspExplizitPermutation) {
        otherSolution = (TspExplizitPermutation) mOtherSolution;
}
if (mOtherSolution instanceof TspPrototypePermutation) {
        TspPrototypePermutation tmp = (TspPrototypePermutation)
                                      mOtherSolution;
        otherSolution = tmp.ThisSolution();
}
if (mModificationInformation != null) {
```

Als Erstes wird eine leere explizite TSP-Lösung erzeugt.

```
        int graphSize = otherSolution.size();
        TspExplizitPermutation tour = new
                                      TspExplizitPermutation();
```

Nun werden alle k Bruchstücke, in die die andere Lösung durch die k-Opt-Modifikation zerfällt, in der richtigen Reihenfolge in den neuen TSP eingefügt. Es wird hier davon ausgegangen, dass die Aufbruchpunkte (siehe Abschnitt 2.3.3 auf der nächsten Seite) in aufsteigender Reihenfolge sortiert sind.

```
        int optSize = mModificationInformation.permutationTeile.
                                                size();
        for (int i = 0; i < optSize;i++){
                int bereichNr = mModificationInformation.
                                permutationTeile.get(i);
                int bereichStart = mModificationInformation.
                                   aufbruchpunkte.get(bereichNr);
                int bereichEnd;
                if (bereichNr < optSize-1) {
                        bereichEnd = mModificationInformation.
                                     aufbruchpunkte.
                                     get(bereichNr+1);
                } else {
                        bereichEnd = mModificationInformation.
                                     aufbruchpunkte.get(0);
                }
```

Ein solches Bruchstück muss evtl. gedreht eingefügt werden.

```
                if (mModificationInformation.switchTeile.get(i)) {
                        int n = bereichEnd;
                        while (n != bereichStart) {
                                if (n==0) n = graphSize;
                                n--;
                                tour.add(otherSolution.get(n));
                } else {
                        int n = bereichStart;
                        while (n != bereichEnd) {
                                tour.add(otherSolution.get(n));
                                n++;
                                if (n>= graphSize) n =0;
                        }
                }
        }
```

Um eventuelle Fehler in der Implementierung des Nachbarschaftsiterators abzufangen, wird hier geprüft, ob die neue Lösung genausoviele Einträge wie die andere hat (Rundtouren haben immer gleiche Länge). Falls dies nicht der Fall ist, wird einfach die andere Lösung zurückgegeben.

```
                        if (tour.size()==otherSolution.size()) {
                                return tour;
                        } else {
                                return otherSolution;
                        }
                } else {
                        return otherSolution;
                }
        }
```

Diese Klasse enthält verschiedene nützliche Konstruktoren:

```
        TspPrototypePermutation(){
                mOtherSolution = null;
                mModificationInformation = null;
        }
        TspPrototypePermutation(Object otherSolution,
                                TspModificationPermutation mod){
                mOtherSolution = otherSolution;
                mModificationInformation = mod;
        }
        TspPrototypePermutation makeNeighbour(){
                TspPrototypePermutation Neighbour = new TspPrototypePermutation();
                Neighbour.mOtherSolution = this;
                return Neighbour;
        }
}
```

2.3.3 TspModificationPermutation

Objekte der Klasse `TspModificationPermutation` repräsentieren k-Opt-Änderungsinformationen am permutationsimplementierten TSP.

Die Grundidee hinter dieser Klasse ist, dass der TSP-Graph durch die k-Opt-Nachbarschaft in k Teile zerfällt. Diese müssen anschließend wieder so verbunden werden, dass es eine *geschlossene* Rundtour gibt. Das entspricht genau dem Sachverhalt, dass diese k Bruchstücke beliebig permutiert werden können. Hierzu wird ihre Position im neuen Graph über einen Eintrag im `permutationTeile`-Vektor gespeichert. Zudem kann jedes dieser Teilstücke noch zusätzlich gedreht werden. Dies wird mit `true` bzw. `false` im `switchTeile`-Vektor repräsentiert.

Diese Bruchstücke werden durch k Aufbruchpunkte im Graphen genau gekennzeichnet. So ist das 0-te Bruchstück genau zwischen dem 0-ten und 1-ten Aufbruchpunkt. Die weiteren Bruchstücke folgen. Das k-te Bruchstück liegt somit zwischen dem k-ten Aufbruchpunkt und dem 0-ten. Damit die Nachbarschaft hinterher problemlos konstruiert werden kann, müssen die Aufbruchpunkte im `aufbruchpunkte`-Vektor in aufsteigender Reihenfolge sortiert sein!

Nun werden die Änderungen so beschränkt, dass das erste Bruchstück im neuen Graphen wieder an erster Stelle steht ohne gedreht worden zu sein. Das verhindert die Existenz vieler nicht-trivialer Gleichheiten zwischen Modifikationen, wie z. B. eine Vertauschung und zusätzliche Drehung der Bruchstücke bei 2-Opt. So lässt sich die Größe der Nachbarschaft reduzieren, ohne auf Lösungen verzichten zu müssen.

```
public class TspModificationPermutation{
        Vector<Integer> aufbruchpunkte;
        Vector<Integer> permutationTeile;
        Vector<Boolean> switchTeile;
        TspModificationPermutation(int n){
                aufbruchpunkte = new Vector<Integer>(n);
                permutationTeile = new Vector<Integer>(n);
                switchTeile = new Vector<Boolean>(n);
                for (int i = 0; i<n;i++){
                        aufbruchpunkte.add(i);
                        permutationTeile.add(i);
                        switchTeile.add(false);
                }
```

```
        }
        TspModificationPermutation(){}
```

Die Äquivalenzfunktion berücksichtigt nur triviale Gleichheiten. Dadurch werden nicht alle als gleich anzusehenden Modifikationen auch als äquivalent erkannt. Obwohl viele nicht-triviale Gleichheiten nicht vorkommen (s.o.), gibt es einige Problemfälle. Ein Beispiel dafür ist die Verdrehung eines Bruchstückes der Länge 1, bei der wieder derselbe Graph entsteht.

```
        public boolean equals(Object obj) {
                if (obj instanceof TspModificationPermutation) {
                        TspModificationPermutation tmpMod =
                                        (TspModificationPermutation) obj;
                        return (aufbruchpunkte.equals(tmpMod.aufbruchpunkte) &&
                              permutationTeile.equals(tmpMod.permutationTeile) &&
                              switchTeile.equals(tmpMod.switchTeile));
                }
                return false;
        }
        public int hashCode() {
                return aufbruchpunkte.hashCode() + permutationTeile.hashCode() +
                       switchTeile.hashCode();
        }
        protected TspModificationPermutation clone() {
                TspModificationPermutation tmp = new TspModificationPermutation();
                tmp.aufbruchpunkte = new Vector<Integer>(aufbruchpunkte.size());
                tmp.aufbruchpunkte.addAll(aufbruchpunkte);
                tmp.permutationTeile=new Vector<Integer>(permutationTeile.size());
                tmp.permutationTeile.addAll(permutationTeile);
                tmp.switchTeile = new Vector<Boolean>(switchTeile.size());
                tmp.switchTeile.addAll(switchTeile);
                return tmp;
        }
}
```

2.3.4 TspIteratorPermutation

Die Klasse `TspIteratorPermutation` ist die Iteratorklasse, die es ermöglicht, die Nachbarschaft einer Rundtour zu durchlaufen. Dadurch können entsprechende 2-Opt- und 3-Opt-Modifikationen erzeugt werden.

```
public class TspIteratorPermutation implements PrototypeIterator {
        private int mKOpt, mGraphSize, mMaxNeighbor;
        boolean mHasMore;
```

Dazu merkt man sich eine andere Lösung, zu der jeweils die Änderungen hinzugefügt werden.

```
        private Object mOtherSolution;
```

Damit der Iterator sich nicht wiederholt, werden für 2-Opt die als letztes und als erstes zurückgelieferte Lösung gespeichert. Dadurch ist es möglich, einen vollen Umlauf zu erkennen.

```
        private TspModificationPermutation mLastModification
        private TspModificationPermutation mInitPermutation;
```

Für 3-Opt wird eine Menge von bisher zurückgelieferten Lösungen gespeichert.

```
        private HashSet<TspModificationPermutation> mOldModification;
```

Mit diesem Konstruktor lässt sich ein entsprechender Iterator erzeugen.

```
public TspIteratorPermutation(int kOpt, int GraphSize,
                              Object otherSolution){
        mOtherSolution = otherSolution;
        mGraphSize = GraphSize;
        mKOpt = kOpt;
```

Für 3-Opt lässt sich exakt berechnen, wieviele Nachbarn zu erwarten sind. Da diese wie gefordert zufällig erzeugt werden, muss anschließend geprüft werden, ob sie bereits zurückgeliefert wurden. Um festzustellen, ob noch weitere Nachbarn existieren, vergleicht man mit der maximalen Anzahl `mMaxNeighbor`.

Bei einem Graph mit n Knoten lassen sich $\binom{n}{3}$ verschiedene Bruchstücke unterscheiden. Es gibt 2^3 Möglichkeiten, diese wieder zu verbinden. Dies ergibt insgesamt

$$\begin{aligned} m_p &= \binom{n}{3} \cdot 3! \cdot 2^3 \\ &= \frac{4}{3} \cdot n \cdot (n-1) \cdot (n-2) \\ m_p &\in \mathcal{O}\left(n^3\right) \end{aligned}$$

unterschiedliche Nachbarn, wobei nicht-triviale Äquivalenzen ignoriert werden (siehe Abschnitt 2.3.3 auf der vorherigen Seite). Um alle Möglichkeiten durch Zufallsexperimente zu erzeugen, benötigte man

$$\begin{aligned} s &= \sum_{k=1}^{m_p} \frac{m_p}{m-k+1} \\ &= m_p \sum_{k=1}^{m_p} \frac{1}{m_p-k+1} \\ &= m_p \sum_{k=0}^{m_p-1} \frac{1}{m_p-k} \\ &= m_p \sum_{k=1}^{m_p} \frac{1}{k} \\ s &< m_p \cdot \log_2\left(m_p+1\right) \\ s &> m_p \cdot \left(\frac{1}{2}\log_2\left(m_p\right)+1\right) \\ s &\in \mathcal{O}\left(n^3 \log\left(n\right)\right) \end{aligned}$$

Versuche, was eine sehr große Anzahl (für $n = 50$ wäre $m = 156800$ und $s \approx 1,97 \cdot 10^6$) ist. Gerade Algorithmen, die (wie bspw. die Tabu-Suche) die komplette Nachbarschaft durchsuchen, brauchen dadurch zur Termination sehr lange.

