



HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG

Fakultät Informatik

Die Tabu-Suche

Prof. Dr. Wilhelm Erben
Wintersemester 2008/09

von

Max Nagl
nagl@fh-konstanz.de

Inhaltsverzeichnis

1	Einleitung	1
2	Die Tabu-Suche	2
2.1	Die grundlegende Idee	2
2.2	Eine einfache Erweiterung	4
2.3	Zweidimensionale Probleme	5
2.4	Probleme	7
3	Das Problem des Handlungsreisenden	9
3.1	Einführung	9
3.2	Implementierung	10
3.3	Lösen mit Hilfe der Tabu-Suche	13
4	Erweiterungen und Einstellungsmöglichkeiten	19
4.1	Speichererweiterungen	19
4.2	Speicherparameter	19
4.3	Weitere Parameter	20
4.4	Weiterentwicklungen	20
4.5	Determinismus	21
5	Fazit	22

1 Einleitung

In der heutigen Zeit möchte man möglichst alle Probleme schnell und einfach mit Hilfe von Computersystemen lösen. Bei einigen Optimierungsproblemen kann man allerdings die ideale Lösung oft nicht in vertretbarer Zeit finden, da kein effizienter Algorithmus dafür existiert oder bis jetzt gefunden wurde.

Aus diesem Grund werden für einige Probleme metaheuristische Optimierungsverfahren eingesetzt. Die Tabu-Suche hat sich als hierbei vielversprechendes Verfahren bewährt.

In dieser Ausarbeitung wird das grundsätzliche Verfahren und einige Erweiterungen beschrieben. Weiterhin wird versucht eine Lösung für Problem des Handlungsreisenden zu finden. Danach wird der Algorithmus mit einem genetischen Algorithmus verglichen. Im Anschluss werden zusätzliche Erweiterungen sowie die verschiedenen Parameter der Tabu-Suche vorgestellt.

Im Rahmen dieser Arbeit ist ein Java-Programm zur graphischen Erklärung des Verfahrens entstanden. Mit Hilfe dieses Programmes lassen sich zweidimensionale Funktionen, sowie Probleme des Handlungsreisenden lösen. Mehrere in dieser Ausarbeitung abgedruckte Bildschirmfotos wurden mit diesem Programm erstellt. Der vollständige Quellcode ist auf der beiliegenden CD enthalten.

2 Die Tabu-Suche

Das Problem bei den meisten metaheuristischen Optimierungsverfahren ist das Steckenbleiben in einem lokalen Minimum. So wird beim Hillclimbing für gewöhnlich das nächste lokale Minimum gefunden. Dort angekommen ist die Suche ohne spezielle Verfahren zur Diversifikation beendet. Eine Möglichkeit dem lokalem Minimum zu entkommen ist beispielsweise die simulierte Abkühlung (simulated annealing).

Auch die Tabu-Suche versucht lokalen Minima zu entkommen. Dazu werden bei der Auswahl des Folgezustandes nicht nur die aktuellen Nachbarschaften sondern auch der bisherige Verlauf betrachtet. Dazu werden die Zwischenergebnisse in so genannten Tabulisten gespeichert.

Das Verfahren gehört zu den ältesten metaheuristischen Optimierungsverfahren. Es wurde im Jahr 1987 von Fred Glover zum ersten Mal veröffentlicht. Seit dem gibt es regelmäßig neue Veröffentlichungen zu diesem Thema, was auf eine kontinuierliche Weiterentwicklung hindeutet.

2.1 Die grundlegende Idee

Im einfachsten Fall werden einfach alle verwendeten Zwischenlösungen in einer Liste gespeichert und dürfen anschließend nicht wieder verwendet werden. Dadurch kann keine Lösung mehrmals verwendet werden. Auf diese Weise kann das Verfahren nicht in einem lokalem Minimum stecken bleiben, noch kann es ständig im Kreis laufen, da sich im Laufe der Zeit der gesamte Lösungsbereich um das Minimum in der Tabuliste befindet und deshalb neue Bereiche erforscht werden müssen.

Während der Tabusuche werden folgende Schritte abgearbeitet:

1. Zu Beginn wird eine zufällige oder eine durch einen geeigneten Algorithmus generierte Lösung festgelegt.
2. Jetzt werden die Nachbarschaften der aktuellen Lösung berechnet.
3. Anschließend wird die Nachbarschaft ausgewählt, welche den besten Wert besitzt und noch nicht in der Tabuliste vorhanden ist. Diese kann auch einen schlechteren Wert als die aktuelle Lösung besitzen.
4. Die neue Lösung wird nun in die Tabuliste aufgenommen.
5. Es wird überprüft, ob die Abbruchbedingung erfüllt ist. Falls dies nicht der Fall ist, springt man zum Punkt 2 zurück.

Der folgende Pseudocode zeigt einen einfachen Implementierungsvorschlag dieses Verfahrens:

```

1 loesung = erzeugeStartLösung()
2 SOLANGE (Abbruchkriterium nicht erfüllt)
3     nachbarschaften = loesung.erzeugeNachbarschaften()
4     foreach (nachbarschaften as nachbarschaft)
5         if (tabuliste.enthält(nachbarschaft))
6             nachbarschaften.entferne(nachbarschaft)
7     if (nachbarschaften.anzahl == 0)
8         break
9     loesung = WähleBestenZug(nachbarschaften)
10    tabuliste.fügeHinzu(loesung)

```

Um dieses einfache Verfahren zu verdeutlichen, wird hier als Beispiel eine Funktion mit einer veränderlichen Variable verwendet. Die Bewertungsfunktion ist definiert mit $f(x) = \sin(x)$. Die Nachbarschaften sind definiert durch $N(x) = (x - 1), (x + 1)$. Dies bedeutet, der x-Wert wird bei jedem Schritt entweder um eins erhöht oder verringert. Der Verlauf des Algorithmus ist in Abbildung 2.1 abgebildet. Die Punkte stellen die verwendeten Werte im Verlauf der Zeit dar. Die grünen Pfeile zeigen den jeweils besseren Nachbarn. Wenn der grüne Pfeil durchgestrichen ist, darf er nicht verwendet werden. An seiner Stelle wird der schlechtere Nachbar – mit einem rotem Pfeil angedeutet – verwendet.

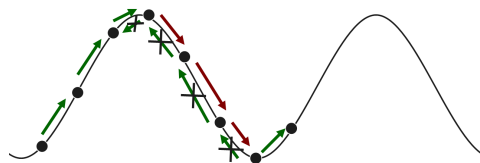


Abbildung 2.1: Optimierung der Funktion $f(x) = \sin(x)$

Als Startwert wurde der Punkt ganz links gewählt. Im ersten Schritt werden nun die Nachbarn berechnet. Als bester Wert ergibt sich nun der Wert weiter rechts. In den nächsten beiden Schritten wird der Wert jeweils weiter verbessert. Anschließend ist der Algorithmus in einem lokalem Maximum angekommen. Im nächsten Schritt sind beide berechneten Werte schlechter als der aktuelle Wert. Dazu kommt, dass der bessere der beiden Nachbarn bereits verwendet wurde und deshalb nicht noch einmal verwendet werden darf, deshalb wird der schlechtere der beiden Werte verwendet. Aus diesem Grund wandert der Punkt immer weiter nach rechts und entkommt somit dem lokalem Maximum.

Es ist leicht zu erkennen, dass der X-Wert immer weiter nach rechts wandern wird, da er nie die Richtung wechseln kann. Wäre der erste Punkt etwas weiter links gewählt worden, könnte im ersten Schritt der Nachbar auf der linken Seite gewählt werden, wodurch der X-Wert nur noch nach links wandern kann. Der erste Schritt bestimmt somit den Verlauf des gesamten Verfahrens.

Dieses Problem tritt zwar bei einer Zunahme der Dimensionalität, also der Anzahl der Veränderlichen, immer seltener auf, aber es bleibt trotzdem bestehen. Eine Lösung für dieses Problem wird im nächsten Abschnitt behandelt.

