Fachhochschule Brandenburg

Einsatz evolutionärer Algorithmen zur Optimierung der Tourenplanung eines Wachschutzunternehmens

Diplomarbeit
zur Erlangung des akademischen Grades
Diplom-Informatiker(FH)

vorgelegt von
Tino Schonert

Brandenburg 2003

Eingereicht am 4. Februar 2003

Betreuer:	Prof. Dr.-Ing. Jochen Heinsohn (FH-Brandenburg) Jörn Schlanert (Safe Wachschutz/Allservice GmbH)
1. Prüfer:	Prof. Dr.-Ing. Jochen Heinsohn
2. Prüfer:	Dipl.-Informatiker Ingo Boersch

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Hintergrund der Arbeit

Das Problem, welches im Rahmen dieser Diplomarbeit untersucht und gelöst werden soll, ist die Tourenplanung des Wachschutzunternehmens Safe Wachschutz/Allservice GmbH. Hier geht es darum, mehrere Kunden bzw. Wachobjekte von einem zentralen Fahrzeugdepot aus anzufahren und zu kontrollieren. Dazu stehen im Depot mehrere Fahrzeuge zur Verfügung. Jedes Wachobjekt muss innerhalb eines definierten Zeitfensters kontrolliert werden. Außerdem kann es vorkommen, dass auf einem Objekt mehrere Kontrollgänge vom Wachpersonal durchzuführen sind.
Aufgabe der Tourenplanung ist es, die unterschiedlichen Objekte so auf die einzelnen Fahrzeuge aufzuteilen, dass alle Zeitrestriktionen erfüllt sind und der dafür benötigte Zeitaufwand bzw. die Gesamtkilometerzahl möglichst minimal wird.
Die Tourenplanung gestaltet sich im Allgemeinen recht schwierig, da es sich dabei um ein sehr komplexes Problem handelt. Bisher erfolgt dies manuell durch den Einsatzleiter, welcher lediglich eine intuitive Zuordnung von verschiedenen Objekten zu einer Tour vornimmt. Die Planung und tatsächliche Abfahrtsreihenfolge liegt dann auf der Seite des Wachangestellten jeder einzelnen Tour. Bei dieser Vorgehensweise stellt sich natürlich die Frage, ob es nicht eine Möglichkeit gibt, die Tourenplanung rechnergestützt vornehmen zu lassen und eventuell effizientere Routen, im Hinblick auf Zeit- und Kostenaufwand, zu erhalten.
Man kann diesen Vorgang der Tourenplanung allgemein als *Optimierungsproblem* verstehen. Das Lösen solcher Optimierungsprobleme ist in einer Vielzahl von industriellen und wirtschaftlichen Bereichen von großer praktischer

Bedeutung. Man wird mit der Frage nach Optimalität konfrontiert, wenn es z.B. darum geht, den Gewinn einer Produktionsanlage zu maximieren, Arbeitsabläufe, Transportwege, die Tourenplanung oder einen Maschinenbelegungsplan so effektiv wie möglich zu gestalten, um viel Zeit und Kosten zu sparen.
Aufgrund der hohen Komplexität individueller Optimierungsprobleme gestaltet sich meistens die Ermittlung einer möglichst optimalen Lösung auf rein intuitiven Wege nahezu aussichtslos. Aus diesem Grund befasst man sich im Bereich des **Operations Research**[1] mit dem Lösen der unterschiedlichsten praktischen Aufgabenstellungen, welche eines gemeinsam haben: Unter Beachtung der gegebenen Umweltbedingungen ist ein bestimmtes Ziel bestmöglich zu erreichen. Das heißt, es ist ein Extremwertproblem zu lösen. Es liegt daher auf der Hand, dass besonders die Lösungsmethoden aus der Mathematik Anwendung beim Lösen solcher Aufgaben finden. Dazu muss zunächst aus der eigentlichen Problemstellung ein mathematisches Modell, beschrieben durch lineare Gleichungen und Ungleichungen, entwickelt werden.
Es gibt jedoch eine in der Praxis sehr wichtige Klasse von Optimierungsproblemen, bei der sich mathematische Methoden nicht zur Lösung eignen, da sich entweder kein mathematisches Modell aufstellen lässt oder die Anzahl der Gleichungen bzw. Ungleichungen astronomisch ansteigt. Dies ist die Klasse der *kombinatorischen Optimierungsprobleme*. Die kombinatorische Optimierung beschäftigt sich mit der typischen Frage: Welche aus einer Menge von (sehr vielen) Kombinationen ist die beste, kürzeste, billigste oder effizienteste?
Das wohl bekannteste kombinatorische Optimierungsproblem ist das Problem des Handlungsreisenden (engl. *Traveling Salesman Problem* (*TSP*)): Ein Handlungsreisender hat ausgehend von seinem Heimatort n Städte nacheinander in beliebiger Reihenfolge zu besuchen und muss schließlich wieder zu Hause ankommen. Gesucht ist eine optimale Route (kürzester Weg) zwischen diesen n Städten, so dass die Zeit, die er für seine Tour benötigt, minimal wird.

[1] Operations Research, auch Unternehmensforschung, Operationsforschung oder Optimalplanung genannt, ist eine Disziplin, die sich mit Verfahren zum Treffen rationaler Entscheidungen befasst. Operations Research ist prinzipiell dort anwendbar, wo es um das Problem geht, aus mehreren Handlungsmöglichkeiten diejenige zu wählen, die einen gewollten Zweck möglichst gut erreichen lässt (vgl. [We72]).

Bei kombinatorischen Optimierungsproblemen spielt die Anordnung und Reihenfolge eine entscheidende Rolle. Wie viele Möglichkeiten hat der Vertreter, diese n Städte nacheinander anzufahren? Für die Wahl der ersten Stadt hat er n Möglichkeiten. Seine zweite Stadt kann er unter den $n-1$ verbleibenden Orten auswählen. Schließlich hat er nur noch den letzten aufzusuchenden (eine Möglichkeit), und danach kehrt er nach Hause zurück. Die Anzahl der möglichen Anordnungen für n Städte ist demnach $n \times (n-1) \times (n-2) \times ... \times 3 \times 2 \times 1$. Dafür schreibt man auch $n!$ und es ergibt sich also eine Permutation der unterschiedlichen Städte. Jede dieser Anordnungen stellt eine mögliche Lösung dar, es geht nur noch darum, die optimale Lösung zu finden. Prinzipiell lassen sich Probleme dieser Art mit dem *generate-and-test* Verfahren[2] lösen. Jedoch ist diese Vorgehensweise sehr ineffizient und zeitaufwendig, denn mit wachsendem n steigt die Anzahl der Möglichkeiten $n!$ extrem schnell an. Es ist $3! = 6$, $4! = 24$, $5! = 120$, $10! = 3\,628\,800$.
Bisher ist es allerdings noch niemanden gelungen einen Algorithmus zu finden, welcher das Problem des Handelsreisenden in annehmbarer Zeit löst (vgl. [Sc95]). Das *Traveling Salesman Problem* gehört zur Klasse der NP-vollständigen Probleme, davon sind mittlerweile die meisten theoretischen Informatiker nach vielen Jahren Arbeit überzeugt (vgl. [DSW99]).
Dies bedeutet, dass die Zeitkomplexität exponentiell mit der Anzahl der Orte n steigt. Für ein TSP mit 12 Orten müsste ein Rechner schon $12! = 479\,001\,600$ mögliche Kombinationen untersuchen, um die optimale Route zu bestimmen. Angenommen, er würde zur Bewältigung dieser Aufgabe eine Zeit von fünf Minuten benötigen, so hätte er zur Lösung eines Travelling Salesman Problems mit nur 2 Orten mehr ($14! = 87\,178\,291\,200$ Möglichkeiten) schon 15 Stunden und 10 Minuten zu tun. Würden wir dann in die Route des Handlungsreisenden noch eine weitere Stadt mit aufnehmen, so beträgt die Rechenzeit schon fast zwei ganze Tage. Dieses Beispiel macht deutlich, welche Auswirkungen die hohe Komplexität des TSP hat und dass es bei Routenplanungen mit einer großen Anzahl von Orten einfach unmöglich ist, in akzeptabler Zeit (bei $n = 17$ wären es in unserem Beispiel bereits 28 Jahre Rechenzeit) alle Möglichkeiten auszuprobieren und die bestmögliche Lösung auszugeben.
Um dennoch solche recht komplexen Optimierungsprobleme wenigstens annähernd bewältigen zu können, sucht man eben nicht nach der exakten Lösung (dem globalen Optimum), sondern gibt sich mit einer fast optimalen

[2] Bei einem solchen Verfahren werden systematisch alle Lösungskandidaten nacheinander erzeugt und bewertet (vgl. [HS99]).

zufrieden. Dazu behilft man sich mit *Heuristiken*: Daumenregeln, die wenigstens grob in die Nähe der optimalen Lösungsqualität führen sollen. Zu solchen *heuristischen Suchverfahren* zählen z.B. das Bergsteigerprinzip, der Sintflut-Algorithmus oder Simuliertes Ausglühen (vgl. [DSW99],[HS99]).
Ein weiterer vielversprechender Ansatz, welcher in den letzten Jahren zunehmend an Bedeutung gewonnen hat, sind **Evolutionäre Algorithmen**. Diese orientieren sich stark am Vorbild der Natur und sind ein vereinfachtes Modell der natürlichen Evolution. Schließlich stellt das Prinzip der Evolution, welches seit Jahrmillionen unter den Lebewesen auf der Erde stattfindet, auch eine Art Optimierungsprozess dar. Dieses eignet sich erstaunlich gut für die Lösung komplexer Problemstellungen. Dazu wird auf dem Rechner ein Evolutionsprozess simuliert, welcher die grundlegenden evolutionären Operationen ausführt. Angetrieben vom Evolutionsgedanken werden sich entwickelnde Lösungsrepräsentationen erzeugt, welche im Laufe der Zeit immer besser werden und sich in Richtung einer annähernd optimalen Lösung bewegen. Der Rechenzeitaufwand ist dabei erheblich geringer als bei einer vollständigen Suche des Lösungsraumes. Für die Praxis ist die Güte dieser suboptimalen[3] Lösungen meist ausreichend und nur gering abweichend vom globalen Maximum bzw. Minimum.

1.2 Aufgabenstellung

Ziel und Inhalt dieser Arbeit ist die Entwicklung und der Einsatz eines geeigneten Verfahrens zur Optimierung der Tourenplanung eines Wachschutzunternehmens. Hierbei sind ausgehend von einem zentralen Fahrzeugdepot mehrere Kunden (Objekte) anzufahren und Kontrollgänge durchzuführen. Jeder Kunde hat für die Überwachung seines Objektes ein bestimmtes Zeitfenster definiert, innerhalb dieses Zeitraumes muss der Kontrollgang seitens des Wachpersonals erfolgen.
Das Problem ähnelt also stark dem bekannten Traveling Salesman Problem. Die Objekte sind so auf mehrere Routen (Fahrzeuge) zu verteilen, dass alle Zeitbedingungen erfüllt und die dafür benötigte Zeit und Kosten minimal werden. Probleme dieser Art nennt man auch Tourenplanungsprobleme (engl. *Vehicle Routing Problem* (*VRP*)), bei denen es darum geht, die anzufahrenden Orte auf mehrere Fahrzeuge zu verteilen.

[3] Eine suboptimale Lösung bezeichnet eine fast optimale Lösung, welche nur geringfügig schlechter ist als das globale Optimum und ist nicht zu verwechseln mit einem lokalen Optimum.

Ziel war es, eine Programmoberfläche zu erstellen, mit der der Benutzer die notwendigen Eingabedaten:

- Namen und Adressen der Wachobjekte
- Zeit- bzw. Streckenlängen zwischen den einzelnen Objekten (in Form eines vollständigen Graphen)
- Definition der Wachanweisungen
- Angabe sämtlicher Zeitbedingungen

eingeben und verwalten kann.

Mit diesen Informationen soll das Problem der Tourenplanung modelliert werden können. Darauf ist ein Optimierungsverfahren (z.B. Genetischer Algorithmus) aufzusetzen, welches möglichst optimale Vorschläge für die Tourenpläne liefert.

Das System soll anhand des aktuellen Nachtrevierplans der Safe Wachschutz/Allservice GmbH getestet und eventuelle Verbesserungsvorschläge bzgl. der Zeit- und Kosteneinsparungen untersucht werden.

1.3 Übersicht

Hier soll eine kurze Übersicht über die einzelnen Abschnitte dieser Arbeit gegeben werden.

In Kapitel 2 Theoretische Grundlagen (Seite 7) werden verschiedene Tourenplanungsprobleme beschrieben und einige Verfahren vorgestellt, welche sich grundsätzlich zur Lösung der Aufgabe eignen.

Kapitel 3 Evolutionäre Algorithmen (Seite 29) stellt kurz die Grundzüge der Evolutionstheorie und der Simulierten Evolution vor. Insbesondere wird dabei auf **Genetische Algorithmen** eingegangen, da diese Hauptströmungsrichtung der Evolutionären Algorithmen in dieser Arbeit als Optimierungsverfahren implementiert wurde.

Im Kapitel 4 Entwicklung und Implementierung (Seite 43) wird das Konzept zur Realisierung und Implementierung eines Genetischen

Algorithmus zum Lösen der Tourenplanung eines Wachschutzunternehmens beschrieben. Dabei wird insbesondere auf die Modellierung des Wachprozesses sowie die dazu verwendeten Datenstrukturen eingegangen.

Im Kapitel 5 Applikation (Seite 92) wird kurz die Programmoberfläche vorgestellt und einige Dialogfenster beschrieben.

Das Kapitel 6 Tests und Auswertung (Seite 106) befasst sich mit dem Einsatz des erstellten Programms anhand eines realen Wachrevierplans. Hier werden erste Ergebnisse präsentiert und einige aufgetretene Probleme erörtert.

Schließlich erfolgt im letzten Kapitel 7 Zusammenfassung und Ausblick (Seite 111) eine Zusammenfassung der erzielten Ergebnisse. Daraufhin werden im Ausblick weitere Einsatzmöglichkeiten und Erweiterungen der Software beschrieben.

Kapitel 2

Theoretische Grundlagen

Touren- bzw. Transportplanungen treten in der Praxis sehr häufig auf, wenn es z.B. darum geht, Güter zu transportieren, mehrere Kunden zu besuchen oder zu beliefern. Der Prozess der Tourenplanung stellt ein Optimierungsproblem dar, bei dem die anfallenden Kosten für die Abwicklung aller Aufgaben möglichst minimal gehalten werden sollen.
Durch effizientes Routing und Verteilung einzelner Prozesse auf mehrere Fahrzeuge kann viel Zeit und Geld gespart werden. An der Lösung des Problems sind eine Reihe von unterschiedlichen Gruppen interessiert: Operations Research, Mathematik, Informatik und nicht zuletzt aus wirtschaftlichen Gründen der „Verursacher“ selbst: die Logistik.