```
        if (mKOpt == 3){
                mMaxNeighbor = (4 * mGraphSize * (mGraphSize-1) *
                                (mGraphSize-2) ) / 3 ;
                mOldModification = new
                                HashSet<TspModificationPermutation>();
        }
        mHasMore = true;
}
public boolean hasMoreElements() {
        return mHasMore;
}
```

Die eigentliche Kernaufgabe dieser Klasse liegt in der folgenden Funktion:

```
public PrototypeSolution nextElement() {
        mHasMore = true;
```

Da die Aufgabenstellung für 2 und 3-Opt sehr unterschiedlich ist, wird differenziert:

```
if (mKOpt==2){
        if (mLastModification == null){
                mLastModification = new
                                TspModificationPermutation(2);
```

Die erste Lösung ist auch bei 2-Opt ein zufälliger Nachbar. Um sicherzustellen, dass die Aufbruchpunkte sortiert sind und nicht übereinander liegen, muss etwas Aufwand betrieben werden:

```
                int k = (int) (Math.random()*(mGraphSize-2));
                int j = (int) (Math.random()*(mGraphSize-k-2));
                j += k+2;
                mLastModification.aufbruchpunkte.set(0,k);
                mLastModification.aufbruchpunkte.set(1,j);
```

Bei einer Änderung genügt es, beide Bruchstücke in ihrer Reihenfolge zu belassen, und lediglich eins zu drehen.

```
                mLastModification.permutationTeile.set(0,0);
                mLastModification.permutationTeile.set(1,1);
                mLastModification.switchTeile.set(0,false);
                mLastModification.switchTeile.set(1,true);
                mInitPermutation = mLastModification.clone();
                return new TspPrototypePermutation(mOtherSolution,
                                        mLastModification.clone());
        } else {
```

Um die zurückgelieferten Nachbarn lexikographisch zu ordnen, habe ich eine entsprechende Nachfolgerfunktion auf den Aufbruchpunkten definiert. Die beiden Aufbruchpunkte sind ein Paar $(i,j) \in [0,\ldots,n-1]^2$ mit $i<j$. Die Nachfolgerabbildung ist

$$(i,j) \mapsto \begin{cases} (i,j+1) & \text{falls } j+1<n \\ (i+1,i+2) & \text{falls } j+1=n \\ (0,1) & \text{falls } i+2=n \end{cases}$$

wobei n die Anzahl der Ecken im Graphen ist. Somit existieren bei 2-Opt $m \in \mathcal{O}\left(n^2\right)$ Nachbarn.

```
                mLastModification.aufbruchpunkte.set(1,
                                mLastModification.
                                aufbruchpunkte.get(1)+1);
                if (mLastModification.aufbruchpunkte.get(1)
                                        == mGraphSize){
                        mLastModification.aufbruchpunkte.set(0,
                                        mLastModification.
                                        aufbruchpunkte.get(0)+1);
                        mLastModification.aufbruchpunkte.set(1,
                                        mLastModification.
                                        aufbruchpunkte.get(0)+1);
                        if (mLastModification.aufbruchpunkte.
                                          get(1) == mGraphSize) {
                                mLastModification.aufbruchpunkte.
                                                set(0,0);
                                mLastModification.aufbruchpunkte.
                                                set(1,2);
                        }
                }
```

Entspricht die aktuelle Änderung der ersten, wird gemeldet, dass es keine weiteren Nachbarn gibt.

```
                if (mLastModification.equals(mInitPermutation))
                    mHasMore = false;
                return new TspPrototypePermutation(mOtherSolution,
                                        mLastModification.clone());
        }
}
```

Die Implementierung von 3-Opt hat folgende Form:

```
                if (mKOpt==3){
                        TspModificationPermutation modNew;
```

Es werden solange neue Änderungen erzeugt, bis eine zuvor noch nicht vorgekommene gefunden wird. Diese Änderung wird zurückgeliefert.

```
                        do {
                                modNew = new TspModificationPermutation(3);
                                int lastAufb = -1;
```

Da auch hier die Aufbruchpunkte sortiert sein müssen, ist dieser Code zum Erzeugen einer zufälligen Permutation etwas aufwändiger. Auch hier kann man auf eine zufällige Position und Drehung des ersten Bruchstückes verzichten.

```
                                for (int j=0;j<3;j++){
                                        int newaufbP = (int) (Math.random()*
                                            (mGraphSize - lastAufb - 1 - 2 + j));
                                        lastAufb = newaufbP + lastAufb + 1;
                                        modNew.aufbruchpunkte.set(j,lastAufb);
                                }
                                modNew.permutationTeile.set(0,0);
                                if (Math.random()<0.5) {
                                        modNew.permutationTeile.set(1,1);
                                        modNew.permutationTeile.set(2,2);
                                } else {
                                        modNew.permutationTeile.set(1,2);
                                        modNew.permutationTeile.set(2,1);
                                }
                                modNew.switchTeile.set(0,false);
                                modNew.switchTeile.set(1,Math.random()<0.5);
                                modNew.switchTeile.set(2,Math.random()<0.5);
                        } while (mOldModification.contains(modNew));
                        mLastModification = modNew;
                        mOldModification.add(mLastModification);
```

Wenn keine weiteren Nachbarn mehr gefunden werden können, wird das signalisiert.

```
                        if (mOldModification.size() >= mMaxNeighbor) {
                                mOldModification.clear();
                                mOldModification.add(mLastModification);
                                mHasMore = false;
                        }
                        return new TspPrototypePermutation(mOtherSolution,
                                                           mLastModification);
                }
                return null;
        }
}
```

2.3.5 TspCostPermutationExplicit

Die Klasse `TspCostPermutationExplicit` implementiert eine Funktion zur Berechnung der Kosten von expliziten in Permutationsrepräsentation vorliegenden TSP-Lösungen.

```
public class TspCostPermutationExplicit implements CalculateCostsByExplicit {
        public double calculateCostsByExplicit(Object solution) {
                if (solution instanceof TspExplizitPermutation) {
                        TspExplizitPermutation tmp =
                                   (TspExplizitPermutation) solution;
```

Hierzu werden die euklidischen Längen aller Kanten im TSP-Graphen aufsummiert:

```
                        double cost=0;
                        for(int i = 0;i<tmp.size()-1;i++){
                                double x = tmp.get(i).getX() -
                                           tmp.get(i+1).getX();
                                double y = tmp.get(i).getY() -
                                           tmp.get(i+1).getY();
                                cost += Math.sqrt(x*x+y*y);
                        }
                        double x = tmp.get(0).getX() -
                                   tmp.get(tmp.size()-1).getX();
                        double y = tmp.get(0).getY() -
                                   tmp.get(tmp.size()-1).getY();
                        cost += Math.sqrt(x*x+y*y);
                        return new Double(cost);
                }
                return 0;
        }
}
```

2.3.6 TspCostPermutationPrototype

Die `TspCostPermutationPrototype`-Klasse implementiert eine Funktion zum Berechnen der Kosten einer TSP-Lösung, die permutationsrepräsentiert als Prototyp vorliegt.

```
public class TspCostPermutationPrototype implements CalculateCostsByPrototype{
        public double calculateCostsByPrototype(PrototypeSolution
                                                solutionPrototype) {
                TspCostPermutationExplicit tmp = new TspCostPermutationExplicit();
```

Hierzu wird die explizite Lösung konstruiert, anschließend werden die zugehörigen Kosten mit der Funktion `TspCostPermutationExplicit` berechnet.

```
                return tmp.calculateCostsByExplicit(
                                        solutionPrototype.ThisSolution());
        }
}
```

2.4 TSPset

Das Paket `TSPset` enthält alle Klassen, die zur Kantenmengen-Implementierung des TSP gehören.

2.4.1 TspExplizitSet

`TspExplizitSet` repräsentiert explizite Lösungen des TSP. Hierzu wird `HashSet<Edge>` überladen. Somit sind bereits alle wesentlichen Funktionen zum effizienten Abspeichern der Daten vorhanden.

```
public class TspExplizitSet extends HashSet<Edge> {
        private static final long serialVersionUID = 1L;
```

Es gibt die Möglichkeit, neue Objekte mit Zufallsrundtouren zu füllen. Außerdem lassen sich bestehende Rundtouren durch zufällige ersetzen.

```
        public TspExplizitSet(){
        public TspExplizitSet(int n){
                super(n);
                newRandomize(n);
        }
        public void newRandomize(int n){
```

```
                this.clear();
                Point newPnt = new Point(Math.random(), Math.random());
                Point oldPnt;
                Point initPnt = newPnt;
                for(int i = 0;i<n-1;i++){
                        oldPnt = newPnt;
                        newPnt = new Point(Math.random(), Math.random());
                        this.add(new Edge(oldPnt, newPnt));
                }
                this.add(new Edge(initPnt,newPnt));
        }
        public void newRandomize(){
                newRandomize(this.size());
        }
}
```

Im Gegensatz zur Permutationsimplementierung wird hier keine spezielle `equals(...)` Funktion benötigt, da die Standardimplementierung im Zusammenspiel mit `equals(...)` von der `Edge` Klasse genau den Anforderungen entspricht.

2.4.2 TspPrototypeSet

Die Klasse `TspPrototypeSet` ist die Implementierung des `PrototypeSolution`-Interface für den TSP mit Kantenmengen.

```
public class TspPrototypeSet implements PrototypeSolution{
        private Object mOtherSolution;
        private TspModificationSet mModificationInformation;
        public Object OtherSolution() {
                return mOtherSolution;
        }
        public TspModificationSet ModificationInformation() {
                return mModificationInformation;
        }
        TspPrototypeSet(Object otherSolution, TspModificationSet mod){
                mOtherSolution = otherSolution;
                mModificationInformation = mod;
        }
```

Bei dieser Repräsentation des TSP lässt sich `ThisSolution()` sehr effizient umsetzen.