2.2 Eine einfache Erweiterung

Bei dieser Erweiterung wird die Größe der Tabuliste auf eine bestimmte Größe k begrenzt. In der Tabuliste werden also nur noch die letzten k Einträge gespeichert. Dieses Verfahren ist das ursprünglich von Fred Glover beschriebene Verfahren. Ein von ihm erstellter Ablaufplan, welcher das Verfahren verdeutlicht, ist in Abbildung 2.2 zu sehen.

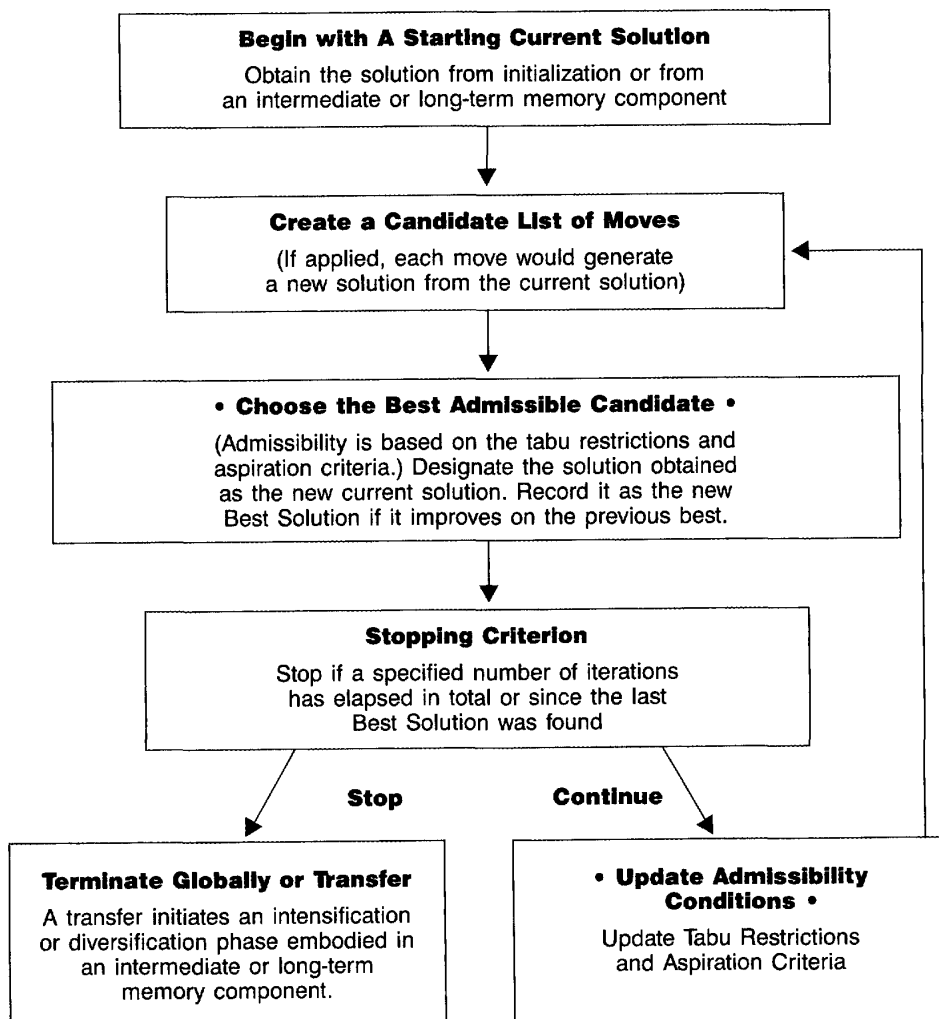


Abbildung 2.2: Ablaufplan für die Tabu-Suche von Fred Glover

Der folgende Pseudocode zeigt erneut eine mögliche Implementierung:

```

1 loesung = erzeugeStartLösung()
2 SOLANGE (Abbruchkriterium nicht erfüllt)
3     nachbarschaften = loesung.erzeugeNachbarschaften()
4     foreach (nachbarschaften as nachbarschaft)
5         if (tabuliste.enthält(nachbarschaft))
6             nachbarschaften.entferne(nachbarschaft)
7     if (nachbarschaften.anzahl == 0)
8         break
9     loesung = WähleBestenZug(nachbarschaften)
10    tabuliste.fügeHinzu(loesung)
11    SOLANGE (tabuliste.anzahl > k) tabuliste.entferneErstes()

```

Der einzige Unterschied zum vorherigen Pseudocode befindet sich in der letzten Zeile. Hier werden die zu alten Elemente aus der Liste entfernt.

Ein konkreter Implementierungsvorschlag in Java für einen Iterationsschritt ist im folgenden Code aufgeführt. Diese Implementierung ist sehr kurz, aber voll funktionstüchtig, und demonstriert, wie einfach der Algorithmus verwendet werden kann.

```

1 class Tabusuche {
2     ...
3     function step() {
4         Solution best = null;
5
6         List<Solution> newStates = path.generateAllNeighbours();
7
8         for (Solution newPath:newStates) {
9             if ( ((best == null) || (best.getValue()>newPath.getValue()))
10                && !tabuSolutions.contains(newPath) )
11                 best = newPath;
12         }
13
14         if (best != null)
15             tabuSolutions.add(best);
16
17         while (tabuStates.size() >= tabusize)
18             tabuStates.remove(0);
19     }
20     ...
21 }

```

2.3 Zweidimensionale Probleme

In den folgenden Beispielen werden zweidimensionale Probleme mit Hilfe der Tabu-Suche minimiert. In Abbildung 2.3 von Prof. Dr. B. Engels, ist der Idealfall einer solchen Suche im zweidimensionalen Raum illustriert. Dabei findet der Algorithmus zu Beginn eine lokales Minimum und springt anschließend zum nächsten Minimum weiter.

In der Realität ist der Verlauf bei Weitem nicht so schön, wie es die Entwickler des Verfahrens in ihren Veröffentlichungen darstellen.

Hier wird nun versucht die zweidimensionale Funktion $|\sin(x/6) * \sin(x)| + |\sin(y/6) * \sin(y)|$ im Wertebereich von $[0,20]$ für x und y zu minimieren. Das globale Minimum befindet sich im bei den Werten $x = 10,95$ und $y = 10,95$. Da die Optimierungsfunktion aus mehreren Sinusfunktionen zusammengesetzt wurde, gibt es einige lokale Minima. Die

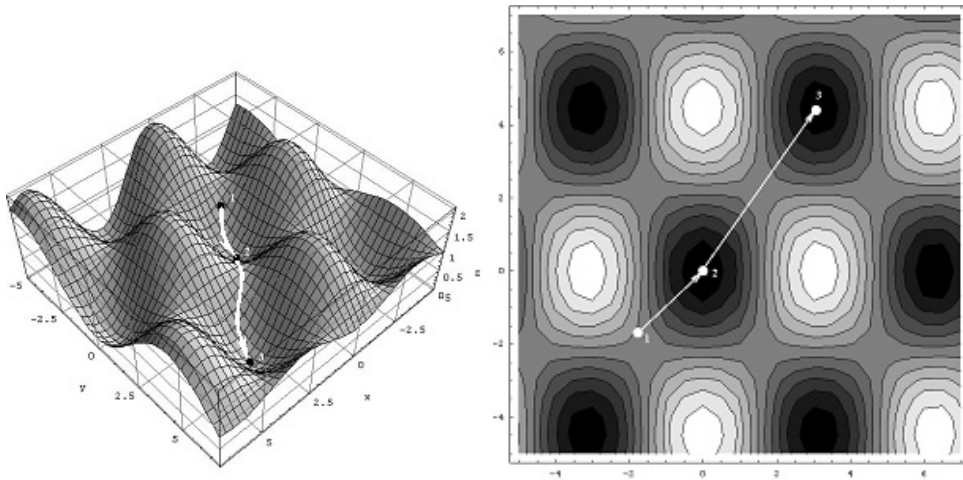


Abbildung 2.3: Der Idealfall im zweidimensionalen Raum

Nachbarschaft wurde definiert durch $N(x, y) = \{(x - 1, y - 1), (x, y - 1), (x + 1, y - 1), (x - 1, y), (x + 1, y), (x - 1, y + 1), (x, y + 1), (x + 1, y + 1)\}$.