In diesem Kapitel sollen verschiedene Tourenplanungsprobleme vorgestellt und daraufhin eine Einordnung für das Problem der Tourenplanung eines Wachschutzunternehmens vorgenommen werden. Im Anschluss daran werden kurz einige Lösungsverfahren diskutiert, die sich grundsätzlich für die Lösung der gegebenen Problemstellung eignen und Aussagen bezüglich der Komplexität und Güte der einzelnen Verfahren gemacht.
Doch zunächst werden in den ersten beiden Abschnitten Grundlagen der Komplexitäts- und Graphentheorie betrachtet.

2.1 Komplexitätstheorie

Die Komplexitätstheorie ist ein wichtiger Forschungszweig der Theoretischen Informatik und befasst sich mit der Untersuchung der Komplexität von Algorithmen. Die Komplexität eines Algorithmus ist ein Maß dafür, welchen Aufwand an Ressourcen er für seine Ausführung braucht. Man unterscheidet

zwischen der Zeitkomplexität, die die benötigte Laufzeit beschreibt und der Speicherplatzkomplexität, die Aussagen über die Größe des benutzten Speichers macht. Damit die Aufwandsuntersuchungen der Algorithmen unabhängig von speziellen Computern ist, wird ein formales Berechnungsmodell zugrunde gelegt. Das wohl bekannteste Modell ist die *Turing-Maschine*[1] .

Die Komplexität eines Algorithmus wird beschrieben als Funktion $f : \mathbb{N} \to \mathbb{R}_+, n \mapsto f(n)$ über die Länge $n \in \mathbb{N}$ der Eingabe. Der Wert von $f(n)$ gibt dabei die Anzahl der Rechenoperationen an, die der Algorithmus für seine Abarbeitung benötigt. Offensichtlich steigt mit wachsendem n auch der Wert für $f(n)$ an. Zur Veranschaulichung dazu wird in Tabelle 2.1 das Wachstum einiger bekannter Funktionen verglichen.

$n \diagdown f(n)$	$\log_2 n$	$\sqrt{n}$	n	$n \cdot \log_2 n$	n^2	n^3	2^n
5	2	2	5	12	25	125	32
10	3	3	10	33	100	10^3	10^3
100	7	10	100	664	10^4	10^6	10^{30}
1000	10	32	10^3	10^4	10^6	10^9	10^{300}

Tabelle 2.1: Vergleich der Wachstumsraten einiger bekannter Funktionen

Häufig ist man nicht an dem exakten Wert einer Funktion $f : \mathbb{N} \to \mathbb{R}_+$ interessiert, sondern nur an der Größenordnung O („Groß O-Notation“). Damit hat man einen qualitativen Verlauf und kann ungefähr abschätzen, welche Laufzeit oder Speicherplatzanforderung ein Algorithmus für ein bestimmtes Problem der Eingabe n benötigt.

Einige Beispiele von Algorithmen und deren Komplexität sind[2]:

Sequentielle Suche :	$O(n)$
Binäre Suche :	$O(\log_2 n)$
Bubble Sort :	$O(n^2)$
Quick Sort :	$O(n \cdot \log_2 n)$
Matrixmultiplikation :	$O(n^3)$

[1] Der Begriff der Turing-Maschine ist im Wesentlichen von Alan Turing(1912-1954) begründet worden (vgl. [Sc95]).

[2] vgl. [Cr99]

So hat beispielsweise der bekannte *Bubble Sort* - Sortieralgorithmus eine höhere Komplexität und somit eine längere Laufzeit als der *Quick Sort* - Algorithmus, obwohl beide das gleiche Problem lösen – *das Sortieren eines Feldes.*
Man unterscheidet demnach zwischen der *Komplexität eines Algorithmus* und der *Komplexität eines Problems* (vgl. [Mu97]). Wie beim Beispiel der Sortieralgorithmen, existieren oftmals mehrere verschiedene Algorithmen zur Lösung ein- und desselben Problems, welche sich hinsichtlich ihrer Komplexität durchaus unterscheiden können. Die Komplexität des Problems ist demnach die Komplexität des bestmöglichen Algorithmus aus der Menge aller Algorithmen, welche das Problem lösen.

In diesem Zusammenhang teilt man die einzelnen Probleme in unterschiedliche Komplexitätsklassen ein. So kann man beispielsweise jedes Problem in Abhängigkeit der Laufzeit von der Größe der Eingabe n in folgende Klassen einteilen:

- man spricht von *linearer Laufzeit*, wenn sich die Laufzeit f aller Eingaben der Länge n nur bis auf einen konstanten Faktor c unterscheiden, d.h. $f(n) = c \cdot n = O(n)$
- analog dazu, spricht man bei $f(n) = c \cdot n^2 = O(n^2)$ von *quadratischer Laufzeit*
- allgemein spricht man bei $f(n) = P(n) = a_0 n^k + a_1 n^{k-1} + \ldots + a_{k-1} n + a_k = O(n^k)$, von polynomieller Laufzeit
- bei einer Laufzeit von $f(n) = 2^{P(n)} = O(2^{P(n)})$ spricht man von exponentieller Laufzeit
- usw.

Besonders wichtig sind in diesem Zusammenhang die Komplexitätsklassen P und NP. Die Klasse P ist die Menge aller Entscheidungsprobleme, die auf einer deterministischen Turing-Maschine in polynomieller Zeit gelöst werden können. Entsprechend ist die Klasse NP definiert als die Menge aller Probleme, deren Lösung auf einer nichtdeterministischen Turing-Maschine in Polynomialzeit möglich ist (vgl. [Sc95]), wobei gilt $P \subseteq NP$[3].

[3]Die Frage, ob $P = NP$ steht übrigens noch aus und ist ein zentrales Problem der Komplexitätstheorie.

Zu beachten ist, wenn zu einem gegebenen Problem kein Algorithmus existiert, welcher das Problem in Polynomialzeit löst, steigt der Zeitaufwand exponentiell an. Das bedeutet, dass bei größeren Problemen die Rechenzeit leicht von einigen Minuten oder Stunden auf bis zu einige Jahre ansteigen kann. Laufzeiten dieser Größenordnung sind einfach nicht mehr akzeptabel und man spricht hierbei von NP-harten Problemen (vgl. [Sc95]).
Darüber hinaus gibt es auch unendliche Probleme (z.B. die exakte Berechnung von π) und sogar Probleme die nicht berechenbar sind.

2.2 Grundlagen der Graphentheorie

In diesem Abschnitt sollen lediglich Grundlagen der Graphentheorie gelegt werden, so wie sie für das Verständnis der Beschreibung der einzelnen Tourenplanungsprobleme notwendig sind.

Die Geschichte der Graphentheorie beginnt mit einer Arbeit von Leonhard Euler aus dem Jahr 1736, in der er das *Königsberger Brückenproblem*[4] löste (vgl. [Kr98]).
Graphen werden dazu verwendet, um beliebige Objekte und deren Beziehungen zueinander darzustellen. Viele alltägliche Probleme lassen sich mit Hilfe eines Graphen repräsentieren, so kann beispielsweise das Straßennetz der Bundesrepublik Deutschland durch einen Graphen abgebildet werden. In diesem Fall entsprechen Orte oder Kreuzungen den Objekten und Verbindungen zwischen diesen Orten (z.B. in Form von Straßen) den Beziehungen. Grundsätzlich wird ein **Graph** $G = (K, B)$ beschrieben durch zwei endliche Mengen K (Knotenmenge) und B (Bogenmenge), dabei besteht die Menge B aus Tupeln (a, b) , mit $a, b \in K$. Die Elemente von K bzw. B heißen Knoten bzw. Bögen (Kanten) von G.

Bei der geometrischen Darstellung eines Graphen werden die Knoten als Punkte bzw. Kreise dargestellt. Die Bögen werden dann als Verbindungslinien zwischen den Knoten eingezeichnet. Abbildung 2.1 zeigt ein Beispiel für die geometrische Darstellung eines Graphen.

[4]Das Königsberger Brückenproblem bestand darin, zu entscheiden, ob es einen Rundweg durch Königsberg gibt, der jede der sieben Brücken *genau ein Mal* überquert.

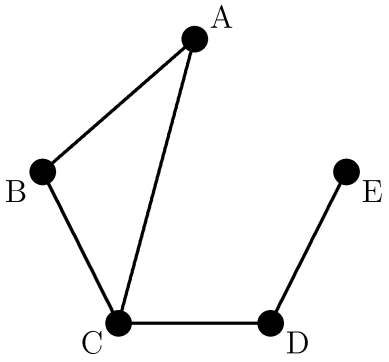


Abbildung 2.1: Geometrische Darstellung eines Graphen

Die Schreibweise dieses Graphen in arithmetischer Form lautet:

$$G = (\{A, B, C, D, E\}; \{(A, B), (A, C), (B, C), (C, D), (D, E)\}).$$

Unterscheidet man in einem Graphen die Richtung der Kanten, so handelt es sich um einen *gerichteten* Graphen. Spielt hingegen die Richtung der Kanten keine Rolle, so spricht man von *ungerichteten* Graphen (siehe Abbildung 2.2).

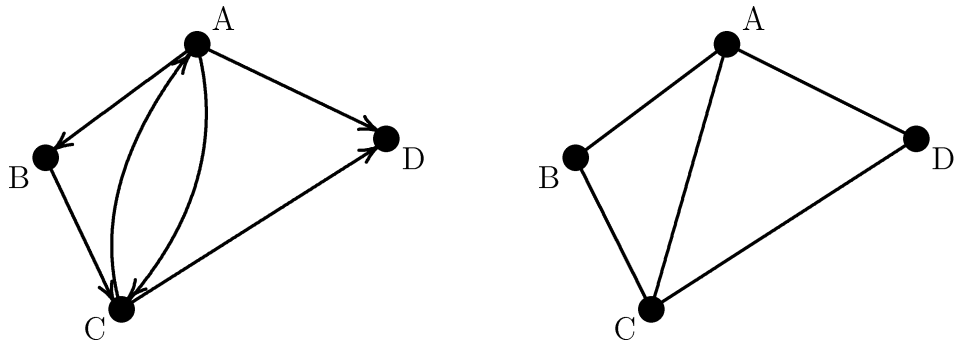


Abbildung 2.2: Darstellung eines gerichteten Graphen (links) und eines ungerichteten Graphen (rechts)

Wir gehen in den nachfolgenden Betrachtungen stets von ungerichteten Graphen aus und werden, auch im Hinblick auf die Routenplanung, einen Ansatz verfolgen, bei dem die Verbindungen zwischen den einzelnen Orten (Kunden) in Form eines ungerichteten bewerteten Graphen gespeichert werden.

Ein *bewerteter* Graph ist ein Graph, dessen Bögen reelle Zahlen zugeordnet sind. Durch diese Bewertung kann man z.B. die Kosten oder Entfernungen zwischen zwei Knoten angeben. Abbildung 2.3 zeigt ein Beispiel eines bewerteten Graphen.

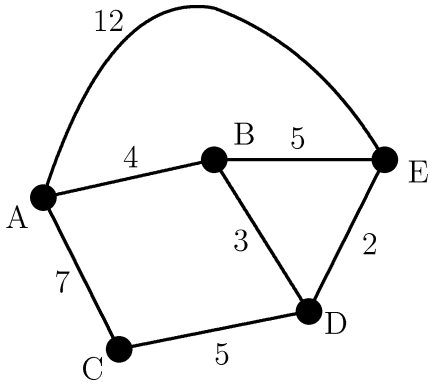


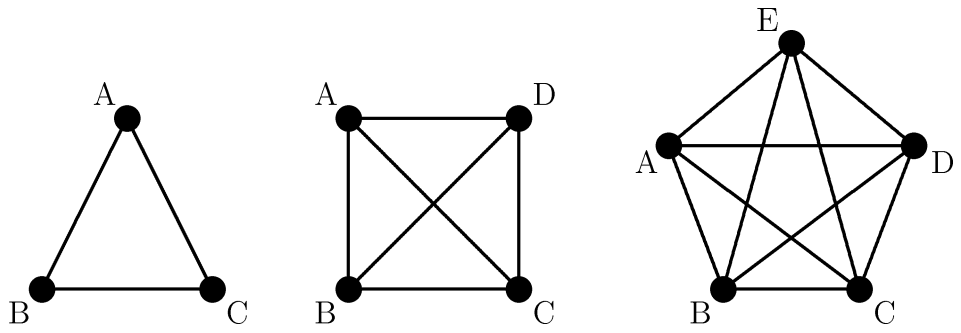
Abbildung 2.3: Bewerteter Graph mit Entfernungsangabe zwischen jeweils zwei Knoten

Als Entfernung kann neben der Distanz (in km) auch die benötigte Zeit (in Min) zwischen zwei Knoten angegeben werden. Somit besteht die Bogenmenge B in einem bewerteten Graphen aus Tripeln der Form (i, j, d_{ij}), welche jeweils den Bogen vom Knoten k_i zum Knoten k_j mit der Entfernung d_{ij} angeben.
In einem ungerichteten Graphen ist die Entfernung d_{ij} gleich der Distanz d_{ji}. Dies stellt natürlich eine Vereinfachung dar, denn wir unterscheiden damit nicht die Weglänge von Ort A nach Ort B ($\vec{AB}$) von der Strecke $\vec{BA}$. Es wird davon ausgegangen, dass die Differenz der Strecken von $\vec{AB}$ und $\vec{BA}$ vernachlässigt werden kann, also die Streckenlängen annähernd gleich sind. Damit spart man zum einen die Hälfte des benötigten Speicherplatzes und zum anderen eine Menge Zeit, wenn es darum geht einen vollständigen Graphen zwischen allen Kunden anzugeben (siehe Kapitel 4.3.2, Ansatz 2).

Ein Graph heißt *vollständig*, wenn jeder seiner n Knoten mit jedem anderen Knoten durch genau einen Bogen verbunden ist. Die Anzahl b aller Bögen in einem vollständigen ungerichteten Graphen G mit n Knoten ergibt sich zu:

$$b = \sum_{j=1}^{n-1} j = \frac{n \cdot (n-1)}{2} \quad . \tag{2.1}$$

Zur Veranschaulichung seien in Abbildung 2.4 einige vollständige Graphen mit drei, vier und fünf Knoten dargestellt.