```
        public TspExplizitSet ThisSolution() {
                TspExplizitSet otherSolution = null;
                if (mOtherSolution instanceof TspExplizitSet) {
                        otherSolution = (TspExplizitSet) mOtherSolution;
                }
                if (mOtherSolution instanceof TspPrototypeSet) {
                        TspPrototypeSet tmp = (TspPrototypeSet) mOtherSolution;
                        otherSolution = tmp.ThisSolution();
                }
                if (mModificationInformation != null) {
                        TspExplizitSet tour = new TspExplizitSet(otherSolution);
                        tour.removeAll(mModificationInformation.mRemove);
                        tour.addAll(mModificationInformation.mAdd);
                        return tour;
                } else {
                        return otherSolution;
                }
        }
}
```

2.4.3 TspModificationSet

Objekte der Klasse `TspModificationSet` repräsentieren k-Opt-Änderungsinformationen am mengenimplementierten TSP.

```
public class TspModificationSet{
```

Änderungen werden als Mengen von zu entfernenden und hinzuzufügenden Kanten gespeichert, wobei diese Mengen auch leer sein können.

```
        HashSet<Edge> mRemove;
        HashSet<Edge> mAdd;
        TspModificationSet(int n){
                mRemove = new HashSet<Edge>(n);
                mAdd = new HashSet<Edge>(n);
        }
        public boolean equals(Object obj) {
                if (obj instanceof TspModificationSet) {
                        TspModificationSet tmpMod = (TspModificationSet) obj;
                        return (mAdd.equals(tmpMod.mAdd) &&
                                mRemove.equals(tmpMod.mRemove) );
                }
                return false;
        }
```

Um die Kosten von Lösungsprototypen schneller berechnen zu können, lässt sich mit dieser Methode die Kostendifferenz, die diese Modifikation zur Folge hat, erfragen.

```
        public double getCost(){
                double sum=0;
                for (Iterator<Edge> it = mRemove.iterator();it.hasNext();){
                        sum -= it.next().getCost();
                }
                for (Iterator<Edge> it = mAdd.iterator();it.hasNext();){
                        sum += it.next().getCost();
                }
                return sum;
        }
        public int hashCode() {
                return mRemove.hashCode() + mAdd.hashCode();
        }
}
```

2.4.4 TspIteratorSet

Die Klasse `TspIteratorSet` ist die Iteratorklasse, die es ermöglicht, die Nachbarschaft einer Rundtour zu durchlaufen. Es lassen sich entsprechende 2-Opt- und 3-Opt-Modifikationen erzeugen.

```
public class TspIteratorSet implements PrototypeIterator {
        private Object mOtherSolution;
        private TspModificationSet mLastModifcation;
        private int mKOpt;
        private int mGraphSize;
        private int mMaxNeighbor;
        private boolean mHasMore;
        public boolean hasMoreElements() {
                return mHasMore;
        }
```

Bei 2-Opt ist die Grundidee, die Menge der Kanten der anderen Lösung in ein Array von Kanten zu überführen. So ist es möglich, über die Variablen `it1` und `it2` zwei Kanten auszuwählen, die entfernt und modifiziert wieder hinzugefügt werden sollen. Diese Variablen können dann wie bei der Permutationsimplementierung (siehe Abschnitt 2.3.4 auf Seite 11) hochgezählt werden.

```
private int it1, it2;
private int it1init, it2init;
private Edge[] mOtherEdges;
```

Bei 3-Opt wird ebenfalls analog zur Permutationsimplementierung (siehe Abschnitt 2.3.4 auf Seite 9) eine Menge von bereits zurückgelieferten Lösungen gespeichert, um Wiederholungen zu vermeiden.

```
private Set<TspModificationSet> mOldSet;

public TspIteratorSet(int kOpt, int GraphSize, Object otherSolution){
        mOtherSolution = otherSolution;
        mGraphSize = GraphSize;
        mKOpt = kOpt;
        mHasMore = true;
        TspExplizitSet mThisSolution = (TspExplizitSet) mOtherSolution;
        Object[] tmp = mThisSolution.toArray();
        mOtherEdges = new Edge[tmp.length];
        for (int i=0;i<tmp.length;i++){
                mOtherEdges[i] = (Edge) tmp[i];
        }
```

Hier ist die Maximalanzahl der Nachbarn allerdings eine andere als bei der Permutationsimplementierung. Dies ist dadurch bedingt, das die in den Änderungen abgespeicherten Kanten zum einen ungerichtet sind und zum anderen untereinander keine Reihenfolge haben. So genügt es nicht, von $2^3 \cdot \binom{n}{3}$ Nachbarn zu sprechen, da hier auch solche enthalten sind, bei denen die Bruchstücke nur aus einzelnen Knoten bestehen. Falls man genau einen solchen Einzelknoten in den Bruchstücken hat, gibt es nur noch 2^2, bei genau zwei solcher Einzelknoten nur noch 2^1 Möglichkeiten diese Bruchstücke echt verschieden anzuordnen. Somit komme ich auf

$$\begin{aligned}
m_s &= 2^3 \cdot \left[\binom{n}{3} - n \cdot (n-4) - n\right] + \\
&\quad 2^2 \cdot [n \cdot (n-4)] + \\
&\quad 2^1 \cdot [n] \\
&= n \cdot \left[\frac{4}{3}(n-1)(n-2) - 4(n-4) - 6\right] \\
&= m_p - n \cdot [4(n-4) + 6] \\
m_s &< m_p \\
m_s &\in \mathcal{O}\left(n^3\right)
\end{aligned}$$

mögliche Nachbarn. Hierbei ist $\binom{n}{3} - n \cdot (n-4) - n$ die Anzahl der Möglichkeiten, drei Kanten aus n zu entfernen, so dass *kein* Bruchstück die Länge 1 hat. $n \cdot (n-4)$ ist die Anzahl der Möglichkeiten, bei denen genau 1 Bruchstück die Länge 1 hat, und n die Anzahl, bei denen 2 Bruchstücke die Länge 1 haben.

Hierbei sind allerdings immer noch einige identische Rundtouren in der Nachbarschaft zu finden. So wird z.B. bei jeder Auswahl von zu entfernenden Kanten auch wieder die Identität als Modifikation aufgebaut (es werden die gleichen Kanten wieder hinzugefügt, die auch entfernt wurden).

```
        if (mKOpt == 3){
                mMaxNeighbor =mGraphSize*(mGraphSize-1)*(mGraphSize-2)*4/3
                              - mGraphSize * (4*(mGraphSize-4)+6);
                mOldSet = new HashSet<TspModificationSet>(mMaxNeighbor);
        }
        if (mKOpt == 2){
                it1init = it1 = (int) (Math.random() * (mGraphSize - 1));
                it2init = it2 = it1 + 1 + (int)
                                (Math.random() * (mGraphSize - it1 - 1));
        }
}
```

Da in dieser Repräsentation die Kanten ungerichtet sind, ist es unmöglich zu wissen, welche Kanten wieder hinzugefügt werden mussen, um eine geschlossene Rundtour zu erhalten. Deshalb werden zufällig verschiedene

Kombinationen erzeugt und mit `tryModification(...)` auf ihre Gültigkeit überprüft. Diese Funktion durchläuft den Graphen, und prüft, ob sie dabei an weniger als `mGraphSize` Ecken vorbeikommt. Ist das der Fall, so ist die Lösung ungültig. Es lässt sich also feststellen, ob zuwenig Kanten vorhanden sind, oder der Graph in mehrere (etvl. nicht geschlossene) Teilstücke zerfallen ist.

```
private boolean tryModification(TspModificationSet modTry){
        TspExplizitSet newSol = new TspPrototypeSet(mThisSolution,modTry).
                                                        ThisSolution();
        if (newSol.size() < mGraphSize) return false;
        int circleLen=0;
        Edge currentEdge = newSol.iterator().next();
        Point initPoint = currentEdge.getPointA();
        Point nextPoint = currentEdge.getPointB();
        do {
               boolean foundSome = false;
               for(Iterator<Edge> it = newSol.iterator(); it.hasNext();){
                        Edge nextEdge = it.next();
                        if (nextEdge == currentEdge) continue;
                        if (nextEdge.getPointA().equals(nextPoint)){
                                nextPoint = nextEdge.getPointB();
                                currentEdge = nextEdge;
                                circleLen++;
                                foundSome = true;
                                break;
                        }
                        if (nextEdge.getPointB().equals(nextPoint)){
                                nextPoint = nextEdge.getPointA();
                                currentEdge = nextEdge;
                                circleLen++;
                                foundSome = true;
                                break;
                        }
                }
                if (!foundSome) break;
        } while (!nextPoint.equals(initPoint));
        if (!nextPoint.equals(initPoint)) return false;
        return circleLen == mGraphSize - 1;
}
```

Die wesentliche Funktion ist in dieser Klasse `nextElement()`. Sie erzeugt fortlaufend neue Modifikationen.

```
public PrototypeSolution nextElement() {
        mHasMore = true;
        if (mKOpt==2){
                it2++;
                if(it2==mGraphSize){
                        it1++;
                        it2 = it1 + 1 ;
                }
                if(it2==mGraphSize){
                        it1 = 0;
                        it2 = 1 ;
                }
                if (it1 == it1init && it2 == it2init) {
                        mHasMore = false;
                }
                Edge a = mOtherEdges[it1];
                Edge b = mOtherEdges[it2];
```

Es gibt zwei Möglichkeiten, die fehlenden Kanten zu ersetzen. Im Folgenden werden beide ausprobiert und die gültige ausgewählt.