Abbildung 2.4 zeigt mehrere Zustände im Verlauf des Verfahrens. Eine helle Hintergrundfarbe deutet einen guten Wert an, eine dunkle Hintergrundfarbe dagegen einen schlechten Wert. Das grüne Kreuz symbolisiert die aktuelle Position, die roten Kreuze stellen die verbotenen Position aus der Tabuliste dar. Links oben befindet sich der Startzustand, gefolgt von Zuständen im Abstand von 20 Iterationsschritten.

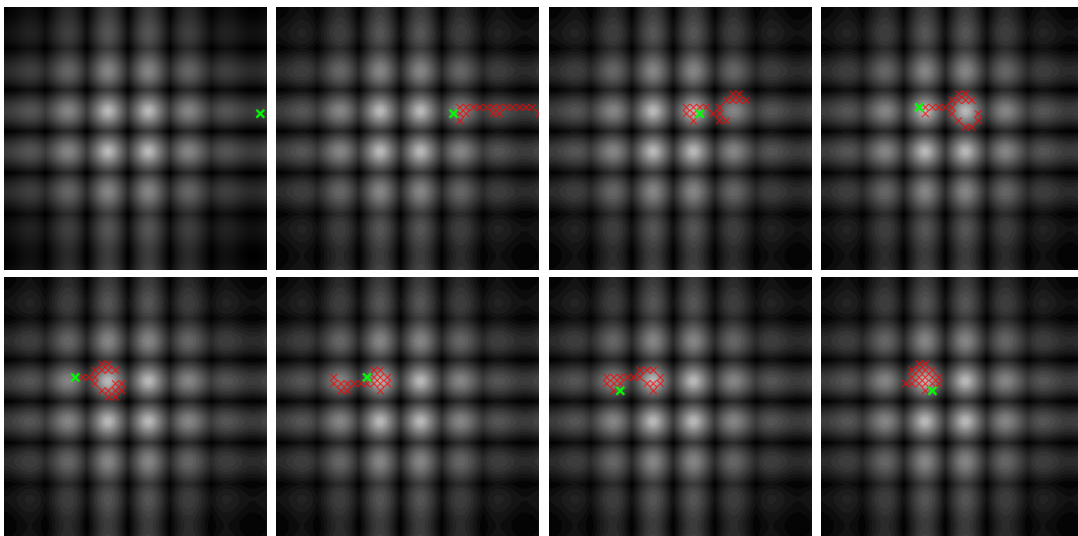


Abbildung 2.4: Minimierung einer Funktion

Zu Beginn läuft der Algorithmus wie gewünscht, indem er ein lokales Minimum findet und diesem wieder entkommt und zum Nächsten springt. Ab dem fünften Bild erkennt man allerdings, dass der Algorithmus in einen Zyklus gefangen ist und immer zwischen

den gleichen Minima hin und herspringt. Um diesem Zyklus zu entkommen wurde die Größe der Tabuliste auf 40 erhöht. Der weitere Verlauf ist Abbildung 2.5 gezeigt. Man erkennt, dass sich die Anzahl der roten Kreuze erhöht hat und weitere Minima, inklusive dem globalem Minimum, gefunden wurden.

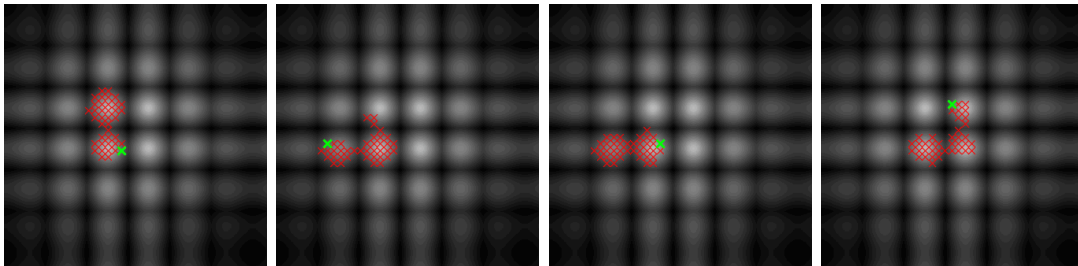


Abbildung 2.5: Minimierung einer Funktion mit vergrößerter Tabuliste

2.4 Probleme

In diesem Abschnitt werden zwei ungelöste Probleme der Tabu-Suche untersucht.

2.4.1 Sackgassen

Bei der Tabu-Suche kann es vorkommen, dass sich das Verfahren in einem Zustand befindet, in dem keine gültigen Nachbarn existieren, da alle möglichen Nachbarn sich in der Tabuliste befinden. Man könnte in so einem Fall von einer Sackgasse sprechen. Verdeutlicht wird dies in Abbildung 2.6. Nach dem nächsten Schritt wird es keine gültigen Nachbarn mehr geben, da alle benachbarten Punkte verboten sind.

Es ist nicht einfach zu entscheiden wie man am besten auf eine solche Situation reagiert. Die einfachste Lösung wäre die Suche hier abzubrechen und die aktuelle Position als Endwert zu verwenden. Dies ist aber nicht sehr günstig, da experimentell festgestellt wurde, dass das Sackgassenproblem nur selten in einem Minimum auftritt. Als weiteres Problem kommt hinzu, dass bei Funktionen mit wenig Dimensionen Sackgassen sehr häufig auftreten, wodurch die Optimierung recht schnell zu Ende wäre. Eine Alternative ist das älteste Element so oft aus der Tabuliste zu entfernen, bis es wieder einen gültigen Nachbarn gibt. Das Hauptproblem hierbei ist, dass sich die Tabuliste bei gehäuften Sackgassen kaum vollständig füllen kann. Dennoch werden hier bei Tests im zweidimensionalen Raum akzeptable Werte erzielt.

2.4.2 Plateaus

Die Tabu-Suche hat enorme Probleme mit Bewertungsfunktionen, welche Plateaus enthalten. Als einfaches Beispiel wurde hier eine Funktion gewählt, welche bei steigendem

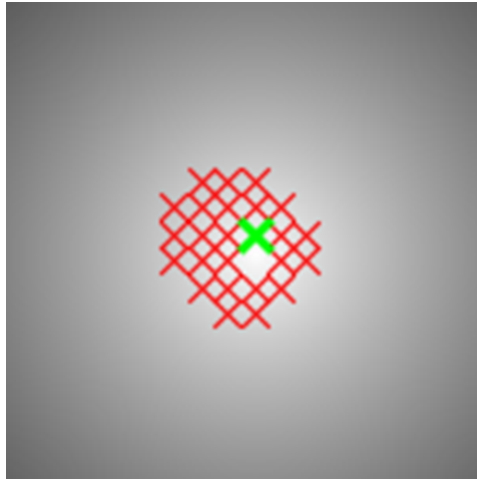


Abbildung 2.6: Nach dem nächsten Schritt befindet sich der Algorithmus in einer Sackgasse

x-Wert einen besseren Wert erzielt. Der y-Wert spielt keine Rolle. Um Plateaus künstlich zu erzeugen, wurde die Funktion gerundet, so dass nicht jede Erhöhung der x-Wertes eine bessere Bewertung erzielt. Das Resultat ist in Abbildung 2.6 zu sehen.