$n = 3$ $b = 3$ | $n = 4$ $b = 6$ | $n = 5$ $b = 10$

Abbildung 2.4: Vollständige Graphen mit drei, vier und fünf Knoten

Durch Hinzufügen eines weiteren Knotens in einen bereits bestehenden vollständigen Graphen $G = (K, B)$ mit der Knotenanzahl n ($n = |K|$, Kardinalität der Menge K), müssen also n neue Bögen eingefügt werden, um wieder einen vollständigen Graphen zu erhalten. Die n Bögen entsprechen dabei den Verbindungen vom neuen Knoten k_{n+1} zu allen bisherigen n Knoten. Dieses Verständnis über vollständige Graphen ist wichtig, wenn es später darum geht einen vollständigen Graphen für ein Kundennetz aufzustellen.

Als Letztes soll nun noch beschrieben werden, was man unter einem *Weg* in einem Graphen versteht. Ein Weg vom Knoten k_i zum Knoten k_j ist eine ununterbrochene Verbindung folgender Art: $(k_i, k_r), (k_r, k_s), (k_s, k_t), \ldots, (k_n, k_j)$.

Somit stellt ein Weg eine ununterbrochene Kantenfolge zwischen zwei beliebigen Knoten dar (siehe Abbildung 2.5). Oftmals gibt es zwischen zwei Knoten mehrere Wege (mit unterschiedlichen Längen) und es ist zu entscheiden, welcher wohl der Kürzeste ist. Betrachtungen dieser Art führen zu den unterschiedlichsten Tourenplanungsproblemen, welche in den nächsten Abschnitten näher vorgestellt werden.

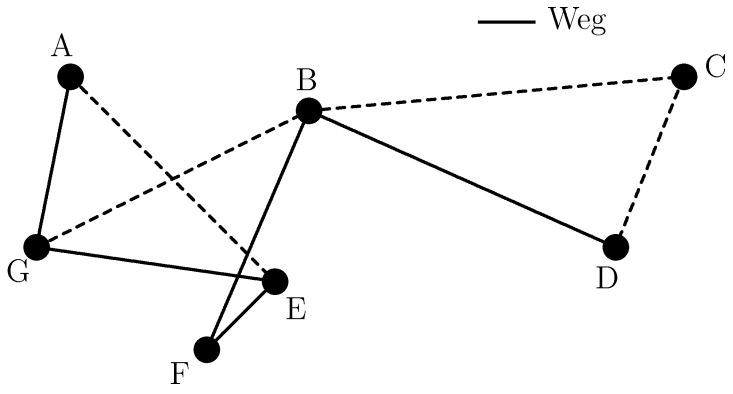


Abbildung 2.5: Weg vom Knoten A nach D

2.3 Tourenplanungsprobleme

2.3.1 Das Routenproblem

Ein typisches Beispiel für ein Suchproblem in einem Graphen ist das *Routenproblem*[5]. Gegeben ist dabei eine Karte mit Städten und Straßen, die diese Städte miteinander verbinden. Gesucht ist nun eine Route von einem gegebenen Startort zu einem Zielort (siehe Abbildung 2.6).

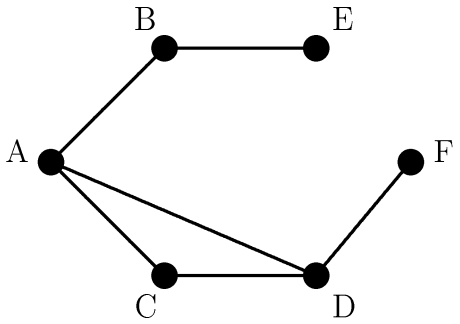


Abbildung 2.6: Das Routenproblem: Suche nach einem Weg von A nach F (aus [HS99])

[5]Zu den folgenden Ausführungen vgl. [HS99].

Dabei sind verschiedene Problemstellungen denkbar:

- existiert überhaupt ein Weg vom Startpunkt zum Zielort
- gibt es eine konkrete Route vom Start zum Ziel
- gibt es mehrere Wege, so kann man nach der optimalen Verbindung, etwa in Bezug auf die Gesamtstrecke oder Fahrzeit, fragen

Das Wissen zu diesem Problem lässt sich mit einem Graphen darstellen, dessen Knoten den Städten und dessen Kanten den Straßen entsprechen. Die Kanten des Graphen werden mit den entsprechenden Entfernungen bzw. Fahrzeiten bewertet.
Zur Lösung des Problems muss man den Graphen aus Abbildung 2.6 in einen Suchbaum transformieren und darauf ein geeignetes Suchverfahren (Tiefensuche, Breitensuche, Gleiche-Kosten-Suche, A*-Algorithmus etc.) anwenden. Ist A der Startort und F der Zielort, so ist (A, C, D, F) eine mögliche Lösung des Routenproblems. Eine kürzere Lösung in Bezug auf die Anzahl der Kanten wäre der Weg (A, D, F). Beim Erstellen des Suchbaums muss darauf geachtet werden, dass keine zyklischen Wege erzeugt werden, sonst wird der Suchbaum unendlich groß und die Suchprozedur kann leicht in Endlosschleifen geraten.

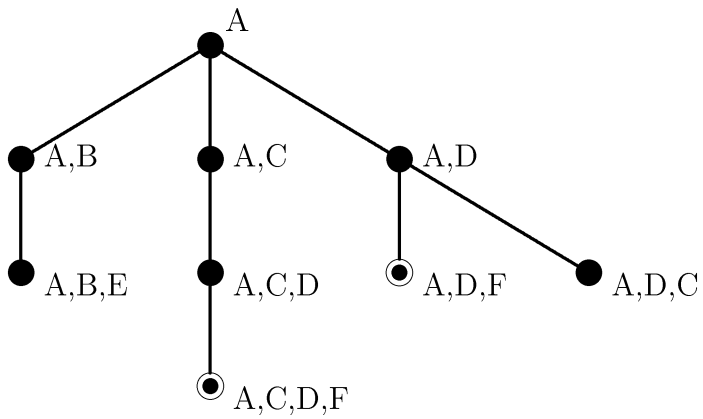


Abbildung 2.7: Transformation des Graphen in einen Suchbaum (aus [HS99])

2.3.2 Hamilton-Zyklus-Problem

Ein weiteres Problem, welches besonders in der Theoretischen Informatik oft untersucht wird, ist das *Hamilton-Zyklus-Problem*, das nach [Sc95] wie folgt formuliert werden kann:

HAMILTON-ZYKLUS-PROBLEM:

gegeben:	Ein ungerichteter Graph $G = (K, B)$.
gefragt:	Besitzt G einen Hamilton-Zyklus? (Dies ist eine Permutation π der Knotenindizes $(k_{\pi(1)}, k_{\pi(2)}, \ldots, k_{\pi(n)})$, so dass für $i = 1, 2, \ldots, n-1$ gilt: $\{k_{\pi(i)}, k_{\pi(i+1)}\} \in B$ und außerdem $\{k_{\pi(n)}, k_{\pi(1)}\} \in B$).

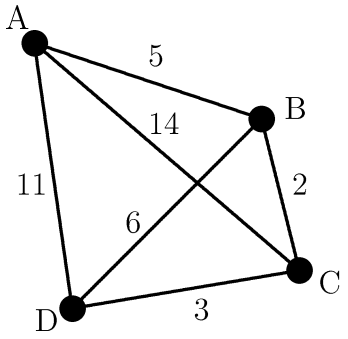
Etwas anschaulicher formuliert lautet die Frage: Hat G einen Pfad, der jeden Knoten $k \in K$ genau ein Mal besucht und dann zu seinem Ausgangspunkt zurückkehrt?
Das Hamilton-Zyklus-Problem ist NP-vollständig, dass bedeutet, es gibt derzeit keinen geeigneten Algorithmus, der das Problem in Polynomialzeit löst.

2.3.3 Traveling Salesman Problem

Wenn sich die Tourenplanung zunächst auf ein Fahrzeug beschränkt, dann handelt es sich dabei um das bekannte *Traveling-Salesman-Problem*[6] (*kurz:* TSP), welches auch als das Problem des Handelsreisenden oder Rundreiseproblem bekannt ist.
Ausgehend von seinem Heimatort hat ein Handelsreisender zur Abwicklung von Geschäften eine Rundreise durch $n-1$ Städte zu unternehmen, wobei die Reihenfolge so gewählt werden soll, dass die zurückgelegte Gesamtstrecke möglichst kurz wird.
Voraussetzung für die Lösung des Problems ist, dass die kürzesten Entfernungen zwischen den einzelnen n Städten in Form eines vollständigen Graphen bekannt sind. Es existiert also eine $n \times n$ Matrix M_{ij}, deren Elemente d_{ij} den Entfernungen zwischen einem beliebigen Knoten k_i zum Knoten k_j entsprechen. Als Vereinfachung geht man von einem ungerichteten Graphen aus, bei dem die Entfernung von k_i nach k_j (d_{ij}) gleich der Distanz d_{ji} ist.

[6] Dieses Problem ist eines der klassischen Probleme des Operations Research und ist bis heute nicht befriedigend gelöst. Seine erste exakte Formulierung stammt wohl von Karl Menger (1930). (vgl. [We72])

Die Entfernungsmatrix ist also eine symmetrische Matrix, d.h. jedes Element kann an der Hauptdiagonalen gespiegelt werden(siehe Abbildung 2.8).



$$M = \begin{pmatrix} X & 5 & 14 & 11 \\ 5 & X & 2 & 6 \\ 14 & 2 & X & 3 \\ 11 & 6 & 3 & X \end{pmatrix}$$

Abbildung 2.8: Vollständiger Graph mit der dazugehörigen Entfernungsmatrix

Problem des Handlungsreisenden:

gegeben:	Ein vollständiger Graph $G = (K, B)$ mit n Knoten sowie einer $n \times n$ Matrix M_{ij} von Entfernungen zwischen allen n Städten.
gefragt:	Finde eine Rundreise mit minimaler Gesamtlänge, die im Ausgangspunkt 0 beginnt und durch alle anderen $n-1$ Orte führt. (Dies entspricht dem Hamilton-Zyklus mit dem geringsten Abstand.)
formal:	argmin $c(\pi)$, wobei π ist Permutation der $n-1$ Städte mit der Kostenfunktion $c(\pi) = d_{0,\pi(1)} + \sum_{i=1}^{n-2} d_{\pi(i),\pi(i+1)} + d_{\pi(n-1),0}$

Die Formulierung und Beschreibung des Traveling Salesman Problems ist sehr einfach, bei der Lösung dieses Problems ergeben sich allerdings große Schwierigkeiten. Es gibt $(n-1)!$ Möglichkeiten, beginnend von einer bestimmten Stadt, alle anderen $n-1$ Städte zu besuchen und wieder zum Ausgangspunkt zurückzukehren. Der Suchraum des TSP umfasst also sämtliche Permutationen (Anordnungen) dieser $n-1$ Städte. Durch Enumeration aller möglichen Touren kann die optimale Rundreise bestimmt werden. Nun wächst die Zahl der zulässigen Touren rapide mit steigender Zahl n.

Beispiel: Wir betrachten das Traveling Salesman Problem aus Abbildung 2.8 mit $n = 4$ Städten (Start- und Zielort ist A). Dabei ergeben sich 6 mögliche Anordnungen der Lösungsmöglichkeiten:

- A,**B**,**C**,**D**,A
- A,**B**,**D**,**C**,A
- A,**C**,**B**,**D**,A
- A,**C**,**D**,**B**,A
- A,**D**,**B**,**C**,A
- A,**D**,**C**,**B**,A

Es ist offensichtlich, dass das bloße Ausprobieren aller möglichen Kombinationen kein brauchbarer Ansatz wäre. Das Problem des Handelsreisenden gehört wahrscheinlich zur Klasse der NP-vollständigen Probleme, d.h. es existiert noch kein Algorithmus, welcher das Problem in polynomialer Zeit löst. Einen Eindruck von der Komplexität des Problems lässt sich durch die Größe des Suchraumes gewinnen. Die Größe des Suchraumes eines TSP mit $n = 14$ Knoten beträgt:

$$(n-1)! = 13! = 6\,227\,020\,800$$

Probleme realer Größenordnungen sind daher über eine Enumeration z.Z. nicht in vernünftigen Rechenzeiten lösbar. Schon bei relativ kleinen Problemen mit 30 und mehr Städten würde die Rechenzeit dafür schon einige Jahre betragen.

2.3.4 Vehicle Routing Problem

Das Problem der Tourenplanung ist auch als *Vehicle Routing Problem* (*kurz:* VRP) bekannt und nimmt eine zentrale Stellung im Logistikmanagement ein. Es ähnelt stark dem bereits untersuchten Problem des Handelsreisenden und lässt sich folgendermaßen darstellen: Gegeben ist eine Menge von n Kunden, die von einem zentralen Depot aus angefahren oder beliefert werden müssen. Die Bedarfe der einzelnen Kunden sind bekannt und in Form von Nebenbedingungen formuliert, welche unbedingt erfüllt werden müssen. Dazu stehen im Depot mehrere Fahrzeuge m mit unterschiedlichen Ladekapazitäten zur Verfügung. Gesucht sind demnach

m im Depot beginnende und endende Rundreisen für die verschiedenen Fahrzeuge, welche unter Berücksichtigung der Fahrzeugkapazität den Besuch jedes Kunden sicherstellen und die Gesamtdistanz minimieren. Dabei sind sämtliche Anforderungen und Bedarfe zu erfüllen.

Das VRP lässt sich formal als 5-Tupel

$$VRP = (K, k_0, M, F, B)$$

mit

K - endliche Menge der Knoten
$k_0 \in K$ - dem Fahrzeugdepot (Start- und Zielknoten)
$M : K \times K \to \mathbb{R}$ - der Entfernungsmatrix
F - die Menge der Fahrzeuge
B - die Menge der Nebenbedingungen

beschreiben.

Der ausgezeichnete Knoten k_0 stellt das Depot dar, in dem die m Fahrzeuge stationiert sind ($m = |F|$). Alle anderen Knoten $K \setminus \{k_0\}$ stellen Städte oder Kunden dar, welche besucht werden sollen.