```
                        mLastModifcation  = new TspModificationSet(2);
                        mLastModifcation.mAdd.add(new Edge(a.getPointA(),
                                                    b.getPointA()));
                        mLastModifcation.mAdd.add(new Edge(a.getPointB(),
                                                    b.getPointB()));
                        mLastModifcation.mRemove.add(a);
                        mLastModifcation.mRemove.add(b);
                        if (!tryModification(mLastModifcation)) {
                                mLastModifcation  = new TspModificationSet(2);
                                mLastModifcation.mAdd.add(new Edge(a.getPointA(),
                                                            b.getPointB()));
                                mLastModifcation.mAdd.add(new Edge(a.getPointB(),
                                                            b.getPointA()));
                                mLastModifcation.mRemove.add(a);
                                mLastModifcation.mRemove.add(b);
                        }
                }
                if (mKOpt==3){
```

Es werden solange neue Nachbarn erzeugt, bis ein bisher unbekannter gefunden wird.

```
                        do {
                                if (mLastModifcation!=null) mOldSet.
                                                            add(mLastModifcation);
```

Da nicht alle neuen Kanten zu einem gültigen Graphen führen, werden hier zufällige Kanten erzeugt, bis ein gültiges 3-Tupel gefunden wurde.

```
                                do {
                                        int it1,it2,it3;
                                        do{
                                                it1 = (int) (Math.random() *
                                                             mGraphSize);
                                                it2 = (int) (Math.random() *
                                                             mGraphSize);
                                                it3 = (int) (Math.random() *
                                                             mGraphSize);
                                        } while (it1 == it2 || it2 == it3 ||
                                                 it1 == it3);
                                        ArrayList<Point> pArr = new
                                                         ArrayList<Point>(6);
                                        Edge a = mOtherEdges[it1];
                                        Edge b = mOtherEdges[it2];
                                        Edge c = mOtherEdges[it3];
                                        pArr.add(a.getPointA());
                                        pArr.add(a.getPointB());
                                        pArr.add(b.getPointA());
                                        pArr.add(b.getPointB());
                                        pArr.add(c.getPointA());
                                        pArr.add(c.getPointB());
                                        Collections.shuffle(pArr);
                                        mLastModifcation = new
                                                           TspModificationSet(3);
                                        mLastModifcation.mAdd.add(new
                                                  Edge(pArr.get(0),pArr.get(1)));
                                        mLastModifcation.mAdd.add(new
                                                  Edge(pArr.get(2),pArr.get(3)));
                                        mLastModifcation.mAdd.add(new
                                                  Edge(pArr.get(4),pArr.get(5)));
                                        mLastModifcation.mRemove.add(a);
                                        mLastModifcation.mRemove.add(b);
```

```
                                        mLastModifcation.mRemove.add(c);
                                } while (!tryModification(mLastModifcation));
                        } while (mOldSet.contains(mLastModifcation));
                        if (mOldSet.size() >= mMaxNeighbor - 1) {
                                mOldSet.clear();
                                mHasMore = false;
                        }
                }
                return new TspPrototypeSet(mOtherSolution, mLastModifcation);
        }
}
```

2.4.5 TspCostSetExplizit

Die Klasse `TspCostSetExplicit` implementiert eine Funktion zur Berechnung der Kosten von expliziten in Mengenrepräsentation vorliegenden TSP-Lösungen.

```
public class TspCostSetExplizit implements CalculateCostsByExplicit {
        public double calculateCostsByExplicit(Object solution) {
                if (solution instanceof TspExplizitSet) {
                        TspExplizitSet tmp = (TspExplizitSet) solution;
                        double cost=0;
                        for (Iterator<Edge> it = tmp.iterator(); it.hasNext();) {
                                Edge tmpEdge = it.next();
                                cost += tmpEdge.getCost();
                        }
                        return cost;
                }
                return 0;
        }
}
```

2.4.6 TspCostSetPrototype

Die `TspCostSetPrototype`-Klasse implementiert eine Funktion zum Berechnen der Kosten einer TSP-Lösung, die mengenrepräsentiert als Prototyp vorliegt.

```
public class TspCostSetPrototype implements CalculateCostsByPrototype {
        public double calculateCostsByPrototype(PrototypeSolution
                                                solutionPrototype) {
                TspPrototypeSet tmp = (TspPrototypeSet) solutionPrototype;
                TspCostSetExplizit calculator = new TspCostSetExplizit();
```

In diesem Fall lassen sich die Kosten aus den Kosten der anderen Lösung zuzüglich der Kosten der Änderung effizient berechnen.

```
                return calculator.calculateCostsByExplicit(tmp.OtherSolution()) +
                        tmp.ModificationInformation().getCost();
        }
}
```

2.5 Bipartition

In dem Paket `Bipartition` liegen alle Klassen, die zur Problemstellung Bipartition gehören.

2.5.1 BipartitObject

Über Objekte der Klasse `BipartitObject` wird im Folgenden entschieden, ob sie zur Lösung dazugehören oder nicht. Sie besitzen eine Identität, Kosten und eine Angabe darüber, ob sie in der Lösung enthalten sind.

```
public class BipartitObject {
        public int mIdent;
        public double mCost;
        public boolean mInSolution;
        BipartitObject(int ident, double cost){
                mIdent = ident;
                cost = mCost;
                mInSolution = false;
        }
        public boolean equals(Object obj) {
                if (obj instanceof BipartitObject) {
                        BipartitObject tmp = (BipartitObject) obj;
                        return tmp.mIdent == mIdent &&
                               tmp.mInSolution == mInSolution;
                }
                return false;
        }
}
```

2.5.2 BipartitExplizit

`BipartitExplizit` repräsentiert explizite Lösungen des Bipartition-Problems. Es entspricht exakt `ArrayList<BipartitObject>`, ist also nur eine Umbenennung.

```
public class BipartitExplizit extends ArrayList<BipartitObject> {
        private static final long serialVersionUID = 1L;
}
```

2.5.3 BipartitPrototypeCostIterativ

Die Klasse `BipartitPrototypeCostIterativ` implementiert sowohl das `PrototypeSolution` als auch das `CalculateCostsByPrototype` Interface. Sie repräsentiert Prototyplösungen des Bipartitionsproblems, ohne die Kosten explizit mitzuspeichern.

```
public class BipartitPrototypeCostIterativ implements PrototypeSolution,
                                                      CalculateCostsByPrototype {
        private double mMaxCost;
        private Object mOtherSolution;
        private BipartitModificationInformation mModificationInformation;
        public Object OtherSolution() {
                return mOtherSolution;
        }
        public BipartitModificationInformation ModificationInformation() {
                return mModificationInformation;
        }
        BipartitPrototypeCostIterativ(Object otherSolution,
                                      BipartitModificationInformation modInf,
                                      double maxCost){
                mOtherSolution = otherSolution;
                mModificationInformation = modInf;
                mMaxCost = maxCost;
        }
```

Mit `ThisSolution()` werden die Änderungen aus der Modifikation angewendet.

```
        public Object ThisSolution() {
                if (mModificationInformation == null) {
                        return mOtherSolution;
                } else {
                        BipartitExplizit otherSolution = null;
```

```
                        if (mOtherSolution instanceof BipartitExplizit) {
                                otherSolution = (BipartitExplizit) mOtherSolution;
                        }
                        if (mOtherSolution instanceof
                                        BipartitPrototypeCostIterativ) {
                                BipartitPrototypeCostIterativ tmp =
                                   (BipartitPrototypeCostIterativ) mOtherSolution;
                                otherSolution =
                                   (BipartitExplizit) tmp.ThisSolution();
                        }
                        BipartitExplizit bipartit =
                                        (BipartitExplizit) otherSolution.clone();
                        if (mModificationInformation.mAdd>0)
                              bipartit.get(mModificationInformation.mAdd).
                                        mInSolution = true;
                        if (mModificationInformation.mRemove>0)
                              bipartit.get(mModificationInformation.mRemove).
                                        mInSolution = false;
                        return bipartit;
                }
        }
```

Die Kostenberechnung erfolgt durch explizite Konstruktion der Lösung und anschließende Kostenberechnung von dieser expliziten Lösung.

```
        public double calculateCostsByPrototype(PrototypeSolution
                                                solutionPrototype) {
                BipartitCostExplicit tmp = new BipartitCostExplicit(mMaxCost);
                return tmp.calculateCostsByExplicit(
                           solutionPrototype.ThisSolution());
        }
}
```

2.5.4 BipartitPrototypeCostSaved

Die Klasse `BipartitPrototypeCostSaved` erweitert die Klasse `BipartitPrototypeCostIterativ` um die explizite Mitspeicherung der Kosten der anderen Lösung. Dadurch ist es schneller möglich, die Kosten dieser Lösung zu bestimmen.

```
public class BipartitPrototypeCostSaved extends BipartitPrototypeCostIterativ {
        private double mOtherCost;
```

Hierzu wird direkt beim Erzeugen eines Objekts die Kosten der anderen Lösung bestimmt und abgespeichert.

```
        BipartitPrototypeCostSaved(Object otherSolution,
                                BipartitModificationInformation modInf,
                                double maxCost) {
                super(otherSolution, modInf, maxCost);
                mOtherCost = 0;
                if (otherSolution instanceof BipartitPrototypeCostSaved) {
                        BipartitPrototypeCostSaved tmp =
                           (BipartitPrototypeCostSaved) otherSolution;
                        mOtherCost = calculateCostsByPrototype(tmp);
                }
                if (otherSolution instanceof BipartitExplizit) {
                        BipartitExplizit tmp = (BipartitExplizit) otherSolution;
                        BipartitCostExplicit tmpCalc = new
                                        BipartitCostExplicit(maxCost);
                        mOtherCost = tmpCalc.calculateCostsByExplicit(tmp);
                }
        }
```

Da die Modifikationen die mit ihnen verbundene Kostenänderung enthalten, lassen sich somit effizient die neuen Kosten bestimmen.