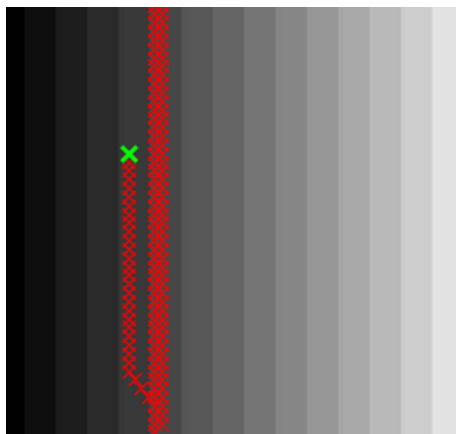


Abbildung 2.7: Die Tabu-Suche hat enorme Probleme mit Plateaus

Wenn die Bewertungsfunktion bei allen Nachbarn den gleichen Wert liefert, wird der erste erzeugte Nachbar verwendet, in diesem Fall der nach oben links verschobene Punkt. Da dies bei Plateaus der Fall ist, läuft das Verfahren bei einem erreichtem Plateau in die Ecke oben links. Falls es sich bei dem Plateau um ein lokales Minimum handelt, also alle Punkte um das Plateau herum einen schlechteren Wert besitzen, kann die Suche im schlimmsten Fall dem Minimum nur entkommen, wenn sich alle Punkte des Plateaus in der Tabuliste befinden. Bei Funktionen mit Plateaus sollte die Größe der Tabuliste also möglichst größer gleich dem größten Plateau gesetzt werden. Da eine zu große Tabuliste allerdings andere Probleme mitbringt, sollte eher versucht werden Plateaus aus der Bewertungsfunktion zu entfernen.

3 Das Problem des Handlungsreisenden

Dieser Abschnitt widmet sich dem Problem des Handlungsreisenden. Der englische Name lautet „Travelling salesman problem“, woher die häufig verwendete Abkürzung TSP stammt. Zu Beginn wird kurz das Problem des Handlungsreisenden beschrieben, anschließend wird eine Implementierung der wichtigsten Teile in Java vorgestellt. Im Anschluss daran wird versucht mittels der Tabu-Suche eine Lösung für das Problem zu finden.

3.1 Einführung

Das Problem des Handlungsreisenden ist ein kombinatorisches Optimierungsproblem. Es geht darum eine vorgegebene Menge an Punkten in einer beliebigen Reihenfolge zu besuchen und im Anschluss wieder am Ausgangspunkt zu enden. Dabei darf jeder Punkt nur exakt ein einziges Mal besucht werden. Die Schwierigkeit besteht darin die Reihenfolge der Punkte so zu wählen, dass die gesamte Strecke möglichst kurz ist.

Der Komplexitätstheorie zufolge ist das TSP NP-vollständig. Das bedeutet, dass im schlimmsten Fall der beste Weg nur gefunden wird, wenn alle Lösungen systematisch durchprobiert wurden. Dies hat eine exponentielle und somit für viele Punkte ungeeignete Laufzeit zur Folge. Aus diesem Grund wird häufig versucht dieses Problem mittels heuristischer Optimierungsverfahren zu lösen.

Das TSP wird meist als Graph modelliert. Hierbei stellen die Knoten die Punkte und das Gewicht der Kanten die Entfernungen zwischen zwei Punkten dar. Sind zwei Knoten nicht durch Kanten verbunden, so existiert keine direkte Verbindung zwischen den zwei Punkten. Zur Vereinfachung wird allerdings meist angenommen, dass es sich um einen vollständigen Graph handelt, bei dem jeder Knoten mit jedem anderen Knoten verbunden ist.

Es gibt verschiedene Arten des TSP. So existieren beispielsweise symmetrische TSP, bei denen jede Kante in beide Richtungen den gleichen Wert besitzt, sowie unsymmetrische, bei denen dies nicht der Fall ist. Des Weiteren spricht man von einem metrischen TSP, wenn die Kantenlängen der Dreiecksungleichung entsprechen. In diesem Fall gilt: $d_{AC} \leq d_{AB} + d_{BC}$, wobei die Funktion d die Distanz zwischen zwei Punkten berechnet.

Meist wird das Problem auf reale Probleme im zweidimensionalen Raum angewendet. Hat ein Handlungsreisender beispielsweise die Aufgabe, sieben Kunden in sieben verschiedenen Städten zu besuchen und anschließend wieder zurückzukehren, so hat man ein geometrisches und symmetrische TSP. Daher auch der Name „Das Problem des Handlungsreisenden“.

In den folgenden Kapiteln werden metrische und symmetrische TSPs beschrieben.

3.2 Implementierung

Um ein TSP mit Hilfe der Tabu-Suche lösen zu können, muss eine Klasse implementiert werden, welche einen Pfad durch das TSP beschreibt. Diese Klasse wird mit dem englischen Ausdruck „Path“ bezeichnet und in diesem Abschnitt beschrieben.

3.2.1 Repräsentation der Daten

Um den Pfad innerhalb der Klasse genau beschreiben zu können wird auf die Darstellung des TSP als Graph zurückgegriffen.

Die Knoten werden in einem Feld von zweidimensionalen Punkten dargestellt. Für die Kanten wird ein Vektor mit Ganzzahlen verwendet, in welchem die Reihenfolge der besuchten Punkte gespeichert wird. Die Nummerierung der Knoten beginnt bei 0. Werden beispielsweise zehn Punkte in aufsteigender Reihenfolge besucht, enthält der Vector die Daten [0,1,2,3,4,5,6,7,8,9].

Da Java die verwendeten Datenstrukturen von Haus aus passend zur Verfügung stellt, ist die Deklaration der Variablen wie im folgenden Quelltext zu sehen sehr einfach.

```
1 public class Path implements Solution {
2     ...
3     Vector<Integer> waypoints;
4     Point[] points;
5     ...
6 }
```

3.2.2 Initialisierung

Bei der Initialisierung der Klasse werden dem Konstruktor die zu besuchenden Punkte übergeben. Die Klasse errechnet dann selbstständig eine gültige Lösung, mit der das Verfahren beginnt. Im folgenden Quelltext wird die Implementierung des Konstruktors gezeigt.

```
1 public class Path implements Solution {
2     ...
3     public Path(Point[] points) {
4         this.points = points;
5         waypoints = new Vector<Integer>();
6         waypoints.add(0);
    }
```

```

7   for (int i = 1; i < points.length; i++)
8       waypoints.insertElementAt(i, (int)(Math.random()*(waypoints.size()+1)));
9   }
10  ...
11 }

```

3.2.3 Berechnung der Nachbarn

Für die Tabu-Suche ist es auch wichtig alle Nachbarn einer Lösung zu berechnen. Deshalb wird die Funktion `generateAllNeighbours` implementiert. Die Funktion gibt eine Liste von Nachbarn zurück.

Es werden alle Nachbarn berechnet, welche durch das Verfahren „Pairwise exchange“, auch „2-opt“ genannt, erzeugt werden können. Bei diesem Verfahren werden zwei Punkte in der Reihenfolge vertauscht. Als Resultat erhält man immer eine gültige neue Lösung. Um alle Möglichkeiten zu berechnen, wird jeder Punkt einmal mit jedem anderen Punkt vertauscht. Abbildung 3.1 zeigt einen Pfad vor und nach dem Vertauschen des zweiten und fünften Knotens.