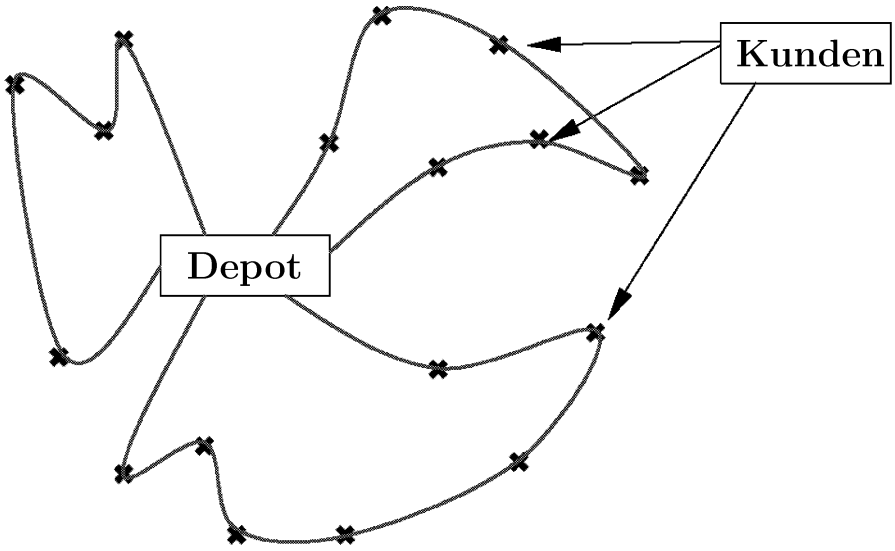


Abbildung 2.9: Darstellung des Vehicle Routing Problems

Probleme dieser Art treten in der Praxis relativ häufig auf und spielen im Rahmen von Optimierungsmodellen eine entscheidende Rolle. Dies liegt daran, dass die Prozesse der Güterverteilung in den letzten Jahren stark gestiegen und der zunehmende Wettbewerbsdruck zwischen den Unternehmen den sparsamen Gebrauch von Ressourcen erforderlich macht.
Die oben beschriebene allgemeine Variante des Vehicle Routing Problems genügt selten den tatsächlichen praktischen Gegebenheiten und muss im Hinblick auf die konkrete Problemstellung einigen Modifikationen unterworfen werden.

In der Praxis ergeben sich zusätzlich weitere Nebenbedingungen und Besonderheiten:

- Zeitfenster: Jeder Knoten k_i muss innerhalb eines bestimmten Zeitintervalls $[a_i, b_i]$ angefahren werden (man spricht bei dieser Variante auch vom *Vehicle Routing Problem with Time Windows* (*kurz:* VRPTW).

- Zeitrestriktionen: Eine einzelne Rundreise für alle oder bestimmte Fahrzeuge darf eine bestimmte Zeitdauer nicht überschreiten (z.B. maximale Lenkzeiten für Fahrer sind zu beachten).

- Längenrestriktionen: Eine einzelne Rundreise für alle oder bestimmte Fahrzeuge darf eine vorgegebene Länge nicht überschreiten.

- Aus- und/oder Rücklieferungen: Abweichend von unserem Standardproblem wäre es auch denkbar, dass die Fahrzeuge zur Gütereinsammlung zur Rücklieferung an das Depot eingesetzt werden oder eine Kombination von Aus- und Rücklieferungen vorliegt.

- Reihenfolge- und Rundreisezuordnungsrestriktionen: Bestimmte Städte dürfen immer nur in einer gemeinsamen Rundreise erscheinen und zwischen diesen Städten existiert eine Vorrangbeziehung bezüglich der zeitlichen Folge des Besuchs.

- Zuordnungszwänge zwischen Fahrzeugen und Städten: Gewisse Städte müssen stets von bestimmten Fahrzeugen oder Fahrzeugtypen angefahren werden.

- Das Problem wird dadurch komplexer, wenn mehrere Depots zu berücksichtigen sind.

Dies macht das Vehicle Routing Problem noch komplexer und unüberschaubarer als das Traveling Salesman Problem. Auch das Vehicle Routing Problem gehört zur Klasse der NP-vollständigen Probleme.

Für Probleme dieser Art lässt sich im Allgemeinen nur schwer und lediglich für eine kleine Anzahl von Kunden ein mathematisches Modell aufstellen.

> „Explizite lineare Programme für Rundreiseprobleme sind bislang für höchstens 9 Städte bekannt. ... [Eine mathematische Beschreibung dieses Problems besteht aus] ... 9 Gleichungen und 42 104 442 Ungleichungen mit 36 Variablen, entsprechend den 36 möglichen direkten Verbindungen zwischen 9 Punkten. ... Für zehn Städte sind es wahrscheinlich 51 043 900 866 Ungleichungen, mit Sicherheit nicht weniger; der Beweis für diese Vermutung steht noch aus. Für größere Rundreiseprobleme ist noch nicht einmal theoretisch ein vollständiges Sortiment von Ungleichungen bekannt... Und selbst wenn es bekannt wäre, würde es, da zu groß, nicht viel helfen.“[GP99]

Somit scheiden exakte mathematische Methoden zur Bestimmung des gesuchten globalen Optimums aus. Auch ist bis heute kein Algorithmus bekannt, welcher Probleme dieser Art in polynomieller Zeit löst. Man ist sich heute ziemlich sicher, dass es einen solchen Algorithmus, welcher globale Optima für NP-vollständige Probleme in angemessener Zeit liefert, nicht geben kann (vgl. [DSW99]).
Eine mögliche (aber sehr naive und uneffektive) Möglichkeit besteht darin, alle möglichen Lösungskandidaten zu erzeugen und zu untersuchen. Dies entspricht einem vollständigen Absuchen des gesamten Lösungsraums. Wie groß dieser schon bei einer relativ geringen Anzahl an Kunden werden kann, wurde bereits in den vorherigen Abschnitten versucht deutlich zu machen.

In der Praxis behilft man sich bei derartigen Problemstellungen, indem man eben nicht nach der exakten Lösung sucht, also der allerkürzesten Rundreise, sondern gibt sich mit einer fast optimalen Lösung zufrieden. Als Lösung bezeichnet man einen Vorschlag, der die Nebenbedingungen erfüllt und praktikabel ist. Je nach den anfallenden Kosten (Gesamtlänge oder Fahrzeit) einer Tour unterscheidet man gute und schlechte Lösungen.

2.4 Lösungsverfahren

Optimierungsverfahren müssen aus einer Menge X von Zuständen[7] einen möglichst optimalen Zustand x_o aussuchen. Man bezeichnet X auch als Lösungsraum. Auf diesem Raum ist eine reellwertige Zielfunktion $f : X \to \mathbb{R}$, $x \mapsto f(x)$ definiert, welche jedem Zustand $x \in X$ eine reelle Zahl $f(x)$ zuordnet. Diese Funktion $f(x)$ nennt man auch Zielfunktion und ist ein Maß für die Qualität von Zustand x. Bezogen auf das Problem der Tourenplanung kann z.B. $f(x)$ die Länge einer Tour x sein.
In diesem Abschnitt werden kurz einige Verfahren vorgestellt, mit denen sich das Vehicle Routing Problem grundsätzlich lösen lässt, ohne dabei den gesamten Suchraum durchzugehen. Diese Algorithmen sind im Allgemeinen relativ effizient und ermöglichen es, trotz eines extrem großen Suchraumes in angemessener Zeit zumindest näherungsweise gute Lösungen für NP-vollständige Probleme zu finden.

2.4.1 Monte-Carlo-Methode

Bei diesem Ansatz handelt es sich um ein stochastisches Verfahren, bei dem per Zufall Lösungen erzeugt und bezüglich ihrer Lösungsgüte bewertet werden. Man hofft dabei, durch den Zufall schneller eine akzeptable, näherungsweise gute Lösung ermitteln zu können, bevor der gesamte Lösungsraum systematisch durchsucht worden wäre. Dieses Verfahren zeichnet sich dadurch aus, dass es auf jede beliebige Funktion angewendet werden kann, andererseits kann es beliebig lange dauern, bis eine optimale Lösung gefunden wird.

2.4.2 Branch&Bound

Die grundsätzliche Vorgehensweise des *Branch&Bound*-Verfahrens besteht darin, das Problem als Suchbaum darzustellen und möglichst nur solche Lösungen zu untersuchen, die optimal sein können. Dazu werden ausgehend von der Wurzel des Baumes nur die Teilbäume näher betrachtet, welche zu einer optimalen Lösung führen können. Durch das Ausschließen von ganzen Teilbäumen verkleinert sich der Suchbaum erheblich und kann deutlich schneller durchsucht werden als der vollständige Suchraum.
Die Grundlage für das Abschneiden von Teilbäumen bildet eine problemspezifische Schätzfunktion. Diese dient dazu, mit geringem Rechenaufwand

[7] Ein Zustand kann eine mögliche Lösung des Problems z.B. eine Tour oder Rundreise sein.

eine untere und eine obere Schranke für die Lösungsmenge eines bestimmten Teilbaums anzugeben. Betrachtet man einen beliebigen Teilbaum, welcher beschrieben wird durch eine obere Schranke S_o und eine untere Schranke S_u, dann sind sämtliche Lösungen, die aus diesem Teilbaum hervorgehen können nicht schlechter als der Wert der oberen Schranke. Andernfalls wird es in diesem Teilbaum auch keine Lösungen geben, welche von der Lösungsqualität her besser sind als der Wert der unteren Schranke. Ist die untere Grenze größer als die beste bisher gefundene Lösung, so kann die Suche in dem betrachteten Teilbaum unterbleiben, da dort garantiert keine besseren Lösungen gefunden werden. Das gleiche gilt, wenn die untere Schranke höher ist als die obere Schranke irgendeines anderen Teilbaums.
Die Effizienz dieses Verfahrens hängt von einigen Faktoren ab, der Schätzfunktion, der Reihenfolge, in der die Teillösungen durchsucht werden und vom Aufbau des Suchbaums. Im schlechtesten Fall, wenn keine Teilbäume ausgeschlossen werden können, müssen alle Permutationen untersucht werden (dies entspricht der vollständigen Enumeration des Suchraums).

2.4.3 Darstellung als Constraintproblem

Es gibt eine Reihe von Problemen, wie z.B. das *Kartenfärbeproblem*[8] oder das *n-Damen-Problem*[9], die sich sehr effektiv lösen lassen. Man bezeichnet derartige Probleme als Zuordnungsprobleme oder auch Constraintprobleme (engl. *constraint satisfaction problem CSP*). Allgemein geht es darum, allen Variablen eine Belegung aus einer gegebenen Menge von möglichen Werten zuzuordnen, so dass gewisse Beschränkungen oder Restriktionen (auch *Constraints* genannt) erfüllt sind. Die durch die Constraints definierten Beziehungen zwischen den Variablen lassen sich durch einen sogenannten Constraintgraphen repräsentieren.
Ausgehend von einem Suchbaum, werden nur die Zweige näher untersucht, die überhaupt gültige Lösungen enthalten können. Dabei können wiederum ganze Teilbäume ausgeschlossen werden, wenn sich darin keine Lösungen mehr ergeben können, die alle Randbedingungen erfüllen.
Ziel dieses Verfahrens ist es also, den Suchbaum durch Streichen von Teilbäumen zu verkleinern und so eventuell in akzeptabler Zeit eine mögliche Lösung zu finden. Auch gibt es spezielle Heuristiken darüber, welche

[8]Beim *Kartenfärbeproblem* geht es darum, jedem Land auf einer Landkarte eine Farbe zuzuordnen, so dass keine zwei benachbarten Länder dieselbe Farbe haben.

[9]Das *n-Damen-Problem* besteht darin, n Damen so auf einem $n \times n$ - Schachbrett aufzustellen, dass keine Dame eine andere bedroht.

Variablen als nächstes belegt werden müssen, um mögliche Konflikte von vornherein zu vermeiden.

2.4.4 Das Hill-Climbing-Verfahren

Man kann sich den Lösungsraum der Touren abstrakt als eine Landschaft oder als ein Gebirge vorstellen und bei der Lösungssuche das Prinzip des Bergsteigers anwenden: „wandere im Lösungsraum umher und versuche dabei zielgerichtet immer bergauf zu gehen“.
Es werden verschiedene Hill-Climbing Ansätze unterschieden (z.B. Gauß-Seidel-Strategie, Gradientenanstieg, Simplexverfahren, ...) (vgl. [Bo01]). Alle Ansätze versuchen ausgehend von einem zufällig gewählten Startpunkt, möglichst den höchsten Gipfel im Lösungsraum zu erreichen.
Der Standort x ist der aktuelle Zustand, die Menge X der Zustände ist die Menge aller Punkte des Gebirges und $f(x)$ ist die Höhe vom Punkt x über dem Meeresspiegel (falls, wie beim TSP das Minimum von $f(x)$ gesucht ist, muss man der Höhe den Wert $-f(x)$ zuweisen). Allerdings kann man die Landschaft um sich herum nicht überschauen, sondern lediglich die nächste Umgebung abtasten. Es handelt sich offensichtlich um ein lokales Verfahren, denn es werden jeweils nur die Punkte betrachtet, die unmittelbar in der Nachbarschaft liegen.
Der Lösungsraum der Rundtouren enthält zwar sehr viele, aber doch nur endlich viele Elemente. Deswegen kann man bei der Zielfunktion von Stetigkeit, Grenzwerten und insbesondere Ableitungen gar nicht reden. Im Allgemeinen kann man jedoch wenigstens davon ausgehen, dass sich bei einer sehr kleinen Änderung an der Lösung auch der Wert der Zielfunktion nur geringfügig ändern wird[10].

Eine mögliche Strategie wäre es, in dieser Landschaft niemals bergab zu wandern, sondern stets bergauf. Dies ist allerdings ein sehr gieriger (*greedy*) und daher letzten Endes schlechter Algorithmus, da er nicht bereit ist kurzzeitig bergab zu gehen, um möglicherweise ein lokales Maximum zu verlassen. Die Wanderungen würden also in den meisten Fällen auf einem lokalen Maximum enden. In unserer abstrakten Landschaft würde dies anschaulich bedeuten, dass man ausgehend von seinem Startpunkt einfach den nächsten kleineren „Hügel“ hinaufwandert und dabei das eigentliche Ziel, den höchsten Gipfel zu finden, verfehlt. Der Grund für dieses Verhalten

[10]Würde nicht einmal das gelten, wäre blindes Durchprobieren so gut wie jedes andere Verfahren.

ist darin zu sehen, dass man von einer einmal erreichten Höhe nicht mehr heruntersteigen darf.