```
        public double calculateCostsByPrototype(PrototypeSolution
                                                solutionPrototype) {
                if (solutionPrototype.ModificationInformation() == null)
                        return mOtherCost;
                else {
                        BipartitModificationInformation mod =
                            (BipartitModificationInformation)
                            solutionPrototype.ModificationInformation();
                        return mOtherCost + mod.mDeltaCost;
                }
        }
}
```

2.5.5 BipartitModificationInformation

Die Objekte der Klasse `BipartitModificationInformation` repräsentieren Änderungsinformationen an einer Bipartitionslösung.

In der Aufgabenstellung waren zwei verschiedene Nachbarschaften und damit auch Modifikationen gefordert. Zum Einen war nur der Austausch von einem Element aus der Lösungsmenge mit einem dort nicht enthaltenen erlaubt. Hierzu lassen sich die Indizes dieser Elemente mit `mRemove` und `mAdd` explizit angeben.

In der anderen Nachbarschaft war zusätzlich ausschließliches Hinzufügen *oder* Entfernen von einem Objekt gestattet. Das geschieht über das Setzen von `mAdd` bzw. `mRemove` auf -1, was andeuten soll, dass nichts hinzugefügt bzw. entfernt werden muss.

```
public class BipartitModificationInformation{
        int mRemove;
        int mAdd;
        double mDeltaCost;
        BipartitModificationInformation(int add, int remove, double deltaCost){
                mRemove = add;
                mAdd = remove;
                mDeltaCost = deltaCost;
        }
        protected BipartitModificationInformation clone(){
                return new BipartitModificationInformation(mAdd, mRemove,
                                                           mDeltaCost);
        }
}
```

2.5.6 BipartitConstructNeighborhood

Um von einer Bipartitionslösung Nachbarn zu erzeugen, lässt sich ein Objekt der Klasse `BipartitConstruct-Neighborhood` verwenden.

```
public class BipartitConstructNeighborhood implements ConstructNeighborhood {
        private int mGraphSize;
        private double mMaxCost;
        private boolean mOnlySwitch;
```

Um die beiden Typen noch Nachbarn zu unterscheiden (siehe Abschnitt 2.5.5), lässt sich die Variable `onlySwitch` setzen. `false` deutet an, dass auch das ausschließliche Entfernen und Hinzufügen gestattet ist. Zudem müssen außer der Problemgröße `graphSize` die maximalen Kosten `maxCost` angegeben werden, die eine Lösung enthalten darf.

```
        BipartitConstructNeighborhood(double maxCost, int graphSize,
                                      boolean onlySwitch){
                mGraphSize = graphSize;
```

```
                mMaxCost = maxCost;
                mOnlySwitch = onlySwitch;
        }
        public BipartitIterator constructNeighborhood(Object solution) {
                return new BipartitIterator(mOnlySwitch,
                                            mMaxCost,mGraphSize,solution);
        }
}
```

2.5.7 BipartitIterator

Die Klasse `BipartitIterator` ist die Iteratorklasse, die es ermöglicht, die Nachbarschaft einer Bipartitionslösung zu durchlaufen.

```
public class BipartitIterator implements PrototypeIterator {
        private BipartitExplizit mOtherSolution;
        private BipartitModificationInformation mCurrentModification;
        private int mGraphSize;
        private double mMaxCost, mOldCost;
        private boolean mHasMore;
        public boolean hasMoreElements() {
                return mHasMore;
        }
        private Vector<Integer> mInSolution;
        private Vector<Integer> mOutSolution;
        private int mInCount, mOutCount;
        private int mInCurr, mOutCurr;
        private int mInFirst, mOutFirst;
        BipartitIterator(boolean onlySwitch, double maxCost, int GraphSize,
                         Object otherSolution){
                mOtherSolution = null;
                if (otherSolution instanceof BipartitExplizit) {
                        mOtherSolution = (BipartitExplizit) otherSolution;
                }
                if (otherSolution instanceof BipartitPrototypeCostIterativ) {
                        BipartitPrototypeCostIterativ tmp =
                                (BipartitPrototypeCostIterativ) otherSolution;
                        mOtherSolution = (BipartitExplizit) tmp.ThisSolution();
                }
                BipartitCostExplicit tmp = new BipartitCostExplicit(maxCost);
                mOldCost = tmp.calculateCostsByExplicit(mOtherSolution);
                mGraphSize = GraphSize;
                mMaxCost = maxCost;
                mInSolution = new Vector<Integer>(GraphSize);
                mOutSolution = new Vector<Integer>(GraphSize);
```

Beim Erzeugen eines neuen Iterators wird eine Liste von in der Lösung enthaltenen Objekten und eine Liste von nicht enthaltenen Objekten angelegt. Falls auch das ausschließliche Hinzufügen und Entfernen von Elementen erlaubt ist, wird in diese Listen jeweils noch -1 aufgenommen.

```
                if (!onlySwitch) {
                        mInSolution.add(-1);
                        mOutSolution.add(-1);
                }
                for (int i = 0; i< mGraphSize;i++){
                        if (mOtherSolution.get(i).mInSolution) {
                                mInSolution.add(i);
                        } else {
                                mOutSolution.add(i);
                        }
                }
```

```
                mInCount = mInSolution.size();
                mOutCount = mOutSolution.size();
```

Als Startlösung wird aus beiden Listen ein Objekt zufällig gewählt.

```
                mInFirst = mInCurr = (int) (Math.random() * mInCount);
                mOutFirst = mOutCurr = (int) (Math.random() * mOutCount);
```

Da nicht sichergestellt ist, dass es überhaupt Nachbarn gibt, muss direkt nach dem ersten gesucht werden.

```
                findNextSolution();
        }
```

Diese Funktion ist hier etwas knapper als gewöhnlich. Es wird die schon im Voraus berechnete Lösung temporär gespeichert und am Ende zurückgeliefert. Dazwischen wird nach der übernächsten Lösung gesucht, um festzustellen, ob sie existiert. Durch die frühzeitige Suche kann das `hasMoreElements()`-Flag direkt gesetzt werden.

```
        public PrototypeSolution nextElement() {
                BipartitModificationInformation tmpMod = mCurrentModification;
                mCurrentModification = findNextSolution();
                return new BipartitPrototypeCostIterativ(mOtherSolution,
                                                         tmpMod, mMaxCost);
        }
```

Die Funktion `findNextSolution()` sucht nach dem nächsten Nachbarn.

```
        private BipartitModificationInformation findNextSolution(){
                mHasMore = true;
```

Diese äußere Schleife sorgt dafür, dass die Suche spätestens nach einem vollen Umlauf abgebrochen wird.

```
                for (int k=0;k<mInCount * mOutCount;k++) {
```

Durch die Einführung der Listen mit hinzufügbaren (nicht enthaltene) und entfernbaren (enthaltene) Objekten kann ich nun die Variablen `mInCurr` und `mOutCurr` lexikographisch hochzählen, ohne besondere Ausnahmen zu beachten. Hier ist im Gegensatz zu `TspModificationPermutation` (siehe 2.3.4 auf Seite 11) nicht verlangt, dass `mInCurr` $<$ `mOutCurr` gilt.

```
                        mInCurr++;
                        if (mInCurr == mInCount){
                                mInCurr = 0;
                                mOutCurr++;
                                if (mOutCurr == mOutCount){
                                        mOutCount = 0;
                                }
                        }
                        if (mInCurr == mInFirst && mOutCurr == mOutFirst) {
                                mHasMore = false;
                        }
                        double cost = 0;
                        int add, remove;
                        remove = mInSolution.get(mInCurr);
                        add = mOutSolution.get(mOutCurr);
                        if (remove != -1){
                                cost += mOtherSolution.get(remove).mCost;
                        }
                        if (add != -1){
                                cost -= mOtherSolution.get(add).mCost;
                        }
                        if (mOldCost + cost > 0) {
```

```
                                return new BipartitModificationInformation(add,
                                                                    remove,cost);
                        }
                }
                mHasMore = false;
                return null;
        }
}
```

2.5.8 BipartitCostExplicit

Mit Objekten der Klasse `BipartitCostExplicit` lassen sich die Kosten von einer explizit vorliegenden Bipartitionslösung berechnen. Um die Optimierung als Minimierungsproblem darzustellen, werden die Kosten aller einzelnen Elemente von `mMaxCost` abgezogen. Ist das Ergebnis < 0, so handelt es sich *nicht* um eine gültige Lösung ("der LKW ist überladen").

```
public class BipartitCostExplicit implements CalculateCostsByExplicit{
        private double mMaxCost;
        BipartitCostExplicit(double maxCost){
                mMaxCost = maxCost;
        }
        public double calculateCostsByExplicit(Object solution) {
                TspExplizitSet solutionBip = (TspExplizitSet) solution;
                double cost=mMaxCost;
                for (Iterator it = solutionBip.iterator(); it.hasNext();) {
                        BipartitObject tmp = (BipartitObject) it.next();
                        if (tmp.mInSolution) cost -= tmp.mCost;
                }
                return cost;
        }
}
```

2.6 Algorithms

In dem Paket `Algorithms` liegen alle Klassen, die zu den vier geforderten Algorithmen aus Aufgabenteil 1.c gehören.

2.6.1 AlgoMeta

`AlgoMeta` ist das Grundgerüst, das allen Algorithmen zugrunde liegt. Um diese abstrakte Klasse zu einem Algorithmus zu vervollständigen, muss die Methode `CalcNext()` überladen werden. Die Kommunikation nach außen erfolgt nur über die Schnittstellen, die `AlgoMeta` zur Verfügung stellt.

```
public abstract class AlgoMeta {
```

Es werden immer die aktuelle und die beste Lösung (was durchaus verschieden sein kann) gespeichert.

```
        private ExplicitSolutionPair mBest;
        protected ExplicitSolutionPair mCurrent;
```

Zudem benötigt jeder Algorithmus Objekte, um Kosten zu berechnen und von einer Lösung eine neue Nachbarschaft zu erzeugen.

```
        protected final CalculateCostsByPrototype mCostByPrototype;
        protected final CalculateCostsByExplicit mCostByExplicit;
        protected ConstructNeighborhood mNeighborhood;
```

Da nicht alle Algorithmen terminieren, ist ein Schrittzähler sowie eine maximale Schrittanzahl vorgesehen. Wird diese Funktion nicht benötigt, setzt man `mStepMax == 0`.

```
protected int mStepCount;
protected int mStepMax;
```

Zudem lässt sich auch beschränken, wie viele Schritte ohne Verbesserung stattfinden dürfen, ehe die Optimierung abbricht. Wird diese Funktion nicht benötigt, setzt man `mStepMaxUnchange == 0.`

```
protected int mStepLastGood;
protected int mStepMaxUnchange;
```

Die `mHasMore`-Variable kann von den einzelnen Algorithmen gesetzt werden, um zu signalisieren, dass sie terminiert sind.