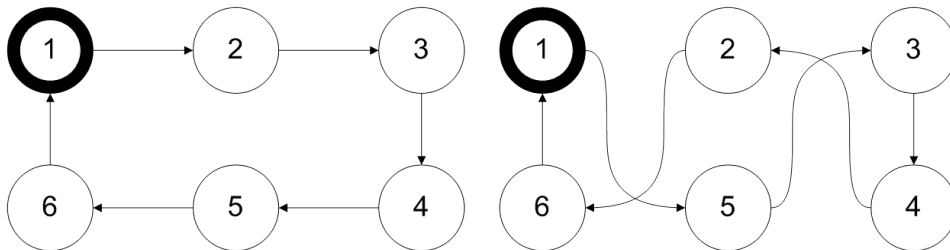


Abbildung 3.1: Vertausch des zweiten und fünften Knotens

Die Anzahl der möglichen Vertauschungen liegt hier $\sum_{i=2}^{n-1} i = \frac{n^2-n}{2} - 1 \approx \frac{n^2}{2}$. Die Anzahl der Zahlen erhöht sich also quadratisch mit der Anzahl der Punkte. Die Quelltext für diese Funktion ist im Anschluss abgedruckt.

```

1 public class Path implements Solution {
2     ...
3     public List<Solution> generateAllNeighbours() {
4         List<Solution> list = new Vector<Solution>();
5         int size = waypoints.size();
6         for (int r1 = 0; r1 < size - 1; r1++)
7             for (int r2 = r1 + 1; r2 < size; r2++)
8                 list.add(generate2Opt(r1, r2));
9         return list;
10    }
11
12    private Path generate2Opt(int r1, int r2) {
13        Path path = clone(this);
14        int t = path.waypoints.get(r1);
15        path.waypoints.set(r1, path.waypoints.get(r2));
16        path.waypoints.set(r2, t);
17        return path;
18    }
19    ...
20 }

```

Der Methode `clone(Path alt)` erzeugt eine exakte Kopie des Pfades `alt`.

3.2.4 Vergleich von zwei Lösungen

Um zu überprüfen, ob eine Lösung in der Tabuliste enthalten ist, müssen zwei Lösungen auf Gleichheit überprüft werden können. Dazu wird in Java für gewöhnlich die Funktion `equals` verwendet, bei welcher einem Objekt ein anderes Objekt übergeben wird und nur dann wahr zurückgeliefert wird, falls beide Objekte identisch sind.

Um zu überprüfen, ob die Objekte identisch sind, wird zuerst überprüft, ob es sich bei dem übergebenen Objekt um einen anderen Pfad handelt und anschließend ob die Kanten sowie die Knoten identisch sind. Da Java hierfür Operationen zur Verfügung stellt, ist dies wie im folgendem Quelltext zu sehen schnell geschehen.

```
1 public class Path implements Solution {
2     ...
3     public boolean equals(java.lang.Object o) {
4         if (!(o instanceof Path)) return false;
5         Path path = (Path) o;
6         return (path.waypoints.equals(waypoints) && (path.points == points));
7     }
8     ...
9 }
```

3.2.5 Bewerten einer Lösung

Um entscheiden zu können, welcher der beste Nachbar ist, müssen die Pfade bewertet werden können. Dazu wird der Klasse `path` eine Funktion `getValue` zur Berechnung des Wertes hinzugefügt. Darin werden die Längen aller Teilstrecken berechnet und summiert. Die Länge einer Teilstrecke entspricht dem euklidischer Abstand zwischen Anfangs- und Endpunkt. Die Algorithmus ist hier abgedruckt:

```
1 public class Path implements Solution {
2     ...
3     public int getValue() {
4         int x = -1, y = -1;
5         int value = 0;
6         for (Integer i:waypoints) {
7             int nx = points[i].x, ny = points[i].y;
8             if (x >= 0) value += Math.hypot(nx-x, ny-y);
9             x = nx; y = ny;
10        }
11        value += Math.hypot(points[0].x-x, points[0].y-y);
12        return value;
13    }
14    ...
15 }
```

Da sich dieser Wert nur ganz zu Beginn und bei der Berechnung neuer Nachbarn ändert, ist es hilfreich, diesen Wert zwischenspeichern und nicht jedes mal beim Aufruf der Funktion neu zu berechnen.

3.2.6 Weitere benötigte Funktionen

Es werden noch einige weitere Funktionen benötigt, deren Quelltext hier nicht aufgelistet ist, da sie nur sehr wenig bis gar keine Anwendungslogik enthalten. Die Funktionen sind hier dennoch kurz beschrieben:

generateNeighbour generiert einen einzelnen zufälligen Nachbarn. Der Verwendungszweck wird später beschrieben.

clone erzeugt eine exakte Kopie des Pfades `alt`. Wird zur Erstellung von Nachbarn benötigt.

paint stellt den Pfad graphisch dar.

toString erstellt eine textuelle Beschreibung des Pfades.

3.3 Lösen mit Hilfe der Tabu-Suche

3.3.1 Der erste Versuch

Bereits beim ersten Versuch mit zwanzig Städten wurde festgestellt, dass das Verfahren bereits recht schnell einen guten Weg findet. Allerdings handelt es sich meist um ein lokales Minimum. Leider wurde auch festgestellt, dass das Verfahren diesem Minimum nur selten entkommt. Dies liegt daran, dass es bei 20 Städten bereits eine riesige Anzahl an Kombinationen gibt. Um einem Minimum zu entfliehen müssen nun alle Nachbarn, welche in der Umgebung liegen, in der Tabuliste enthalten sein. Dazu muss die Tabuliste allerdings sehr groß sein. Bei einer Größe von mehreren Tausend Einträgen war es aber möglich den meisten Minima zu entkommen.

Ein große Tabuliste hat allerdings den Nachteil, dass es sehr viele Schritte braucht, bis die Tabuliste vollständig gefüllt wurde. Des Weiteren steigt auch der Speicher- und CPU-Bedarf mit der Größe der Tabuliste an. Dies war auch eindeutig im Test zu erkennen. Die Laufzeit und der benötigte Arbeitsspeicher stiegen schlagartig an.

Als Alternative kann man nun nicht nur Lösungen direkt vergleichen, sondern auch die Teilnehmer an der Vertauschung. Dies wird im folgenden Abschnitt erklärt.

3.3.2 Vergleich der Vertauschungen

Um die Größe der Tabuliste zu minimieren, sollen nur noch die Vertauschungen überprüft werden. Dies wird in Abbildung 3.2 gezeigt. Ausgehend vom Pfad links oben werden zuerst die Punkte 2 und 7 und im Anschluss daran die Punkte 3 und 6 vertauscht. Wir erhalten nun den Pfad unten links. Würden wir die Pfade exakt vergleichen, wäre es nun zulässig, erneut die Punkte 2 und 7 zu vertauschen, da die Reihenfolge des Ergebnisses

noch nicht in der Tabuliste enthalten ist. Vergleichen wir allerdings die Vertauschungen, so ist diese Lösung nicht zulässig.

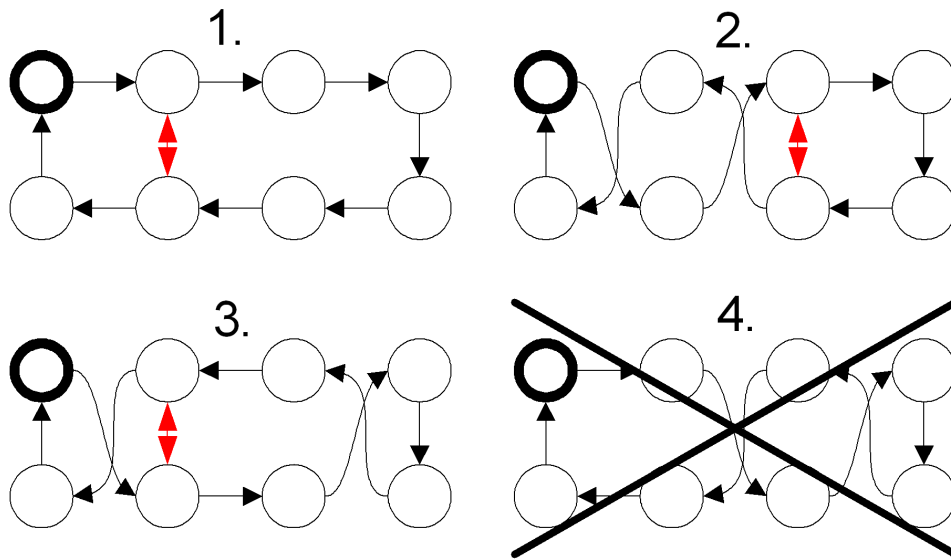


Abbildung 3.2: Vergleich der Vertauschungen

Dieses Verfahren verringert die Größe der Tabuliste dramatisch. Wie wir bereits bei der Generierung der Nachbarn festgestellt haben, nimmt die Anzahl der möglichen Vertauschungen quadratisch mit der Anzahl der Punkte zu. Die Menge möglicher Reihenfolgen vergrößert sich hingegen exponentiell. Bei zwanzig Städten existieren beispielsweise 121645100408832000 Reihenfolgen, aber nur 189 Vertauschungen.