2.4.5 Toleranzschwellenverfahren

Das Toleranzschwellenverfahren (engl. *treshhold accepting*) (vgl. [DSW99]) modifiziert die Bergsteigerregel ein wenig, indem man eine Toleranzschwelle T hinzufügt und nun auch Schritte zulässt, die um diesen Abstand T nach unten führen. Damit wird aus der Bergsteigerregel „gehe niemals bergab“ die neue Regel „gehe niemals um mehr als T Stufen bergab“. Damit ist es möglich, von einem lokalen Maximum wieder herunter zu gelangen und ggf. höhere Gipfel zu erklimmen.
Zu Beginn des Verfahrens wählt man den Wert von T mit Bedacht relativ groß, so dass man zunächst ungehindert im Lösungsraum „umherwandern“ kann. Im Laufe des Verfahrens senkt man T langsam auf null ab und nähert sich so allmählich dem Bergsteigen an.

„Das Toleranzschwellenverfahren ist ein sehr einfaches Verfahren, das aber für viele praktisch relevante Probleme erstaunlich gute Resultate liefert“ (vgl. [HS99]). Problematisch ist nur, wie groß die Toleranzschwelle T und der Absenkungsfaktor k bestimmt werden müssen, um gute Ergebnisse zu erhalten.

2.4.6 Der Sintflut-Algorithmus

Ein weiteres einfaches und auch sehr erfolgreiches Optimierungsprinzip, welches sich ebenfalls zur Lösung von kombinatorischen Optimierungsproblemen eignet, ist der *Sintflut-Algorithmus* (vgl. [DSW99]).
Nehmen wir wiederum an, wir befinden uns als Bergsteiger in einer abstrakten Landschaft. Diesmal erzeugen wir eine Sintflut und lassen es regnen. Als wasserscheuer Wanderer laufen wir zwar im Regen umher, dürfen dabei allerdings niemals überflutetes Gelände betreten. Anstatt einer Toleranzschwelle definiert man eine Zielfunktionsschranke Wasserstand W, die in Folge der Sintflut langsam ansteigt. Der Sintflut-Algorithmus akzeptiert jede (beliebig schlechte) Lösung x, sofern der Zielfunktionswert $f(x)$ nur höher liegt als der aktuelle Wasserstand W. Jedes Mal, wenn dieser Algorithmus eine neue Lösung akzeptiert hat, wird im Programm die Variable Wasserstand W ein wenig erhöht.

2.4.7 Simulated Annealing

Beim simulierten Ausglühen (engl. *Simulated Annealing*) (vgl. [HS99]) wird ein thermodynamischer Prozess aus der Natur nachgebildet. Hierzu wird ein Festkörper erhitzt und anschließend extrem langsam abgekühlt. In einem heißen Material können sich die Atome im Kristallgitter relativ frei bewegen, während mit zunehmender Abkühlung diese Bewegungsfreiheit der Atome abnimmt. Dieses Verhalten nutzt man besonders in der Metall- und Glasverarbeitung aus, um sprödes Verhalten zu vermeiden.
Überträgt man diese Idee auf das Problem des Bergsteigens, so bedeutet dies, dass der Bergsteiger zu Beginn des Verfahrens eine große Bewegungsfreiheit hat, um eventuell auch bergab zu laufen. Im weiteren Verlauf nimmt diese Bewegungsfreiheit ab und die Wahrscheinlichkeit bergab zu gehen wird immer geringer.

2.4.8 Evolutionäre Algorithmen

Die Idee, die sich hinter diesem Verfahren verbirgt, entspricht dem Prinzip der *Evolution* und stammt eigentlich aus der Natur. Man kann das Problem der Anpassung der unterschiedlichen Lebewesen an ihre Umwelt als Optimierungsproblem verstehen. Im Verlauf der Evolution hat sich auf unserem Planeten durch Vorgänge wie Vererbung, Mutation und Selektion eine Vielzahl von Arten entwickelt, welche sich speziell an ihren jeweiligen Lebensraum angepasst haben.
Evolutionäre Verfahren haben also als Ziel, die grundlegenden Prinzipien der Evolution (zumindest stark vereinfacht) nachzuahmen. Dabei codiert man mögliche Lösungskandidaten (Individuen) als Chromosom und wendet darauf genetische Operatoren an.
Ausgehend von einer zufällig erzeugten Anfangspopulation[11] entwickeln sich im Laufe von mehreren Iterationsschritten (Generationen) durch Anwenden der Simulierten Evolution nach und nach bessere Lösungsvorschläge. Die Effizienz dieses Verfahrens wird deutlich von äußeren Parametern und der Wahl einer geeigneten Fitnessfunktion (welche jedem einzelnen Individuum einen Wert bezüglich der Lösungsgüte zuordnet) geprägt.

[11] Eine Population bezeichnet eine Menge von Individuen.

2.4.9 Zusammenfassung

Aufgrund der großen praktischen Bedeutung der kombinatorischen Optimierungsprobleme wurden in den letzten Jahren eine Reihe von Lösungsmöglichkeiten entwickelt und vorgeschlagen. Die hier vorgestellten Verfahren würden sich grundsätzlich zur Lösung des Vehicle Routing Problems eignen. Jeder Ansatz hat seine Berechtigung und wird in der Praxis erfolgreich eingesetzt. In der Tabelle 2.2 sind die vorgestellten Verfahren nach verschiedenen Kriterien bewertet. Anhand dieser Kriterien erfolgt die Auswahl eines geeigneten Lösungsverfahrens.

Aufwand der Implementierung

Dies ist eine Abschätzung, wie komplex in etwa eine programmtechnische Umsetzung (Datenstrukturen, Algorithmen etc.) für das entsprechende Optimierungsverfahren werden kann.

Speicherplatzbedarf

Der Speicherplatzbedarf eines Suchverfahrens gibt an, wie groß die Anforderungen an den Speicherplatz sind (z.B. für Verwendung einer Agenda etc.).

Anytime-Lösung

Hier wird angegegeben, ob zu jedem Zeitpunkt der Optimierung eine mögliche Lösung vorliegt oder nicht. Dies hat den Vorteil, das man jederzeit eine Lösung zur Verfügung hat und diese ggf. verwenden kann, bis eine bessere gefunden wurde.

Anforderungen an den Benutzer

Hier wird angegeben, wie hoch die Anforderungen an den Anwender zur Bedienung des Optimierungsverfahrens sind. Kann er eventuelle Parametereinstellungen selbstständig vornehmen? Dabei fließt natürlich auch mit ein, ob das Prinzip des Optimierungsverfahrens leicht oder schwer verständlich zu machen ist.

persöhnliche Bewertung

Dieses Kriterium gibt an, wie vielversprechend das Verfahren im Hinblick auf die gestellte Problemstellung ist.

	Aufwand	Speicher-bedarf	Anytime-Lösung	Anf. an Benutzer	Bewertung
Monte-Carlo	gering	niedrig	ja	gering	$--$
B&B	sehr hoch	hoch	nein	gering	$+/-$
CSP	sehr hoch	hoch	nein	gering	$+/-$
Hill-Climbing (*Gauß-Seidel*)	gering	niedrig	ja	gering	$-$
TA	gering	niedrig	ja	mittel	$+$
Sintflut	gering	niedrig	ja	mittel	$+$
SA	gering	niedrig	ja	mittel	$+$
EA	mittel	mittel	ja	mittel	$++$

Tabelle 2.2: Vergleich der unterschiedlichen Verfahren

Aus der Tabelle 2.2 geht hervor, dass besonders die heuristischen Suchverfahren, wie Toleranzschwellenverfahren, der Sintflut-Algorithmus, Simuliertes Ausglühen und evolutionäre Algorithmen, für das Problem der Tourenplanung geeignet erscheinen.
Letztere haben besonders in den vergangenen Jahren große Aufmerksamkeit auf sich gezogen, indem sie in einer Vielzahl von industriellen Problemen erfolgreich Anwendung gefunden haben. Auch im Rahmen dieser Diplomarbeit sollen evolutionäre Algorithmen Anwendung finden und ihr Einsatz bei der Optimierung der Tourenplanung eines Wachschutzunternehmens näher untersucht und beschrieben werden.

Kapitel 3

Evolutionäre Algorithmen

In den letzten Jahren haben im Bereich der praktischen Optimierung besonders Methoden beträchtlich an Bedeutung gewonnen, die man als naturanalog bezeichnen kann, weil sie sich an Vorbildern der Natur orientieren. Zu den bekanntesten Entwicklungen zählen zweifelsohne die Evolutionären Algorithmen. Evolutionäre Algorithmen lösen Optimierungsprobleme nicht auf herkömmlichen Wege, sondern nach dem Vorbild der biologischen Evolution und der molekularen Genetik. Man hat erkannt, dass sich die Mechanismen der Evolution gut für die Lösung von komplexen Problemstellungen eignen, bei denen konventionelle Methoden scheitern würden.
So haben EA durch ihre völlig neuen Lösungsmöglichkeiten bei praktischen Anwendungen in den letzten Jahren große Aufmerksamkeit erregt. Viele Unternehmen haben das Potential dieser naturanalogen Methoden erkannt und ihrerseits Forschungsgruppen zu diesem Thema ins Leben gerufen.
Viele erfolgreiche Anwendungen von Evolutionären Algorithmen finden sich in den verschiedensten Gebieten, wie der Strukturoptimierung, beim Chip-Entwurf, bei der Erstellung eines Maschinenbelegungsplans oder der Steuerung eines Roboters.

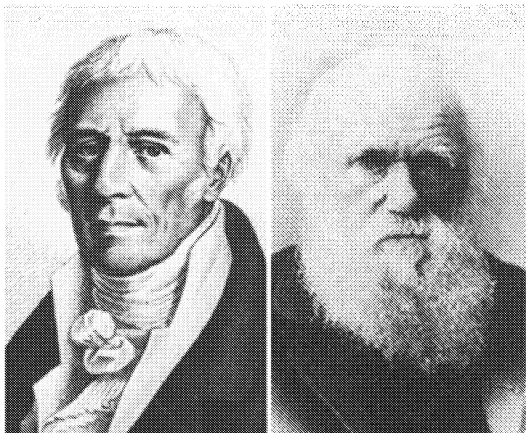
3.1 Grundelemente der Evolutionstheorie

Bis zum Ende des 18. Jahrhunderts galt die allgemeine Lehrmeinung von der Unveränderlichkeit der Arten (vgl. [BK89],[SHF94]). Bis dahin waren die meisten Menschen der Ansicht, dass die Arten, so wie wir sie jetzt vorfinden, seit Beginn der Welt vorhanden waren. Dieses Bild wurde auch stark von der Kirche geprägt, spiegelt es doch die biblische Schöpfungsgeschichte wieder. Außerdem ist die individuelle Lebenszeit viel zu kurz, um überhaupt eine

Veränderung der Arten wahrnehmen zu können. Einzig der Fund der Fossilien und Überreste bezeugen, dass es andere Arten gegeben haben muss, jedoch wusste man lange Zeit nichts mit diesen Funden anzufangen.
Bereits Leonardo da Vinci (1452-1519) erahnte die eigentliche Bedeutung der Fossilienfunde. Der Zoologe Georges Baron de Cuvier (1769-1832) war Begründer der Paläontologie als Lehre von den Lebewesen der Vorzeit. Er verglich den Aufbau der unterschiedlichen fossilen Lebensformen mit den heute lebenden Tierarten und fand heraus, dass zu bestimmten erdgeschichtlichen Epochen verschiedene Formen von Tieren gelebt haben. Aus seinen Erkenntnissen leitete er seine *Katastrophentheorie* ab, in der er Naturkatastrophen für das Aussterben der Arten verantwortlich machte. Nach jeder Katastrophe bildeten sich dann wieder völlig neue Lebensformen heraus. Die Theorie Cuviers wurde durch die moderne Geologie widerlegt.
Der französische Zoologe Jean-Baptiste de Lamarck vertrat 1809 als erster die Auffassung, dass die Lebewesen der Jetztzeit von ausgestorbenen abstammen. Lamarck nahm an, dass Eigenschaften, die ein Lebewesen während seines Lebens erwirbt, auf Nachkommen vererbt werden können. Wenn z.B. Giraffen durch dauerndes Strecken ihrer Hälse im Laufe ihres Lebens etwas längere Hälse bekämen, so könnten sie ihren Nachkommen diese erworbene Eigenschaft vererben. Im Laufe der Zeit könnten sich so aus kurzhalsigen Urgiraffen durch stetigen Gebrauch der Hälse beim Nahrungserwerb langhalsige Giraffen entwickelt haben. Andererseits würde der Nichtgebrauch von Organen zu deren Verkümmerung führen. So erklärte Lamarck die Rückbildung von Organen, z.B. die Beinlosigkeit der Schlangen.
Lamarcks Theorie fand in seiner Zeit keine Anerkennung und auch die moderne Genetik widerlegt seine Ansicht. Seine Leistung muss dennoch gewürdigt werden. Er war einer der ersten, der mit seiner Theorie die Ansicht von der Konstanz der Arten zu überwinden versuchte.
Erst Charles Darwin verhalf mit seiner Theorie, die in ihren Grundzügen auch heute noch gültig ist, der Abstammungslehre zur allgemeinen Anerkennung. Nach dieser Lehre hat sich ausgehend von einfachsten Grundformen im Laufe von Jahrmillionen auf der Erde eine unüberschaubare Vielfalt an Lebewesen entwickelt. Der dabei zugrundeliegende Mechanismus der Evolution wurde von Darwin in seinem bedeutenden Werk „On the Origin of Species by Means of Natural Selection“ (Über den Ursprung der Arten), das im Jahre 1859 erschien, beschrieben.
Demnach kann man sich die Evolution als einen ständig fortschreitenden Anpassungs- und Optimierungsprozess in einer sich verändernden Umwelt vorstellen. Lebewesen, die aufgrund bestimmter Eigenschaften besser an ihre

Umgebung angepasst sind, haben eine größere Chance Nachkommen zu erzeugen, als diejenigen, die aufgrund ihrer Ausprägungen nicht so gut mit den gegebenen Umweltbedingungen zurechtkommen. Dieses Prinzip nennt man auch „survival of the fittest“. Nur die Arten, welche sich am besten an die Randbedingungen des Lebensraumes anpassen, werden überleben.