```
protected boolean mHasMore;
```

Die `mStop` Variable ist public und dient dazu, einen laufenden Algorithmus-Thread zu unterbrechen. Sie wird nach jedem Schritt abgefragt.

```
public boolean mStop;
```

Die Funktion `StepCount()` gibt die aktuelle Anzahl der Berechnungsschritte zurück.

```
public int StepCount(){
        return mStepCount;
}
```

Der Standard-Konstruktor erwartet eine initiale Lösung des zu optimierenden Problems sowie zwei Objekte zur Kostenberechnung von expliziten Lösungen und Lösungsprototypen, außerdem ein Objekt zur Konstruktion der Nachbarschaft.

```
public AlgoMeta(Object initSolution,
                CalculateCostsByPrototype prototypeCost,
                CalculateCostsByExplicit explicitCost,
                ConstructNeighborhood neighborhood) {
        mCostByPrototype = prototypeCost;
        mCostByExplicit = explicitCost;
        mStepMax = 0;
        mStepMaxUnchange = 0;
        ResetStartSolution(initSolution,neighborhood);
}
```

Durch `ResetStartSolution(...)` wird der Algorithmus mit einer neuen Startlösung initialisiert. Zudem wird auch ein neuer Nachbarschaftskonstruktor erwartet, da dieser bei meiner Implementierung abhängig von der Problemgröße geändert werden muss.

```
public void ResetStartSolution(Object initSolution,
                ConstructNeighborhood neighborhood){
        mNeighborhood = neighborhood;
        mStop = true;
        mStepCount = 0;
        mStepLastGood = mStepMaxUnchange;
        mHasMore = true;
        double cost = mCostByExplicit.calculateCostsByExplicit(
                                    initSolution);
        mCurrent = mBest = new ExplicitSolutionPair(initSolution,cost);
        ResetStartSolution();
}
```

Jeder der einzelnen Algorithmen hat die Möglichkeit, eigene Initialisierungen vorzunehmen. Hierfür muss nur `ResetStartSolution()` überladen werden.

```
protected void ResetStartSolution(){}
```

Mit `ComputeUntilEnd(...)` lässt sich der Algorithmus so starten, dass er die Optimierung bis zur Termination weiterführt. Hierbei kann man ein `AlgoCallBack`-Objekt angeben, dessen Memberfunktion mit den aktuellen Zwischenergebnissen aufgerufen wird. Diese Funktion lässt sich von außen durch das Setzen der globalen `mStop`-Variable nach jedem Schritt unterbrechen.

```
public void ComputeUntilEnd() {
        ComputeUntilEnd(null);
}
public void ComputeUntilEnd(AlgoCallBack callBack) {
        mStop = false;
        mCallBack = callBack;
        if (mCallBack == null) {
                while(mHasMore && !mStop){
                        nextElement();
                }
        } else {
                while(mHasMore && !mStop){
                        mCallBack.NewSolution(nextElement());
                }
        }
}
```

Die Funktion `calcNext()` muss in jedem Algorithmus implementiert werden. Sie entspricht genau einem Optimierungsschritt. Es wird erwartet, dass das neue Zwischenergebnis samt der Kosten in `mCurrent` abgelegt wird.

```
protected abstract void calcNext();
```

Falls nicht gleich ein Endergebnis erwartet wird, lassen sich von außen auch einzelne Berechnungsschritte starten. Dies geschieht mit `nextElement()`. Diese Funktion merkt sich zudem die beste Lösung, die während der Optimierung vorgekommen ist, da dies nicht zwangsläufig die zuletzt zurückgelieferte Lösung sein muss. Wie oben beschrieben wird auf Verlangen die Optimierungsschrittzahl und die Anzahl der Schritte ohne Verbesserung überwacht. Gegebenenfalls bricht der Algorithmus ab und liefert nicht die letzte, sondern die gespeicherte beste Lösung zurück.

```
public Object nextElement(){
        if (mHasMore) {
                mStepCount++;
                calcNext();
        }
        if (mCurrent.cost < mBest.cost) {
                mStepLastGood = mStepCount + mStepMaxUnchange;
                mBest = mCurrent;
        }
        if (mStepMax!=0 && mStepCount>=mStepMax) mHasMore = false;
        if (mStepMaxUnchange != 0 && mStepLastGood <= mStepCount)
                mHasMore = false;
        if (!mHasMore)  mCurrent = mBest;
        return mCurrent.Solution;
}
```

Die Funktion `hasMoreElements()` gibt Auskunft darüber, ob noch weitere Optimierungsschritte möglich sind.

```
public boolean hasMoreElements() {
        return mHasMore;
}
```

Da die Kosten der aktuellen Lösung nicht standardmäßig zurückgegeben werden, lässt sich dies über `getCurrentCost()` erfragen. Auch die aktuelle Lösung lässt sich mit `getCurrentSolution()` erfragen.

```
        public double getCurrentCost() {
                return mCurrent.cost;
        }
        public Object getCurrentSolution(){
                return mCurrent.Solution;
        }
}
```

2.6.2 AlgoCallBack

Das `AlgoCallBack`-Interface wird von einer meiner GUI-Klassen implementiert, um Benachrichtigungen über neue Lösungen erhalten zu können.

```
public interface AlgoCallBack {
        void NewSolution(Object next);
}
```

2.6.3 ExplicitSolutionPair

Die `ExplicitSolutionPair`-Klasse kapselt eine explizite Lösung zusammen mit ihren Kosten ein. Dadurch ist es möglich, solche Paare in Collections abzulegen.

```
class ExplicitSolutionPair{
        Object Solution;
        double cost;
        ExplicitSolutionPair(){}
        ExplicitSolutionPair(Object _Solution, double _cost){
                Solution = _Solution;
                cost = _cost;
        }
        public boolean equals(Object obj) {
                if (obj instanceof ExplicitSolutionPair) {
                        ExplicitSolutionPair tmp = (ExplicitSolutionPair) obj;
                        return Solution.equals(tmp.Solution);
                }
                return false;
        }
        public int hashCode() {
                return Solution.hashCode();
        }
}
```

2.6.4 AlgoLocalSearch

Die `AlgoLocalSearch`-Klasse ist die Implementierung der lokalen Suche.

```
public class AlgoLocalSearch extends AlgoMeta {
```

Der Konstruktor wird unverändert an die Superklasse übergeben. Außer den dort vorgesehenen Initialisierungen sind keine weiteren erforderlich.

```
        public AlgoLocalSearch(Object initSolution,
                               CalculateCostsByPrototype prototypeCost,
                               CalculateCostsByExplicit explicitCost,
                               ConstructNeighborhood neighborhood) {
                super(initSolution, prototypeCost, explicitCost, neighborhood);
        }
```

`calcNext()` berechnet die nächste Lösung. Hierzu werden die Nachbarn solange durchsucht, bis ein Nachbar mit einem kleineren Kostenwert gefunden wurde. Falls kein solcher Nachbar existiert, terminiert die Suche.

```
        protected void calcNext() {
                ExplicitSolutionPair tmp = mCurrent;
                mHasMore = false;
                for(PrototypeIterator it =
                    mNeighborhood.constructNeighborhood(tmp.Solution);
                    it.hasMoreElements(); ){
                        Object nextSol = it.nextElement().ThisSolution();
                        double nextCost = mCostByExplicit.
                                        calculateCostsByExplicit(nextSol);
                        if (nextCost < mCurrent.cost){
                                mCurrent = new ExplicitSolutionPair(nextSol,
                                                                    nextCost);
                                mHasMore = true;
                                break;
                        }
                }
        }
}
```

2.6.5 AlgoTabuSearch

Die `AlgoTabuSearch`-Klasse ist die Implementierung von Tabusuche.

```
public class AlgoTabuSearch extends AlgoMeta {
        public AlgoTabuSearch(Object initSolution,
                              CalculateCostsByPrototype prototypeCost,
                              CalculateCostsByExplicit explicitCost,
                              ConstructNeighborhood neighborhood) {
                super(initSolution, prototypeCost, explicitCost, neighborhood);
```

Das Setzen der Variable `mStepMaxUnchange` auf 2 hat hier zur Folge, dass nach einem Pendeln zwischen einem Minimum und dessen Nachbarn die Optimierung terminiert.

```
                mStepMaxUnchange = 2;
        }
        protected void calcNext() {
                ExplicitSolutionPair tmp = mCurrent;
                boolean init = false;
```

Bei der Tabusuche wird, wie aus der Vorlesung bekannt, die ganze Nachbarschaft nach der besten Lösung durchsucht.

```
                for(PrototypeIterator it = mNeighborhood.
                    constructNeighborhood(tmp.Solution); it.hasMoreElements(); ){
                        Object nextSol = it.nextElement().ThisSolution();
                        double nextCost = mCostByExplicit.
                                        calculateCostsByExplicit(nextSol);
                        if (nextCost < mCurrent.cost || !init){
```

Bei der Suche ist die aktuelle Lösung als nächste Lösung *tabu*.

```
                                if (tmp.Solution.equals(nextSol)) continue;
                                mCurrent = new ExplicitSolutionPair(nextSol,
                                                                    nextCost);
                                init = true;
                        }
                }
        }
}
```

2.6.6 AlgoSimAnealing

Die `AlgoSimAnealing`-Klasse ist die Implementierung von Simulated Anealing (simuliertes Abkühlen).

```
public class AlgoSimAnealing extends AlgoMeta {
        public AlgoSimAnealing(Object initSolution,
                               CalculateCostsByPrototype prototypeCost,
                               CalculateCostsByExplicit explicitCost,
                               ConstructNeighborhood neighborhood) {
                super(initSolution, prototypeCost, explicitCost, neighborhood);
```

Die folgenden Werte sind empirisch ermittelt. Sie müssten eigentlich der jeweiligen Problemgröße angepasst werden. Dies hätte allerdings zur Folge, dass die Algorithmen nicht mehr so generisch wären. Diese Werte sind auf jeden Fall für Rundtouren mit bis zu 100 Ecken brauchbar.