Hierzu müssen die Funktionen `generateAllNeighbours` sowie `equals` angepasst werden. In der ersten muss gespeichert werden, welche Knoten vertauscht wurden und in der zweiten sollten die Vertauschungen anstatt der Reihenfolge verglichen werden.

Neue Tests ergaben nun, dass das Verfahren bereits bei einer Tabuliste mit zwanzig Einträgen einem lokalen Minimum mit Leichtigkeit entkommt.

Im Allgemeinen kann man sagen, dass bei der Tabu-Suche sich die Größe der Tabuliste verringert, wenn man nur die Schritte, welche zur Lösung führen, vergleicht und nicht die Lösungen auf exakte Gleichheit überprüft. Weiterführende Informationen hierzu sind in [Glover, 1990b] zu finden.

Ein Nachteil dieses Verfahrens ist allerdings, dass die beste Lösung eventuell nicht gefunden werden kann. Man stelle sich vor, das Beispiel aus Abbildung 3.2 verläuft in genau entgegengesetzter Reihenfolge. Der Startwert ist also der Pfad rechts unten. Zuerst werden die Punkte 2 und 7 sowie 3 und 6 vertauscht. Nun erhält man den Pfad rechts oben. Eine erneute Vertauschung der Punkte 2 und 7 ist nun verboten, obwohl dies zur optimalen Lösung führen würde. Eine Möglichkeit dieses Problem zu verringern wird im nächsten Abschnitt gezeigt.

3.3.3 Aspirationskriterien

Um zu vermeiden, dass ein sehr guter Nachbar nicht genommen wird, da er auf Grund der Tabuliste als ungültig deklariert wurde, ist es günstig bestimmte Kriterien einzuführen, damit dieser Nachbar dennoch verwendet werden kann. Diese Kriterien werden als Aspirationskriterien bezeichnet. Aspiration stammt aus den Lateinischen und bedeutet in diesem Zusammenhang „Bestrebung“.

Ein einfaches Aspirationskriterium, welches bei nahezu jedem Problem verwendet werden kann, ist einen Nachbarn auf jeden Fall zuzulassen, wenn dieser das beste Ergebnis seit dem Beginn der Suche liefert. Hierfür muss allerdings ein Langzeitgedächtnis eingeführt werden, in dem die beste Lösung seit Beginn des Verfahrens gespeichert ist. Des Weiteren muss für jeden ungültigen Nachbarn ein weiterer Vergleich durchgeführt werden. Dies sind allerdings, im Vergleich zum gewonnenen Nutzen, nur sehr geringe Kosten.

Auch Aspirationskriterien kann man häufig in der Literatur über die Tabu-Suche finden. Allerdings werden für verschiedene Probleme unterschiedliche Kriterien benötigt. Das einzige allgemeingültige Kriterium aus der Literatur wurde oben beschrieben.

3.3.4 Partielle Suche von Nachbarn

Wie bereits mehrmals angesprochen steigt die Anzahl der möglichen Nachbarn mit der Anzahl an Knoten quadratisch an. Aus diesem Grund ist es bei großen TSPs nur schwer möglich alle Nachbarn zu berechnen. Bei 100 Knoten existieren 5049 Nachbarn, bei 1000 Knoten sind es bereits 500499. Eine so große Anzahl an Nachbarn ist auf Grund des limitierten Arbeitsspeichers von PCs nicht mehr zu berechnen. Man braucht allerdings nicht alle Nachbarn zu berechnen, es reicht aus nur einen Teil zu betrachten. Hierfür wird die Funktion `generateNeighbour` benötigt, welche einen zufälligen Nachbar erzeugt.

Wenn nun nicht alle Nachbarn erzeugt werden, kann dies zur Folge haben, dass die beste Lösung trotz direkter Nachbarschaft nicht gefunden wird. Dabei können weder eine kleine Tabulisten, noch Aspirationskriterien helfen.

Weiterhin reduziert sich durch eine partielle Suche der Nachbarn leider nicht die benötigte Größe der Tabuliste. Wenn beispielsweise 10000 Nachbarn existieren und nur 100 Nachbarn generiert werden, so wird im Schnitt nur jede 100 Lösung gefunden. Bei einer Tabuliste mit 20 Einträgen liegt die Wahrscheinlichkeit einen oder mehrere Nachbarn zu erzeugen, welche in der Tabuliste vorhanden sind, bei 20 Prozent.

Trotz dieser Probleme haben Experimente ergeben, dass ein kleiner Bruchteil der Nachbarn ausreicht um die Suche in die richtige Richtung voranzutreiben. Die negativen Aspekte wirken sich hauptsächlich auf die Intensivierung, also die exakte Bestimmung der Minima aus.

3.3.5 Ein konkretes Beispiel

In Abbildung 3.3 ist ein Bildschirmaufnahme des Programms zu Beginn der Optimierung eines TSP zu sehen. Am oberen Rand ist eine Leiste angebracht, in der man das zu optimierende Problem mit einigen Parametern wählen kann. In diesem Fall ist als Problem ein zufällig generiertes TSP gewählt worden. Alternativ kann man auch zweidimensionale Funktionen oder vorgegebene TSPs optimieren lassen. Der erste Schieberegler bestimmt die Größe des TSP, der Zweite die Anzahl der zu generierenden Nachbarn. Falls dieser auf Null steht, werden alle Nachbarn berechnet. Der letzte Schieberegler bestimmt die Größe der Tabuliste. Diese Einstellungen werden nur dann angewandt, falls ein neues Problem zur Optimierung erstellt wird.

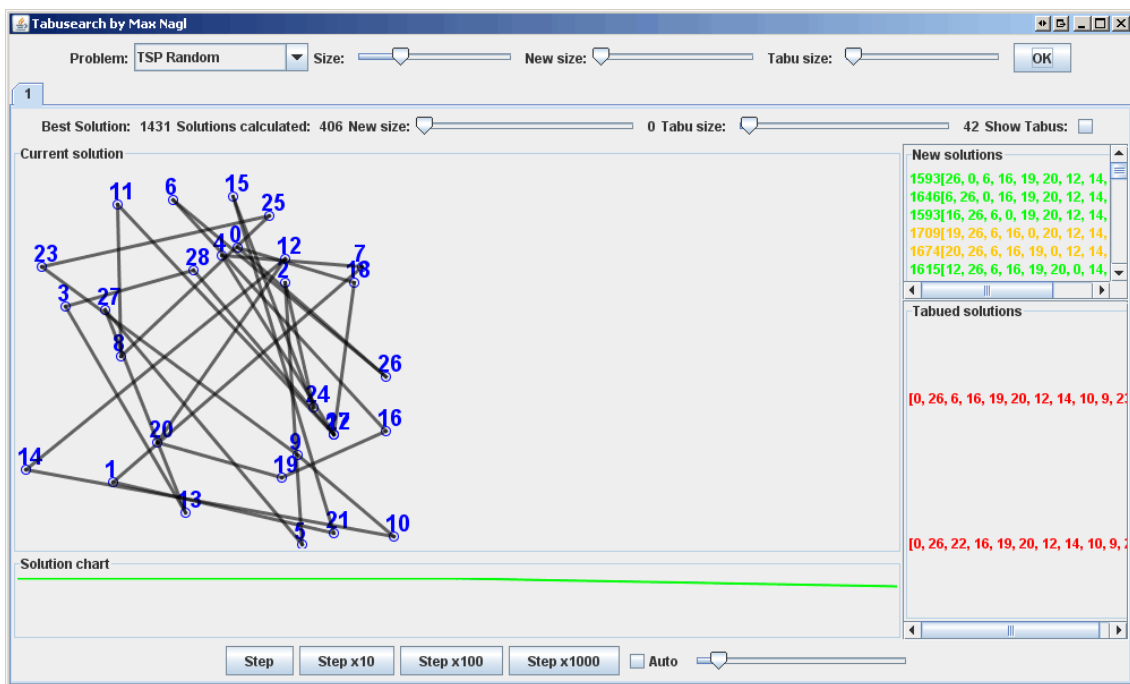


Abbildung 3.3: Tabusort zu Beginn einer Optimierung

Sobald ein neues Problem erstellt wurde, entstehen in der Zeile darunter neue Schieberegler für die Anzahl der Nachbarn sowie die Größe der Tabuliste. Diese Regler können verwendet werden um die Eigenschaften während der Optimierung zur Laufzeit anzupassen.