J.B. de Lamarck (1744-1829)



Charles Darwin (1809-1882)

Abbildung 3.1: Jean-Baptiste de Lamarck und Charles Darwin, zwei Pioniere der Evolutionsforschung

Die Komplexität und Vielfalt der Lebensformen auf unserem Planeten kann auf wenige Operatoren zurückgeführt werden, die in Populationen wirksam werden. Im Mittelpunkt steht die Fortpflanzung von Individuen. Damit verbunden ist die Weitergabe von Erbinformationen. Durch Faktoren wie Mutation (stochastische Abweichung) und Rekombination kommt es zur Veränderung und Vermischung des Erbguts.
Auf diese Weise entstehen Nachkommen mit unterschiedlichen Eigenschaften, die ihrerseits konkurrieren im Wettkampf um Überleben und Fortpflanzung. Dabei besitzt jedes Individuum eine unterschiedliche Fitness (Tauglichkeit). Im Rahmen der natürlichen Auslese (Selektion) setzen sich im Allgemeinen die Individuen gegen ihre Konkurrenten durch, die den gegebenen Umweltbedingungen besser angepasst sind und geben wiederum die Erbinformation weiter.
Dieses Wechselspiel aus Variation (Mutation, Rekombination) und Selektion stellt einen stufenweisen Optimierungsprozess dar, welcher seit Jahrmillionen unter den Lebewesen auf der Erde stattfindet. Dieser Prozess der Optimierung hat bisher viele facettenreiche Arten hervorgebracht, die sich ihren Umweltbedingungen hervorragend angepasst haben, so z.B. die unterschiedlichsten Arten von Insekten, Fischen oder Säugetieren (darunter

auch den Menschen). Aufgrund der mangelnden Anpassungsfähigkeit sind im Zuge der Evolution auch zahlreiche Arten, wie z.B. die Dinosaurier, ausgestorben.
Wie der Evolutionsprozess stattgefunden hat, lässt sich zwar nicht lückenlos beweisen, dennoch besteht an der Tatsache der Evolution heute kein Zweifel mehr. So ist DARWINS Theorie noch heute gültig und zumindest in ihren Grundzügen allgemein bekannt.

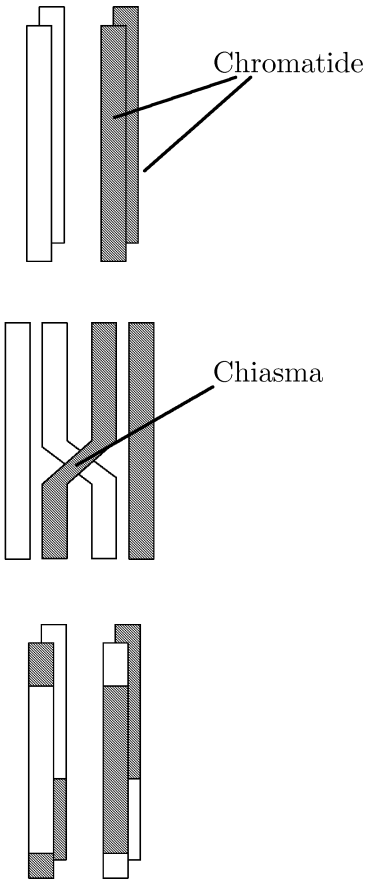
In den nächsten Abschnitten sollen nur kurz einige wichtige biologische Grundbegriffe erläutert werden, die zu einem besseren Verständnis der späteren programmtechnischen Umsetzung vonnöten sind.

3.1.1 Unterscheidung von Genotyp und Phänotyp

Man unterscheidet zwischen den im genetischen Code niedergelegten Erbinformationen (Genotyp) und dem Erscheinungsbild (Phänotyp) von Lebewesen. Die anatomischen Ausprägungen (Eigenschaften) der Individuen ist das Produkt der Umsetzung des genetischen Codes, der im Zellkern einer jeden lebenden Zelle in Form von Chromosomen vorhanden ist.
Unter einem Gen versteht man einen Abschnitt von *Desoxyribonucleinsäure* (DNS). Die Gesamtheit der Gene, das sog. Genom, ist dabei für die Ausbildung eines bestimmten Individuums mit seinen individuellen Eigenschaften (Phänotyp) verantwortlich.

3.1.2 Die Modifikation der genetischen Information

Es existieren grundsätzlich zwei Mechanismen, die für eine Veränderung des Erbguts sorgen und es somit ermöglichen, dass die Nachkommen andere Eigenschaften aufweisen als ihre Eltern. Dies ist zum einen die *Rekombination* und die *Mutation* des Erbmaterials.
Die Rekombination des Erbmaterials tritt generell bei der geschlechtlichen Fortpflanzung bei Organismen auf. Jeweils gleichartige Chromosomen der Eltern werden zu neuen Chromosomen kombiniert. Dabei ist das Kombinieren nur an bestimmten Stellen des Chromosoms, nämlich den Grenzen der Gene, möglich. Hier werden die Chromosomenstücke durch ein Crossover ausgetauscht (siehe Abbildung 3.2). Dabei findet eine Durchmischung des genetischen Materials beider Elternteile statt und führt dazu, dass die Nachkommen unterschiedliche Eigenschaften der beiden Elternteile besitzen.



Ausgangspunkt der Rekombination sind zwei gleiche (homologe) Chromosomen mit je zwei Chromatiden.
Die Chromatiden der Eltern lagern sich während der sogenannten *Meisose* eng aneinander. Chromosom1 (hell) stamme von Elter 1 und Chromosom 2 (dunkel) stammt von Elter 2.

Während der anschließenden Paarung der Chromosomen erfolgen Brüche der DNS-Stränge, die oft über Kreuz verheilen. Die Kreuzungen werden als Chiasma bezeichnet. Je nach Länge eines Chromosoms sind 1-8 Chiasmata beobachtet worden. Das Vertauschen der Chromosomenstücke wird als *Crossover* bezeichnet.

Nach vollzogener Paarung tragen die neu entstandenen Chromosomen Eigenschaften von beiden Elternteilen.

Abbildung 3.2: Rekombination durch Crossover während der Reifeteilung (Meiose) (aus [St99])

Als Mutation bezeichnet man zufällige Veränderungen im Erbmaterial. Durch diese Veränderung entstehen gelegentlich phänotypisch wirksame Varianten des Grundtyps einer Art, d.h. Individuen mit neuen Eigenschaften. Auslöser für Mutationen sind z.B. chemische Stoffe, ionisierte oder energiereiche Strahlung.

Mutationen können in unterschiedlicher Form auftreten. Als Genmutationen betreffen sie nur einzelne Gene. Genmutationen sind die „Materiallieferanten“ der Evolution. Bereits kleine Veränderungen können zu starken Auswirkungen im Phänotyp führen. Bei der Chromosomenmutation wird die Chromosomenstruktur beeinflusst, hierbei sind kleinere oder größere Stücke der DNS betroffen.

Genommutationen verändern die Anzahl einzelner Chromosomen oder ganzer Chromosomensätze. Beim Menschen führt dies in der Regel zu sehr großen Anomalien bei den entstehenden Individuen.

3.1.3 Evolutionsfaktor Selektion

Der bedeutendste Evolutionsfaktor ist neben der Rekombination und Mutation die *Selektion*. Wesentlich ist die Vorstellung eines Überschusses an Nachkommen. Abweichungen in den Eigenschaften der Nachkommen führen zu unterschiedlicher Fitness dieser Individuen im Überlebenskampf. Im Selektionsprozess haben diejenigen Lebewesen, die aufgrund ihrer Eigenschaften den Umweltbedingungen besser angepasst sind, eine höhere Chance Nachkommen zu erzeugen und so ihre Erbanlagen weiterzugeben. Diesen Prozess nennt man auch *natürliche Auslese*.

3.2 Simulierte Evolution

Man kann sich den Evolutionsprozess als einen seit Jahrmilliarden andauernden Optimierungsvorgang in einer sich verändernden Umwelt vorstellen. Bereits in den späten 50er und frühen 60er Jahren versuchten einige Wissenschaftler den Evolutionsgedanken aufzugreifen und für praktische Zwecke einzusetzen. Dabei wurden die Grundmechanismen der Evolution abstrahiert und auf einer sehr abstrakten Ebene nach dem Vorbild des natürlichen Optimierungsprozesses simuliert. So entstanden eine Reihe von Pionierarbeiten, die zu den frühen Vorläufern heutiger Evolutionärer Algorithmen zählen. Man unterscheidet heute vier verschiedene Hauptströmungen von EA:

- Evolutionsstrategien
- Evolutionäre Programmierung
- Genetische Programmierung
- Genetische Algorithmen

Die Entwicklung der *Evolutionsstrategien* (ES) geht zurück auf Arbeiten von Ingo Rechenberg und Hans-Paul Schwefel an der TU Berlin Mitte der 60er Jahre. Sie konnten damals das Problem der Optimierung des Windwiderstandes eines Stromlinienkörpers nicht auf mathematischem

Wege lösen und kamen auf den Gedanken, durch Nachahmung evolutionärer Prinzipien, eine Optimierungsmethode zu entwickeln.

Evolutionäre Programmierung (EP) geht zurück auf Arbeiten von Lawrence J. Fogel, Alvin J. Owens und Michael J. Walsh Mitte der 60er Jahre. Das damalige Ziel bestand darin, nach einer Möglichkeit zu suchen, künstlich intelligente Automaten zu entwerfen. Diese sollten in der Lage sein, Probleme auf innovative Weise zu lösen. Gleichzeitig erhoffte man sich von den Experimenten ein besseres Verständnis des Phänomens „Intelligenz“ und der Organisation des menschlichen Verstandes. Damit standen Fogel, Owens und Walsh mit ihren Zielen dem heutigen Artificial Life näher als dem Bearbeiten von Optimierungsaufgaben.

Die *Genetische Programmierung* (GP) wurde maßgeblich von John R. Koza in den 90er Jahren geprägt und in seinem Buch [Ko93] beschrieben. Die Grundidee besteht darin, Problemlösungen in Form von Computerprogrammen automatisch, durch den Einsatz evolutionärer Prinzipien, zu erzeugen.

Genetische Algorithmen (GA) gehen auf Arbeiten von John Holland in den 60er Jahren zurück. Holland wollte vor allem die Mechanismen adaptiver Systeme erklären und in Form sogenannter *reproductive plans* (erst später als GA bezeichnet) auf Computern implementieren. Dabei nahm er sich die biologische Evolution zum Vorbild. Diese Ideen wurden bald für Optimierungszwecke eingesetzt.

GA sind heute die in Forschung und Anwendung zahlenmäßig dominierende Hauptströmung der Evolutionären Algorithmen. Auch im Rahmen dieser Diplomarbeit wurde auf Grundlage eines Genetischen Algorithmus ein Optimierungsverfahren entwickelt, welches die Tourenplanung eines Wachunternehmens übernimmt und möglichst optimale Lösungen generiert. Deshalb ist eine genauere Beschreibung genetischer Algorithmen Gegenstand des nächsten Abschnittes. Eine detaillierte Darstellung der drei anderen Hauptformen findet sich übrigens in [Ni97].

Alle unterschiedlichen EA-Formen laufen nach einem allgemeinen Ablaufschema ab. Ausgehend von einer stochastisch generierten Startmenge (Anfangspopulation) von einzelnen Lösungsmöglichkeiten (Individuen) werden mehrere Iterationen (Generationen) durchlaufen, wobei jeweils aus den alten Lösungen neue, modifizierte Lösungsvorschläge erzeugt werden.

In jeder Generation werden Individuen aus der aktuellen Population durch Selektion ausgewählt. Dies erfolgt in Abhängigkeit ihres Fitnesswertes (Funktion die es zu optimieren gilt). Die Fitness eines Individuums ergibt sich aus der Kombination seiner Eigenschaften und ist somit ein Maß für die Güte der Lösung.
Diese Individuen erzeugen Nachkommen, indem die jeweilige Lösungscodierung kopiert (Replikation) oder variiert (z.B. durch Mutation oder Crossover) wird. Somit ergeben sich Nachkommen, die ihrerseits bewertet werden und ggf. in die neue Population aufgenommen werden.

Dieses Wechselspiel aus Selektion (Bevorzugung der besten Lösungen) und Variation (ungerichtete Veränderung von Lösungen) führt im Laufe vieler Generationen zu sukzessiv besseren Lösungsvorschlägen. Der Evolutionsprozess wird solange fortgesetzt, bis ein Abbruchkriterium erreicht worden ist, dies kann z.B. die maximale Anzahl von Generationen $g_{\max}$ sein. Abschließend wird dem Benutzer die beste gefundene Lösung mitgeteilt und das Verfahren terminiert. Allerdings garantiert dieses Verfahren nicht, dass das globale Optimum gefunden wird.

3.3 Genetische Algorithmen

In diesem Abschnitt wird zunächst ganz allgemein das GA-Grundkonzept vorgestellt, welches dann im 4. Kapitel erweitert und auf das Vehicle Routing Problem angepasst wird.
Bei einem *Genetischen Algorithmus* (*kurz*: GA) handelt es sich um ein heuristisches Suchverfahren, welches besonders für Problemklassen eingesetzt werden kann, bei denen konventionelle Lösungsmethoden zu keinem Ergebnis führen.
Ein GA verwendet Strategien aus der Evolutionstheorie, um zu einem gegebenen Problem eine möglichst gute Lösung zu finden. Die Lösungsrepräsentation eines Problems muss dabei so formuliert werden, dass sie als *Chromosom*, bestehend aus einer Folge oder Menge von elementaren Bausteinen (*Genen*), aufgebaut ist.
Man betrachtet den *Merkmalsraum* $M^n = \{0,1\}^n$ $(n \geq 1)$, dann lässt sich ein *Chromosom* $c \in M^n$ als Vektor der Form $c = \langle c_1, c_2, \ldots, c_n \rangle$ $(c_i \in \{0,1\})$ darstellen. Ein Chromosom c mit einer beliebigen Ausprägung (z.B. $\langle 0,1,0,0,1,1 \rangle \in M^6$) der einzelnen Gene bezeichnet man als *Individuum*. Sämtliche Eigenschaften des Individuums (Phänotyp) werden durch die

Genotypdarstellung (Chromosom) codiert. Zur Veranschaulichung ist in Abbildung 3.3 schematisch ein Beispiel für eine binäre Lösungsrepräsentation, in Form eines Chromosomen-Strings, dargestellt. Es ist üblich, die einzelnen Bits auf dem String als Gene und ihre konkrete Ausprägung (0 oder 1) als *Allel* zu bezeichnen.