```
                mStepMaxUnchange = 25000;
                mStepMax = 40000;
        }
```

Bei der Berechnung des nächsten Elements wird wie bei der lokalen Suche vorgegangen. Der Unterschied besteht nur darin, dass hier mit einer gewissen Wahrscheinlichkeit auch schlechtere Nachbarn genommen werden.

```
        protected void calcNext() {
```

Diese Wahrscheinlichkeit ist zum einen von der aktuellen Systemtemperatur abhängig. Diese lasse ich hier mit $\frac{1}{t}$ abfallen. Die konkrete Formel für die Systemtemperatur T ist

$$T = \frac{1}{100} \cdot \frac{\text{max. Schrittanzahl}}{\text{akt. Schrittanzahl}}$$

```
                double curTemp = 0.01 * mStepMax / mStepCount;
                for(PrototypeIterator it = mNeighborhood.
                    constructNeighborhood(mCurrent.Solution);
                    it.hasMoreElements();){
                        Object nextSol = it.nextElement().ThisSolution();
                        double nextCost = mCostByExplicit.
                                          calculateCostsByExplicit(nextSol);
                        if (nextCost < mCurrent.cost){
                                mCurrent = new ExplicitSolutionPair(nextSol,
                                                                    nextCost);
                                break;
                        }
```

Die Wahrscheinlichkeit, dass ein schlechterer Nachbar genommen wird, ist wie im Skript beschrieben

$$p = e^{\frac{c(\text{aktuelle Lösung}) - c(\text{nächste Lösung})}{T}}$$

wobei c die Kostenfunktion ist.

```
                        double prop = Math.exp((getCurrentCost() - nextCost)/
                                               curTemp);
                        if (Math.random() <= prop){
                                mCurrent = new ExplicitSolutionPair(nextSol,
                                                                    nextCost);
                                break;
                        }
                }
        }
}
```

2.6.7 AlgoEvolutionStrategy

Die `AlgoEvolutionStrategy`-Klasse ist die Implementierung von Evolutionsstrategien.

```
public class AlgoEvolutionStrategy extends AlgoMeta {
        public AlgoEvolutionStrategy(Object initSolution,
                                     CalculateCostsByPrototype prototypeCost,
                                     CalculateCostsByExplicit explicitCost,
                                     ConstructNeighborhood neighborhood) {
                super(initSolution, prototypeCost, explicitCost, neighborhood);
```

Die folgenden Werte sind empirisch ermittelt. Sie müssten eigentlich der jeweiligen Problemgröße angepasst werden. Dies hätte allerdings zur Folge, dass die Algorithmen nicht mehr so generisch wären. Diese Werte sind auf jeden Fall für Rundtouren mit bis zu 100 Ecken brauchbar.

```
                mStepMaxUnchange = 15000;
                mStepMax = 0;
        }
```

Für die Evolutionsstrategie wird eine Population (`mPopulation`) von Lösungen benötigt.

```
        private static final int mPopSize = 50;
        private int mNextPotParent;
        private Vector<ExplicitSolutionPair> mPopulation;
        public void ResetStartSolution(){
                mNextPotParent = 0;
                mPopulation = new Vector<ExplicitSolutionPair>(mPopSize);
```

Diese Population muss initialisiert werden. Hierzu werden einige Nachbarn der übergebenen Startlösung genommen. Allerdings werden diese nicht alle mit dem gleichen Iterator erzeugt, da das zu sehr ähnlichen initialen Populationsmitgliedern führen würde (z.B. bei lexikographischen Iterator).

Es wird hier von der Implementierung des Iterators erwartet, dass das erste zurückgegebene Element ein rein zufälliges ist.

```
                for (int n=0; n<mPopSize;n++){
                        PrototypeIterator it = mNeighborhood.
                                constructNeighborhood(mCurrent.Solution);
                        Object nextSol = it.nextElement().ThisSolution();
                        double nextCost = mCostByExplicit.
                                        calculateCostsByExplicit(nextSol);
                        ExplicitSolutionPair tmp = new ExplicitSolutionPair(
                                                        nextSol, nextCost);
                        mPopulation.add(tmp);
                }
```

Das beste Mitglied der initialen Population wird gesucht und als erste Lösung genommen.

```
                for (int n=0; n<mPopSize;n++){
                        if (mPopulation.get(n).cost < mCurrent.cost)
                                mCurrent = mPopulation.get(n);
                }
        }
```

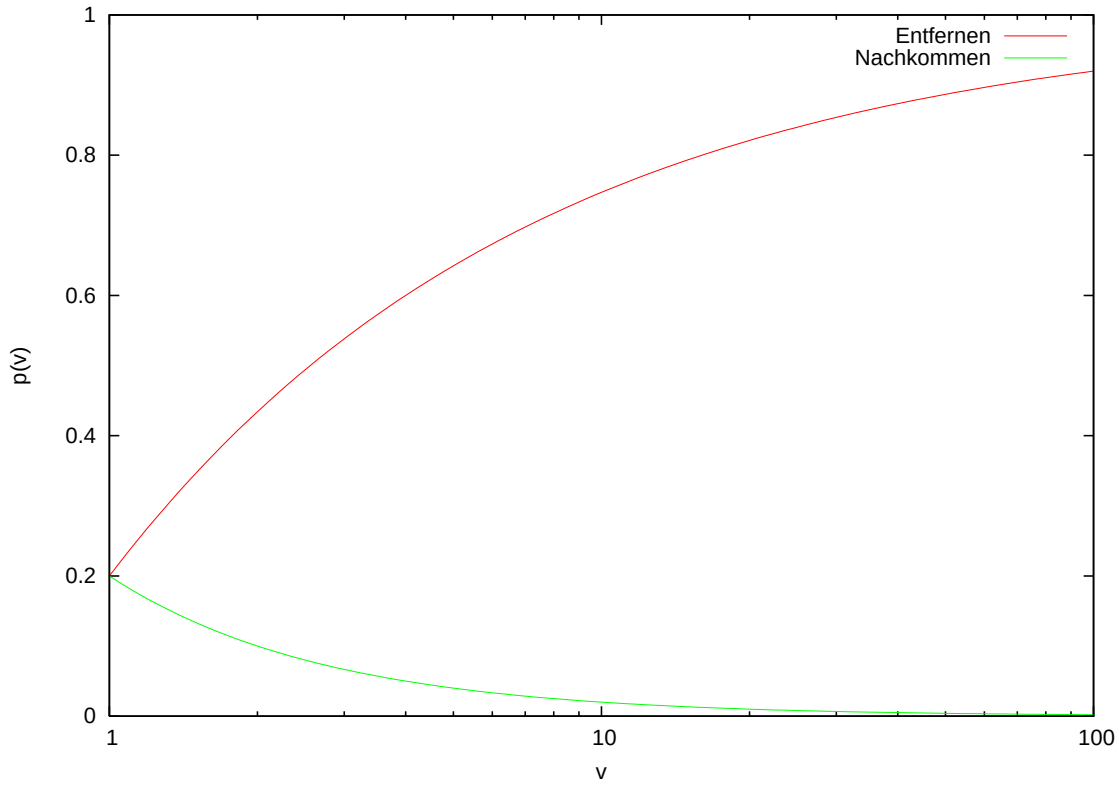
Die Berechnung der nächsten Lösung erfordert mehrere Schritte.

```
        protected void calcNext() {
                double prop;
```

Im Quellcode werden Wahrscheinlichkeitsformeln benutzt, die mithilfe der folgenden Notationen erklärt werden:

- $c(x)$ sind die Kosten der Lösung x

Abbildung 1: Fortplanzungs- und Entfernungswarscheinlichkeit einer Lösung bei einer Population von $n = 5$



- $c(x^*)$ sind die Kosten der besten Lösung der aktuellen Population
- n ist Größe der Population
- $v(x) = 50\frac{c(x)}{c(x^*)} - 49$ ist ein relatives Kostenmaß für Lösungen. Es ordnet der besten Lösung in der Population den Kostenwert 1 zu. Danach steigt es linear zu den absoluten Kosten der Lösung x an. Diese Maß hat sich in meinen Versuchen als sinnvoll erwiesen.

Die Wahrscheinlichkeit, dass eine Lösung sich fortpflanzen darf, liegt bei

$$p_f(x) = \frac{1}{n} \cdot \frac{1}{v(x)}$$

Die Wahrscheinlichkeit, dass ein Element aus der Population entfernt wird, ist

$$p_e(x) = 1 - \left(1 - \frac{1}{n}\right) \frac{1}{\sqrt{v(x)}}$$

Siehe hierzu auch Abbildung 1.

Zunächst wird ein Element der Population gesucht, das sich fortpflanzen darf.

```
double propFactor = 0.02 * getCurrentCost() / (double) mPopSize;
do {
        if (mNextPotParent==mPopSize) mNextPotParent=0;
        prop = propFactor /
               (mPopulation.get(mNextPotParent).cost
               - 0.98 * getCurrentCost());
        mNextPotParent++;
} while (Math.random() >= prop);
```

Nun wird von der ausgesuchten Lösung ein Nachkomme erzeugt. Das ist die erste Lösung, die der Nachbarschaftskonstruktor zurückliefert.

```
                PrototypeIterator it = mNeighborhood.constructNeighborhood(
                                       mPopulation.get(mNextPotParent - 1)
                                       .Solution);
                Object nextSol = it.nextElement().ThisSolution();
                double nextCost = mCostByExplicit.calculateCostsByExplicit(
                                                  nextSol);
```

Als nächstes wird nach einer Lösung gesucht, die aus der Population entfernt wird.

```
                propFactor = (1 - (1 / (double) mPopSize)) *
                             Math.sqrt(0.02 * getCurrentCost());
                do {
                        if (mNextPotParent==mPopSize) mNextPotParent=0;
                        prop = 1 - propFactor / Math.sqrt(
                               (mPopulation.get(mNextPotParent).cost -
                               0.98 * getCurrentCost()));
                        mNextPotParent++;
                } while (Math.random() >= prop);
```

Nun wird anstelle der eben ausgesuchten Lösung der oben erzeugte Nachkomme in die Population eingesetzt.