Darunter auf der linken Seite ist die aktuelle Lösung des Problems in schwarzer Farbe zu sehen. Unter der Ansicht ist ein Diagramm, das den Wert der Bewertungsfunktion über die Zeit anzeigt. Die grünen Teile zeigen eine Verbesserung, die roten Teile eine Verschlechterung an. Fährt man mit der Maus über das Diagramm, so wird nicht nur die aktuelle Lösung angezeigt, sondern es wird auch die vorhergegangene Lösung zu einem bestimmten Zeitpunkt mit rot eingezeichnet.

Auf der rechten Seite werden im oberen Bereich alle möglichen Nachbarn angezeigt. Lösungen, welche einen besseren Wert als die Aktuelle besitzen werden grün, schlechtere Lösungen mit gelb und verbotene Lösungen mit rot angezeigt. Im unteren Bereich sind die Lösungen aus der Tabuliste zu sehen. Auch hier gilt, dass die Lösung unter dem Mauszeiger angezeigt wird.

Am unteren Fensterrand sind nun einige Knöpfe zu sehen, mit denen man einen oder mehrere Iterationsschritte ausführen kann. Mit dem Kontrollkästchen „Auto“ wird die automatische Berechnung aktiviert. Mit dem Schieberegler kann man dann einstellen, wie viele Schritte pro Sekunde berechnet werden sollen.

In Abbildung 3.4 ist das Programm nach eintausend Schritten zu sehen. Man erkennt eine deutliche Verbesserung der Lösung. Weiterhin sieht man nun, dass sich die Tabuliste mit Einträgen gefüllt hat und aus diesem Grund auch einige Nachbarn ungültig sind. Im Diagramm erkennt man, dass sich der Algorithmus am Anfang sehr schnell verbessert hat und ein lokales Minimum gefunden hat. Anschließend hat sich der Wert mehrmals verschlechtert und wieder verbessert. Man kann deshalb davon ausgehen, dass mehrere lokale Minima gefunden wurden. Ob der beste Weg gefunden wurde, kann man leider nicht sagen.

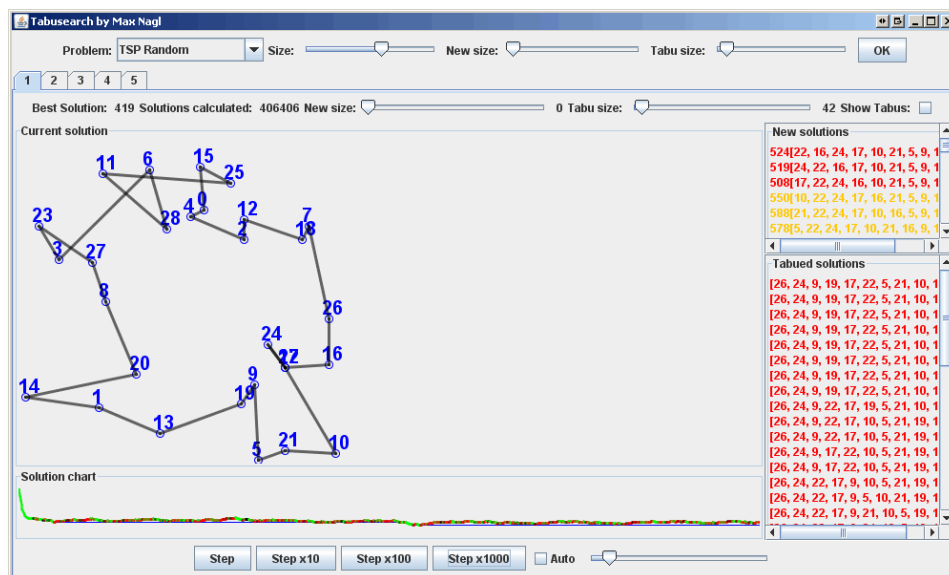


Abbildung 3.4: Tabusort zu nach 1000 Schritten

3.3.6 Vergleich mit GEATbx

GEATbx ist eine Matlab-Bibliothek für genetische Algorithmen. Mit Hilfe dieser Bibliothek wurde in [Nagl u. Hofmann, 2008] das bayerische 29-Städte-Problem gelöst. Hierbei handelt es sich um ein bekanntes TSP, mit dessen Hilfe oft verschiedene Optimierungsverfahren verglichen werden.

In der Arbeit von Herrn Nagl und Herrn Hofmann wurde versucht, für dieses Problem mit Hilfe von genetische Algorithmen eine gute Lösung in möglichst kurzer Zeit finden. Bei den Experimenten wurde ein Algorithmus gefunden, welcher in 17s eine durchschnittliche Entfernung von 2079km findet. Auch der beste Wert von 2020km wurde in zehn Versuchen gefunden.

Das erstellte Programm für die Tabusuche wurde auf das gleiche Problem angesetzt. Es wurden keine speziellen Anpassungen für das Problem vorgenommen. Die Tabuliste war auf 42 Einträge beschränkt. Bei einem Limit von 100 Iterationsschritten wurde ein Durchschnittswert von 2172km und ein Bestwert von 2072km erreicht. Die Laufzeit betrug weit weniger als eine Sekunde. Bei 1000 Iterationsschritten und einer Laufzeit von ungefähr einer Sekunde lagen die Ergebnisse mit im Durchschnitt 2065km und einem Bestwert von 2020km schon deutlich unter denen des genetischen Algorithmus.

Dazu muss noch gesagt werden, dass die Tabu-Suche in Java implementiert wurde und der genetischen Algorithmus in Matlab, welches eine erheblich schlechtere Performance liefert. Allerdings ließe sich das Java-Programm noch deutlich optimieren.

4 Erweiterungen und Einstellungsmöglichkeiten

In diesem Kapitel werden die Erweiterungen und Einstellungsmöglichkeiten der Tabu-Suche beschrieben. Einige Erweiterungen wie beispielsweise die Aspirationskriterien und die partielle Suche von Nachbarn wurden bereits erklärt und werden hier nicht nochmals im Detail beschrieben.

4.1 Speichererweiterungen

Neben der Tabuliste kann es durchaus sinnvoll sein noch weitere Speicher zu verwenden. Hier werden einige dieser Erweiterungen vorgestellt.

In einem Langzeitgedächtnis werden alle Lösungen oder Züge gespeichert, die bis jetzt durchgeführt wurden. Auch kann man speichern, wie oft eine Lösung bisher verwendet wurde. Diese Informationen kann man beispielsweise verwenden um Nachbarn, welche schon oft verwendet wurden, mit Strafpunkten zu versehen.

Ein Qualitätsgedächtnis speichert die besten gefundenen Lösungen. Diese können später verwendet werden um an ihnen weiter zu arbeiten.