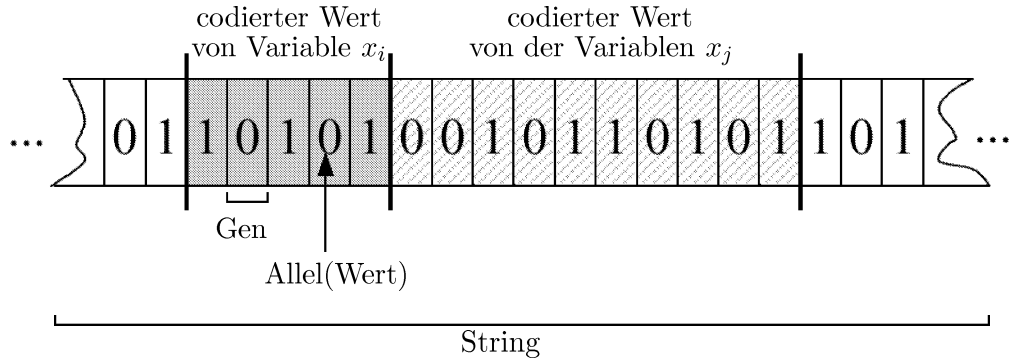


Abbildung 3.3: Binäre Lösungsrepräsentation eines Chromosoms (aus [Ni97])

Eine Variable x_i wird durch eine Bitfolge (Segment) codiert. Die Länge dieser Bitfolge ergibt sich aus dem Wertebereich der jeweiligen Variablen. Mit der Binärcodierung kann man generell sämtliche verschiedene Variablentypen darstellen. In vielen praktischen Anwendungen ist es oftmals sinnvoll, von der binären Codierungsform abzuweichen und zu anderen Lösungsrepräsentationen überzugehen, die die strukturellen Eigenheiten der jeweiligen Problemstellung berücksichtigen[1]. Mit Hilfe einer segmentweise vorgehenden Decodierungsfunktion können aus dem binären Chromosom die einzelnen Variablenwerte gewonnen werden. Jedes Chromosom c stellt also eine mögliche Lösung des Problems (Punkt im Suchraum) dar. Eine Menge $P \subseteq M^n$ solcher Individuen heißt *Population*. Der allgemeine Ablaufplan eines Genetischen Algorithmus ist in Abbildung 3.4 dargestellt.
Ausgehend von einer zufällig generierten Anfangspopulation P_0 erzeugt man durch genetische Operatoren (Mutation, Crossover und Reproduktion) Nachkommen der vorhandenen Individuen. Es entstehen neue Generationen von Individuen, welche nach Möglichkeit besser bezüglich der zu optimierenden Funktion (auch Zielfunktion oder *Fitnessfunktion* ϕ genannt) sind.

[1] Auf unterschiedliche Darstellungsformen der Chromosomen wird im Kapitel 4 näher eingegangen und speziell für das Vehicle Routing Problem eine mögliche Lösungsrepräsentation angegeben.

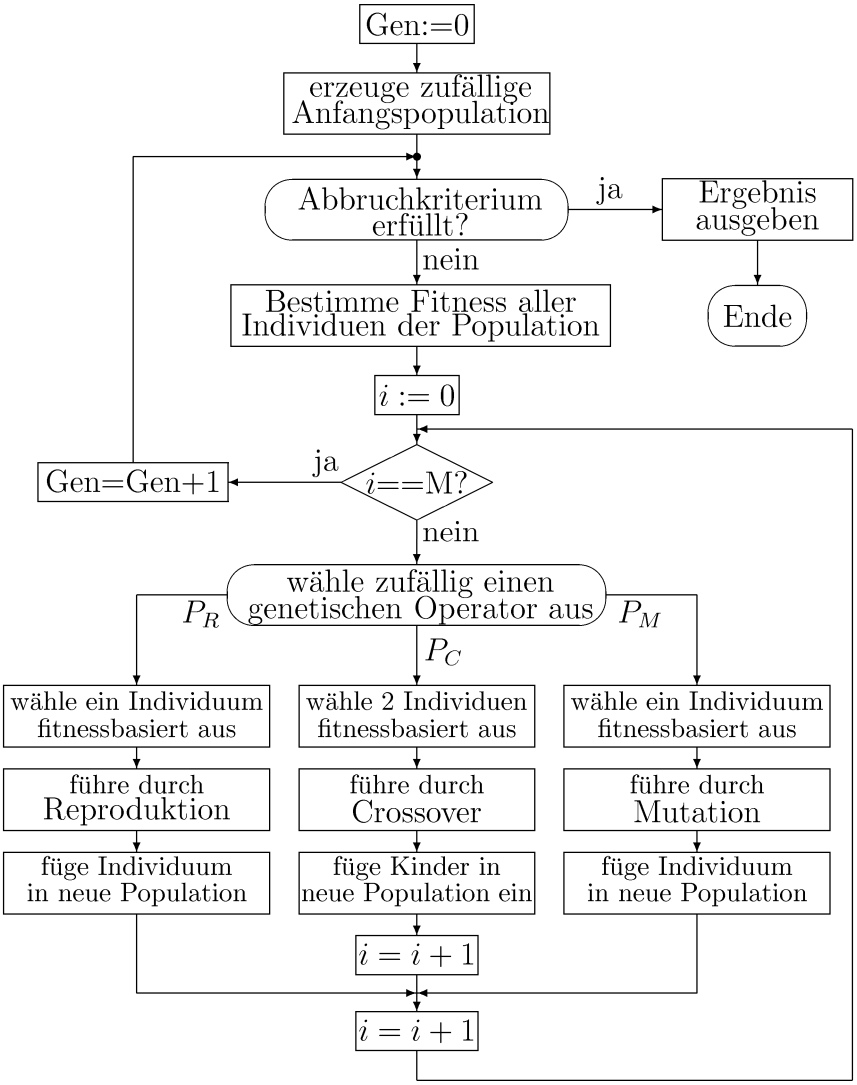


Abbildung 3.4: Programmablaufplan eines GA (aus [Bo01])

Durch diese *Fitnessfunktion* $\phi : M^n \to \mathbb{R}, c \mapsto \phi(c)$ wird jedem Individuum c ein Maß, bezüglich der Lösungsgüte, zugewiesen. Dieses Fitnessmaß hat eine entscheidende Bedeutung bei der Auswahl der Individuen für die verschiedenen Vererbungsmechanismen. So haben diejenigen Individuen mit einer guten Fitness eine höhere Wahrscheinlichkeit zu überleben (Reproduktion) und ihre Erbinformationen durch Mutation und Crossover (Rekombination) weiterzugeben. Der Evolutionsprozess wird solange fortgesetzt, bis ein Abbruchkriterium erreicht worden ist. Dies kann entweder der Ablauf einer maximalen Anzahl an Generationen $g_{\max}$ sein oder aber das Erreichen eines definierten Zielfunktionswertes.

Im den folgenden Abschnitten sollen kurz die grundlegenden genetischen Operatoren und Mechanismen vorgestellt werden.

3.3.1 Reproduktion

Ein ausgewähltes Individuum wird unverändert in die nächste Population übernommen. Der Grundgedanke, der dahinter steckt ist, dass Individuen mit guter Fitness in der Population überleben und in der nächsten Generation erhalten bleiben. Dabei legt eine Reproduktionsrate P_R fest, ob das ausgewählte Individuum reproduziert werden soll oder nicht.

3.3.2 Mutation

Die Mutation entspricht einer zufälligen Veränderung des Erbgutes. Dazu werden ein oder mehrere Gene mit einer Mutationsrate P_M geflippt, d.h. aus einer 0 wird eine 1 und umgekehrt (siehe Abbildung 3.5). Dadurch erhält man Divergenz und Inhomogenität in der Population und verhindert vorzeitige Konvergenz (*premature convergence*) des GA.

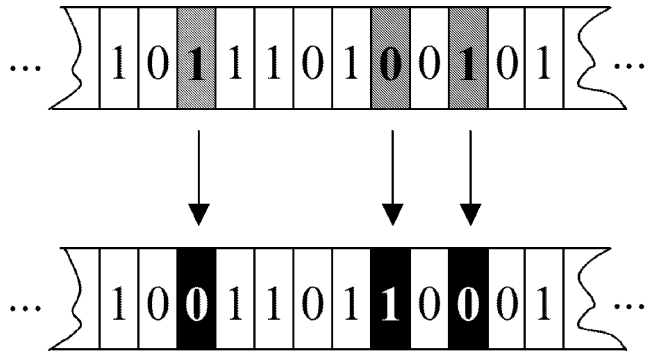


Abbildung 3.5: Zufälliges Kippen von Bits im Chromosom

3.3.3 Crossover

Der Crossover (Rekombination) auf einem binären Chromosom funktioniert ähnlich wie in der Natur. Dazu werden zunächst zwei Individuen aus der Population fitnessbasiert ausgewählt, welche als Elternteile fungieren und Nachkommen erzeugen. Ob ein Crossover durchgeführt wird oder nicht, wird mit einer Crossover-Rate P_C ermittelt.

Die einfachste Crossover-Variante ist der 1-Punkt Crossover. Hierbei wird stochastisch auf Basis einer Gleichverteilung ein Crossover-Punkt zwischen 1 und $L-1$ (L entspricht der Länge der Chromosomen) bestimmt, der für beide Elternchromosomen identisch ist. An diesem Schnittpunkt findet der Crossover statt, indem die Bits rechts vom Crossover-Punkt zwischen den beiden Eltern-Chromosomen vertauscht werden. Es entstehen zwei Nachkommen (Kinder), welche Eigenschaften von beiden Elternteilen besitzen (siehe Abbildung 3.6).

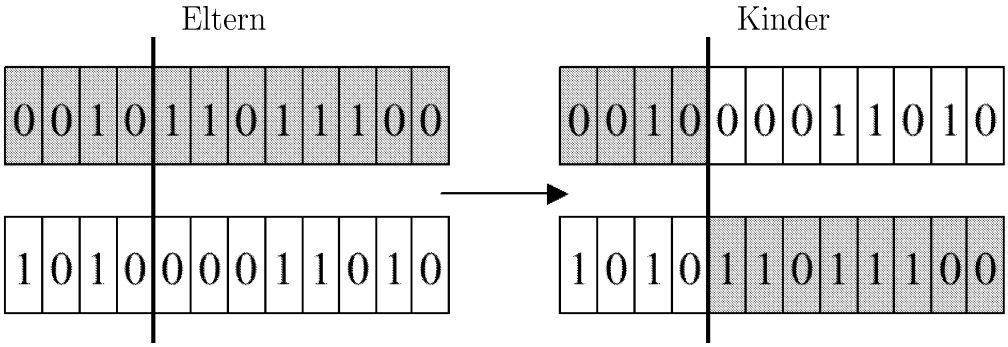


Abbildung 3.6: 1-Punkt Crossover

Beim n-Punkt-Crossover werden n Schnitte durchgeführt und die Zwischenstücke abwechselnd ausgetauscht. Am häufigsten findet der 2-Punkt-Crossover Verwendung, bei dem zwischen zwei Crossover-Punkten das Mittelstück ausgetauscht wird (siehe Abbildung 3.7).

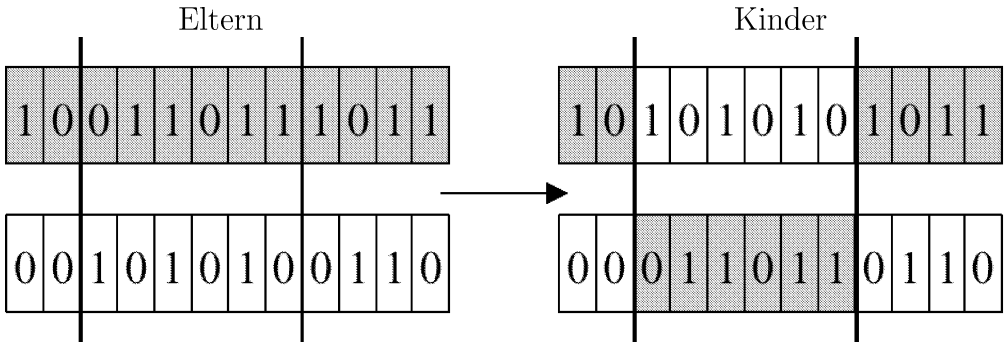


Abbildung 3.7: Prinzip des 2-Punkt Crossover

3.3.4 Selektion

Im Selektionsverfahren werden η Individuen, in Abhängigkeit ihres Fitnesswertes aus der aktuellen Population als Eltern ausgewählt. Aus diesen Individuen werden durch Reproduktion, Mutation oder Crossover Nachkommen erzeugt und in die nächste Generation übernommen. Dabei ist das einfachste Ersetzungsschema das *generational replacement*, bei dem die aktuelle Population vollständig durch ihre Nachkommen ersetzt wird. Ein weiteres Schema ist das *Elitismusprinzip*. Hierbei werden stets die α besten Individuen in die nächste Generation übernommen und verbleiben somit in der Population. Meistens soll das beste Individuum erhalten bleiben, dies bedeutet: α wird gleich eins gesetzt.

Die wichtigsten Konzepte für Selektionsalgorithmen sind:

- fitnessproportionale Selektion
- rangbasierte Selektion (*ranking*)
- Wettkampf-Selektion (*tournament selection*)
- Overselection

fitnessproportionale Selektion

Bei der fitnessproportionalen Selektion erhält jedes Individuum c eine seiner Bewertung $\phi(c)$ direkt proportionale Chance, sich in der nächsten Generation fortzupflanzen. Die Selektionswahrscheinlichkeit p_s eines Individuums c_j ergibt sich zu:

$$p_s(c_j) = \frac{\phi(c_j)}{\sum_{k=1}^{\eta} \phi(c_k)}$$

Dabei werden die Individuen „mit Zurücklegen“ gezogen, so dass Duplikate möglich sind.
Anschaulich kann man sich diese Selektionsform anhand eines Glücksrads mit η unterschiedlich großen Abschnitten vorstellen, wobei die Breite eines Abschnitts der Selektionswahrscheinlichkeit eines Individuums entspricht. Nun dreht man η-mal das Glücksrad, um die Eltern für den nächsten Generationsschritt zu ermitteln.

Rangbasierte Selektion

Hierbei hängt die Selektionswahrscheinlichkeit jedes Individuums von seiner Position in einer auf der Fitness basierenden Rangordnung aller Populationsmitglieder ab. Die Individuen werden nach ihren Rängen sortiert und danach ausgewählt. Dabei spielt die absolute Höhe des Fitnesswertes eine nebensächliche Rolle.

Wettkampf-Selektion

Die übliche Vorgehensweise besteht darin, aus der Population ξ Individuen (wobei $2 \leq \xi < \eta$) mit gleicher Selektionswahrscheinlichkeit $p_s = 1/\eta$ zu ziehen und dann das beste unter ihnen auszuwählen. Dieser Vorgang wird so oft wiederholt, bis alle η Individuen ausgewählt wurden. Über den Wert von ξ läßt sich der Selektiondruck steuern. Mit steigendem ξ nimmt bei konstanter Populationsgröße η der Selektionsdruck zu. Der Vorteil dieses Verfahrens ist, dass keine zeit- oder rechenaufwändigen Operationen notwendig sind. So muss im Gegensatz zu den ersten beiden Verfahren weder die gesamte Population sortiert noch die Summe aller Fitnesswerte gebildet werden, um den Fitnesswert ins Verhältnis zu setzen.