```
                mPopulation.set(mNextPotParent - 1,
                                new ExplicitSolutionPair(nextSol,nextCost));
```

Zum Schluss wird die Population nach der aktuell besten Lösung durchsucht.

```
                mCurrent = mPopulation.get(0);
                for (int i = 1; i< mPopulation.size();i++){
                        if (mCurrent.cost > mPopulation.get(i).cost)
                                        mCurrent = mPopulation.get(i);
                }
        }
}
```

3 GUI

Die von mir programmierte GUI zum TSP unterstützt sowohl 2-Opt wie auch 3-Opt, und das jeweils in der Mengen- und der Permutationsimplementierung. Zur Auswahl der gewünschten Kombination wird zum Programmstart ein Auswahlfenster (siehe Abbildung 2) angezeigt.

Meine GUI besteht aus 5 fest angeordneten Fenstern. Eines dient zur Steuerung, die anderen 4 zeigen jeweils einen Algorithmus und seine Fortschritte bei der Optimierung.

In den Abbildungen 3 und 4 auf der nächsten Seite sind zwei Screenshots zu sehen, die die Oberfläche vor dem Start und nach der Termination der Algorithmen zeigt.

Im Steuerfenster stehen 5 Bedienelemente zur Auswahl:

Knoten Die Knotenanzahl lässt sich über ein Auswahlfeld zwischen 5 und 500 variieren, was mir als sinnvolles Intervall erschien. Zudem wird in einigen Berechnungen die Annahme getroffen, dass mindestens 5 Kanten existieren.

Abbildung 2: Startdialog der GUI. Es lassen sich hier die verschiedenen Repräsentationen des TSP und dessen Nachbarschaft auswählen.

Abbildung 3: GUI mit initialer Rundtour

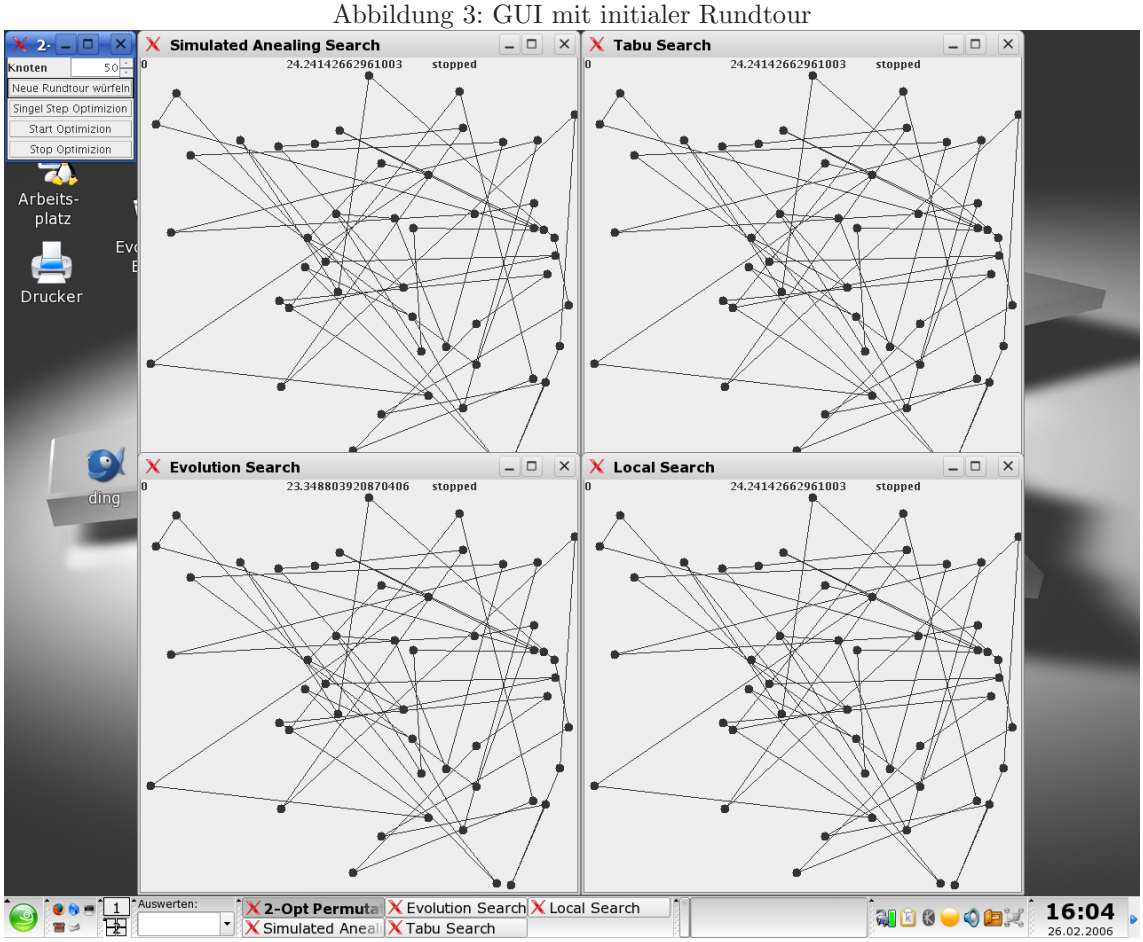
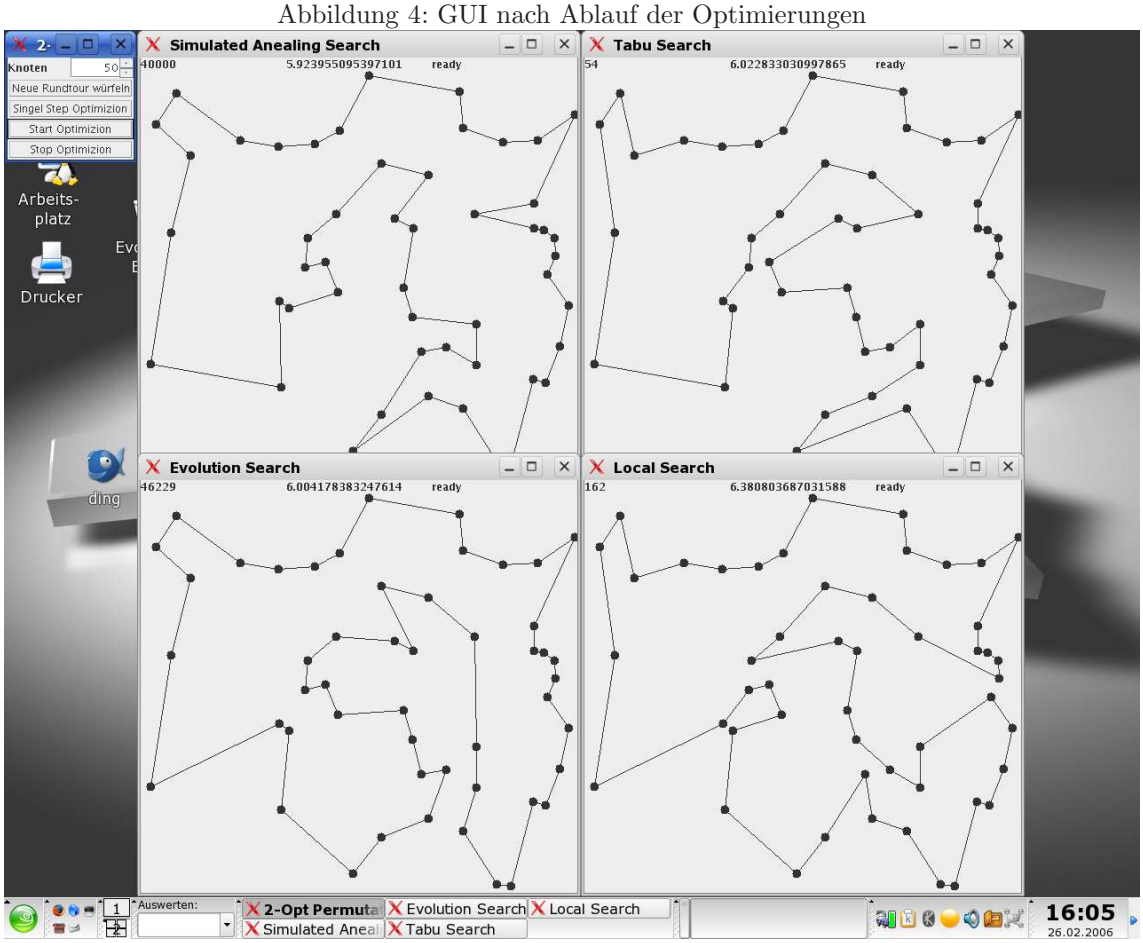


Abbildung 4: GUI nach Ablauf der Optimierungen



Ab einer Anzahl von 100 wird bei 2-Opt die Laufzeit der Tabusuche extrem lang, bei 3-Opt schon bei einer Anzahl von 25. Aus diesem Grund ist der Standartwert für die Knotenanzahl bei 2-Opt auf 50 und bei 3-Opt auf 15 gesetzt.
Die anderen Algorithmen skalieren besser mit der Systemgröße als die Tabusuche.

Neue Rundtour Würfeln erstellt eine neue zufällige Rundtour mit der angegebenen Knotenanzahl. Diese Rundtour wird allen Algorithmen als neue (Start-)Lösung übergeben und von ihren Fenstern angezeigt.

Single Step Optimation veranlasst jeden Algorithmus, genau einen weiteren Berechnungsschritt durchzuführen.

Start Optimation erstellt für jeden Algorithmus einen neuen Thread und lässt ihn darin solange rechnen, bis er terminiert.

Stop Optimation unterbricht eine mit *Start Optimation* begonnene Optimierung. Diese kann durch *Start Optimation* fortgesetzt werden.

Die zu den Algorithmen assoziierten Fenster zeigen jede Änderung in den Zwischenergebnissen direkt an. Zudem wird die aktuelle Schrittanzahl, die Länge der Rundtour und der Zustand des Algorithmusses (*stopped* / *running* / *ready*) angezeigt.