Der Verbesserungsspeicher beinhaltet Züge, durch die das Ergebnis stark verbessert wurde. Diese Züge kann man nach dem eigentlichen Durchlauf der Tabu-Suche zusammen mit dem Qualitätsgedächtnis verwenden um mit Hilfe der Kombination aus guten Ergebnissen sowie guten Zügen einen neuen Bestwert zu erreichen. Außerdem können Nachbarn, welche durch einen guten Zug zu Stande gekommen sind, einen Bonus erhalten.

4.2 Speicherparameter

Die Größe der Tabuliste ist einer der wichtigsten Parameter. Ist sie zu klein, so bleibt der Algorithmus recht wahrscheinlich in einem lokalem Minimum stecken und hat somit ein schlechtes Diversifikationsverhalten. Ist sie zu groß, so wird die Intensivierung verschlechtert und die beste Lösung wird eventuell nicht gefunden.

Für gewöhnlich wird diese Größe experimentell bestimmt, es gibt aber auch Ansätze den idealen Wert automatisiert zur Laufzeit anzupassen. Wird beispielsweise immer wieder das gleiche lokale Minimum gefunden, so deutet dies auf eine zu geringe Größe hin. Dies

kann algorithmisch recht einfach erkannt werden. Daraufhin wird die Liste vergrößert. Nähert sich das Verfahren einem neuen Minimum, so sollte die Größe wieder verringert werden. Dies ist allerdings nicht so einfach zu erkennen. Weitere Informationen hierzu sind in [Bramer u. Devedzic, 2005] zu finden.

In [Glover, 1990a] wird beschrieben, dass eine Tabuliste mit fünf bis zwölf Einträgen sinnvoll ist. Als besonders gut habe sich die Sieben bewährt. Dies ist bemerkenswert, da sich der Mensch bei einem Optimierungsproblem, wie man in Experimenten festgestellt hat, auch in etwa sieben Lösungen merkt. Die Evolution ähnelt hier dem metaheuristischen Verfahren. In den Versuchen, die zu dieser Ausarbeitung durchgeführt wurden, lieferten solch kleinen Tabulisten allerdings nur selten gute Ergebnisse. Besonders bei einem TSP mit mehr als 50 Knoten und einer Tabuliste mit weniger als 20 Einträgen, blieb das Verfahren nahezu in jedem lokalem Minimum stecken.

Des Weiteren ist zu bestimmen wie viel Einfluss die anderen Speicher, wie zum Beispiel das Qualitätsgedächtnis, haben. Hierfür gibt es keine allgemein gültige Empfehlungen, da dies sehr stark vom zu optimierenden Problem abhängt.

4.3 Weitere Parameter

Einige weitere Parameter wurden in dieser Ausarbeitung schon besprochen, werden hier aber noch ein mal erwähnt.

Zum einen sollte man sich zwischen einer kompletten oder einer partiellen Nachbarschaftssuche entscheiden. Diese Entscheidung wird einem häufig abgenommen, da eine vollständige Suche oft gar nicht realisierbar ist.

Für die Vergleichsoperation kann man entweder die Lösung direkt vergleichen oder auch den Weg, der zur Lösung geführt hat. Beides bietet Vor- und Nachteile.

Weiterhin ist es sinnvoll sich für das Problem geeignete Aspirationskriterien zu überlegen.

4.4 Weiterentwicklungen

Wie bereits erwähnt wird die Tabu-Suche auch nach mehr als fünfzehn Jahren immer noch weiterentwickelt. Aktuelle Forschung wird besonders in den folgenden Bereichen betrieben:

Multiprozessorsysteme: Ziel ist es, das Verfahren auf möglichst vielen Prozessoren gleichzeitig auszuführen. Da die Synchronisation zwischen Prozessoren sehr gering ist (weniger als 1ms) kann hier problemlos innerhalb jedes Iterationsschrittes mehrmals synchronisiert werden.

Verteilte Systeme: Hierbei ist das Ziel, das Verfahren auf möglichst vielen, durch ein Netzwerk verbundenen Computern gleichzeitig auszuführen. Da die Synchronisati-

on hier im Bereich von Sekunden liegt, sollte eine Synchronisation nur nach mehreren Iterationsschritten stattfinden, wodurch dies um einiges schwerer zu realisieren ist als bei Multiprozessorsystemen.

Embedded-Systeme: Um das Verfahren auf Embedded-Systemen zum Laufen zu bringen muss unter anderem der Speicherbedarf auf ein Minimum reduziert werden.

Da die Tabu-Suche sehr einfach zu Implementieren ist, wird sie sehr häufig mit anderen metaheuristischen Optimierungsverfahren kombiniert. Auch hier wird noch Forschung betrieben.

4.5 Determinismus

Bei der Tabu-Suche handelt es sich in der originalen Fassung um ein deterministisches Verfahren. Dies bedeutet, dass das Verfahren mit gleichen Anfangswert und gleichen Parametern immer das gleiche Ergebnis liefert. Dies liegt daran, dass keine stochastischen Funktionen verwendet werden.

Verwendet man nun aber nicht die vollständige Nachbarschaftssuche, sondern ermittelt zufällige Nachbarn, so handelt es sich nicht mehr um ein deterministisches Verfahren, da die erzeugten Nachbarn von Zufall abhängen.

Auch der Determinismus ist Vergleichbar mit dem menschlichen Denken, da auch der Mensch auf deterministische Weise versucht eine Lösung zu finden.

Der Determinismus bringt Vor- sowie Nachteile mit sich, so macht es beispielsweise wenig Sinn das Verfahren mit gleichen Einstellungen auf mehreren Rechnern laufen zu lassen, da immer das gleiche Endergebnis gefunden wird.

5 Fazit

Bei der Tabu-Suche handelt es sich um ein sehr schnelles und einfach zu implementierendes metaheuristisches Optimierungsverfahren. Im Vergleich zu anderen Verfahren schneidet es meist recht gut ab.

Ein weiterer wichtiger Vorteil ist, dass es sehr flexibel ist. Man kann bei den meisten Problemen mit einer einfachen, allgemein gültigen Implementierung starten und später erweiterte und auf das Problem spezialisierte Verfahren hinzufügen.

Des Weiteren kann man das Verfahren auch gut mit anderen Optimierungsverfahren kombinieren, was häufig auch gemacht wird.

Literaturverzeichnis

- [Bramer u. Devedzic 2005] BRAMER, Max ; DEVEDZIC, Vladan: *Artificial Intelligence - Applications and innovations*. Springer Science + Business Media, Inc., 2005
- [De Jong u. a. 1997] DE JONG, K. ; FOGEL, D. ; SCHWEFEL, HP: Handbook of Evolutionary Computation. In: *IOP Publishing Ltd and Oxford University Press* (1997)
- [Glover 1989] GLOVER, F.: Tabu Search-Part I. In: *INFORMS Journal on Computing* 1 (1989), Nr. 3, S. 190
- [Glover 1990a] GLOVER, F.: Tabu search: A tutorial. In: *Interfaces* 20 (1990), Nr. 4, S. 74-94
- [Glover 1990b] GLOVER, F.: Tabu Search-Part II. In: *ORSA Journal on Computing* 2 (1990), Nr. 1, S. 4-32
- [Heppner 2005] HEPPNER, C.: *Tabu-Search-Übersicht/Einführung in eine moderne Meta-Heuristik*. 2005
- [Nagl u. Hofmann 2008] NAGL, Max ; HOFMANN, Andreas: *Ausarbeitung im Fach Genetische Algorithmen*. 2008
- [Russell u. a. 1995] RUSSELL, S.J. ; NORVIG, P. ; CANNY, J.F. ; MALIK, J. ; EDWARDS, D.D.: *Artificial intelligence: a modern approach*. Prentice Hall Englewood Cliffs, NJ, 1995
- [Yi u. a. 2006] YI, HE ; GUANGYUAN, LIU ; YUHUI, QIU: A Parallel Tabu Search Algorithm Based on Partitioning Principle for TSPs. In: *IJCSNS* 6 (2006), Nr. 8A, S. 146