Overselection

Beim Overselection wird zunächst die gesamte Population anhand des Fitnesswertes in zwei Gruppen geteilt, wobei die erste Gruppe das beste Drittel der Population darstellt und die zweite Gruppe von den restlichen zwei Dritteln gebildet wird. Die Auswahl der Individuen erfolgt so, dass die Individuen der ersten Gruppe mit 80% und entsprechend die Individuen der zweiten Gruppe mit 20% Wahrscheinlichkeit gezogen werden.
Diese Methode ist sehr gierig (*greedy*) und kann den Evolutionsprozess erheblich beschleunigen. Der Nachteil hingegen ist, dass dadurch der Selektionsdruck eventuell zu stark werden kann. Hier besteht die Gefahr der *premature convergence* und damit des Stehenbleibens bei einem lokalen Minimum.

Kapitel 4

Entwurf und Implementierung

In diesem Kapitel wird ausgehend vom *Basiskonzept* (siehe Abbildung 4.1), welches eine Unterscheidung zwischen dem *Optimierungsverfahren* und dem eigentlichen *Wachschutzprozess* vorsieht, der Entwurf und die Implementierung des Programms beschrieben.
Die Modellierung des Wachprozesses und die Bedienung des Optimierungsverfahrens soll durch den Benutzer erfolgen. Dabei bildet die Modellierung die Grundlage für das darauf aufsetzende Optimierungsverfahren.

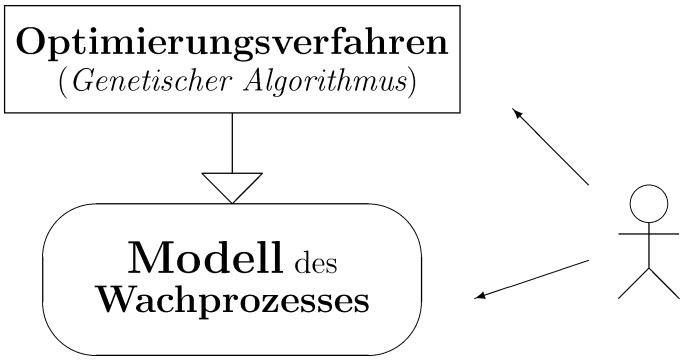


Abbildung 4.1: Basiskonzept des Programms

In den nächsten Abschnitten wird zunächst ein Modell des Wachprozesses entworfen, welches sämtliche Informationen, die für dessen Beschreibung notwendig sind, berücksichtigt.
Im Anschluss daran wird beschrieben, wie die theoretischen Grundlagen konkret in die Praxis umgesetzt wurden und speziell für das Problem der Tourenplanung ein Optimierungsverfahren in Form eines Genetischen Algorithmus entworfen wurde.

Daraufhin wird die Entwicklungsumgebung kurz vorgestellt, sowie anhand des Klassenmodells einige Auszüge der Implementierung erläutert.

4.1 Modellierung des Wachprozesses

Das Problem welches im Rahmen dieser Diplomarbeit untersucht und gelöst werden soll, ist die Tourenplanung eines Wachschutzunternehmens. Das Problem kann allgemein als Vehicle Routing Problem beschrieben werden.
Es gibt mehrere Wachobjekte, die von einem zentralen Fahrzeugdepot aus angefahren und kontrolliert werden müssen. Jedes Wachobjekt muss innerhalb eines definierten Zeitfensters kontrolliert werden. Außerdem kann es vorkommen, dass auf einem Objekt mehrere Kontrollgänge vom Wachpersonal durchzuführen sind. Dazu stehen im Depot mehrere Fahrzeuge zur Verfügung. Die Fahrzeugkapazität spielt hierbei keine Rolle. Jedes Fahrzeug wird durch einen Wachangestellten geführt, welcher auch die Kontrollen auf den einzelnen Objekten durchzuführen hat. Hinzu kommen die Zeitrestriktionen an die einzelnen Fahrer selbst, wie Arbeitsbeginn, Arbeitende, minimale und maximale Arbeitszeit.
Ein *Tourenplan* soll beschreiben, wie die einzelnen Objekte auf mehrere unterschiedliche *Routen* zeitlich aufgeteilt sind. Er gibt also die Abfahrtsreihenfolge jedes Wachangestellten vor. Die Erstellung des Tourenplans stellt ein Optimierungsproblem dar, bei dem es darum geht, die Objekte so auf n Routen aufzuteilen, dass alle Zeitbedingungen erfüllt werden und der dafür benötigte Zeitaufwand bzw. die Gesamtkilometerzahl möglichst minimal wird.

Dabei sind folgende Informationen für die Modellierung des Wachprozessses im Hinblick auf die Implementierung eines entsprechenden Optimierungsverfahrens zu berücksichtigen:

- Anlegen und Aktualisieren von Wachobjekten
- Definition der Wachanweisungen
  - Angabe der Zeitfenster
  - Anzahl der Überwachungen eines Objekts
  - Gültigkeit einer Wachanweisung bezüglich der Wochentage
  - Angabe der notwendigen Kontrollzeiten für jedes Objekt

- Streckenangaben (Entfernungen bzw. Fahrzeiten zwischen den einzelnen Objekten)
- Zeitrestriktionen der Fahrer

Bevor die einzelnen Punkte in den nachfolgenden Abschnitten näher erläutert werden, soll das *Entity-Relationship-Modell* [1] in Abbildung 4.2 zunächst die Beziehungen zwischen den verschiedenen Datentypen beschreiben.

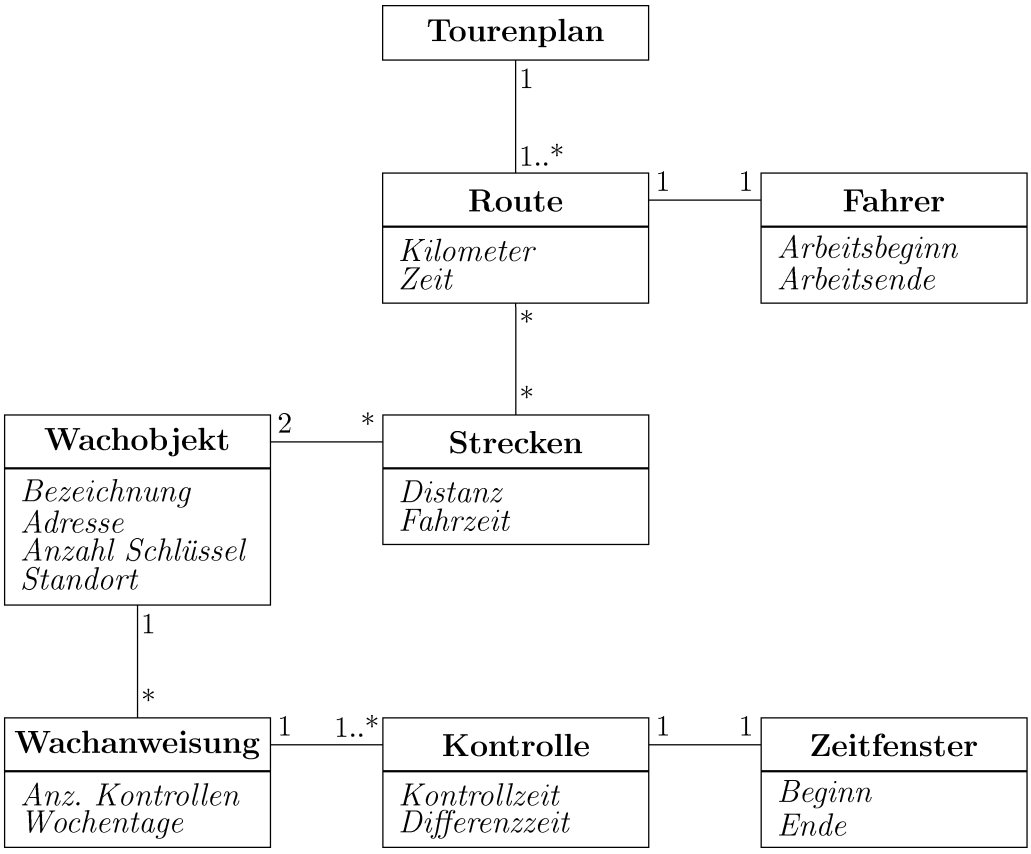


Abbildung 4.2: Entity-Relationship-Modell des Wachprozesses

[1] Das Entity-Relationship-Modell (*kurz:* ER-Modell) wurde 1976 von P. Chen zur Datenmodellierung entwickelt (vgl.[Ba01]).

4.1.1 Anlegen von Wachobjekten

Die Beschreibung eines Wachobjekts erfolgt durch die Angabe einer Objektbezeichnung und Adresse sowie des Standorts innerhalb einer digitalen Straßenkarte[2]. Eine weitere wichtige Information zum Wachobjekt ist die Angabe über die Anzahl der vorhandenen Schlüssel. Dadurch wird beschränkt, von wie vielen Wachangestellten die Kontrollgänge des Objekts maximal durchgeführt werden können. Folglich kann ein Objekt nur in so vielen Touren auftreten, wie letztendlich Schlüssel für dieses Objekt vorhanden sind.

Zum Anlegen von Wachobjekten dient die Klasse `ObjektStruct`:

```
class ObjektClass
{
public:
    char Bezeichnung[80];
    char Adresse[255];
    unsigned int Knoten,X,Y;
    unsigned char Anz_Key;
};
```

4.1.2 Definition der Wachanweisungen

Eine Wachanweisung legt genau fest, wie die Kontrolle für ein bestimmtes Objekt auszusehen hat und ist mit dem Kunden vertraglich festgelegt worden. In jeder Wachanweisung sind folgende Punkte eindeutig beschrieben:

- Angabe der Zeitfenster
- Anzahl der Überwachungen eines Objekts
- Gültigkeit einer Wacchanweisung bezüglich der Wochentage
- Angabe der notwendigen Kontrollzeiten für jedes Objekt

[2]Die Position innerhalb des Straßennetzes wird dazu verwendet, die Entfernungen zwischen den einzelnen Objekten zu bestimmen. Nähere Informationen über die Repräsentation des Straßennetzes finden sich im Abschnitt 4.3.

Es folgt nun eine Beschreibung der einzelnen Punkte:

Angabe der Zeitfenster

Zeitfenster legen fest, in welcher Zeitspanne ein zu bewachendes Objekt angefahren und kontrolliert werden soll (z.B. um 20.00 Uhr, zwischen 3 und 4 Uhr, nach Mitternacht). Dieser Zeitraum wird vom Kunden, dessen Objekt bewacht werden soll, vertraglich festgelegt und ist für die Fahrer verbindlich.

Eine Zeitspanne wird definiert mit:

von *(earliest start time)* ist der Zeitpunkt, zu dem frühestens mit einer Kontrolle beim Kunden begonnen werden kann

bis ist der letztmögliche Zeitpunkt, zu dem bei einem Kunden mit einem Kontrollgang begonnen werden muss

Anzahl der Überwachungen eines Objekts

Ebenfalls mit dem Kunden vertraglich festgelegt, ist die Anzahl der durchzuführenden Kontrollen auf dem zu überwachenden Objekt. Maximal werden bis zu vier Kontrollen auf einem Objekt durchgeführt (`MAX_KONTROLLEN=4`). Für die einzelnen Kontrollgänge gibt es wiederum separate Zeitfenster. So kann beispielsweise vom Kunde festgelegt worden sein, dass sein Objekt zweimal in der Nacht zu kontrollieren ist. Dabei soll eine Kontrolle vor und die zweite erst nach Mitternacht stattfinden.

Bei der Durchführung von zwei oder mehreren Kontrollgängen ist allerdings darauf zu achten, dass nicht direkt im Anschluss an eine Kontrolle gleich nochmals ein Kontrollgang auf demselben Objekt durchgeführt wird, oder sich zwei Wachangestellte auf dem Objekt begegnen. Aus diesem Grund erscheint es sinnvoll, zusätzlich eine zeitliche Differenz anzugeben, welche die Zeitspanne angibt, die mindestens zwischen zwei aufeinanderfolgenden Kontrollen eines Objekts liegen sollte.

Gültigkeit einer Wachanweisung bezüglich der Wochentage

Wachanweisungen gelten nur an bestimmten Wochentagen. Oft kommt es vor, dass z.B. innerhalb der Woche andere Vorschriften und Zeitfenster bestehen als am Wochenende. Aus diesem Grund muss man festlegen können,

an welchen Wochentagen eine Wachanweisung gilt und gegebenenfalls neue Wachanweisungen (mit anderen Zeitfenstern etc.) für davon abweichende Wochentage definieren.

Angabe der notwendigen Kontrollzeiten eines Objekts

Für jede durchzuführende Kontrolle erfolgt die Angabe einer Kontrollzeit. Das ist die Zeit, die der Wachangestellte benötigt, um auf dem Objekt seinen Kontrollgang durchzuführen.

Aus den oben beschriebenen Eingabedaten ergibt sich folgende Klassendefinition für eine Wachanweisung:

```
class AnweisungClass
{
public:

   unsigned char Objekt_Id;
   bool  Montag,
         Dienstag,
         Mittwoch,
         Donnerstag,
         Freitag,
         Samstag,Sonntag,Feiertag;
   unsigned char Anz_Kontrollen;
   int   Diff[MAX_KONTROLLEN-1],
         von[MAX_KONTROLLEN],
         bis[MAX_KONTROLLEN],
         Kontrollzeit[MAX_KONTROLLEN];
};
```

Dabei gibt `Objekt_Id` den Index des Wachobjektes an, auf das sich diese Wachanweisung bezieht. Die booleschen Variablen `Montag`, `Dienstag`,...,`Sonntag` und `Feiertag` beschreiben, an welchen Wochentagen die Wachanweisung gültig (`true`) oder nicht gültig ist (`false`).
Die Anzahl der durchzuführenden Kontrollen wird angegeben durch die Variable `Anz_Kontrollen`. Für die Angabe der Zeitfenster der einzelnen Kontrollen werden die Felder `von[]` und `bis[]` verwendet. Ebenso wird für jede Kontrolle die dafür benötigte Zeit (`Kontrollzeit[]`) angegeben.

Die zwischen den einzelnen Kontrollgängen einzuhaltende Differenzzeit wird mit `Diff[]` beschrieben. Dabei gibt `Diff[0]` die Zeit an, die mindestens zwischen Kontrollgang 1 und 2 vergangen sein muss, `Diff[1]` beschreibt demnach die Zeit zwischen Kontrolle 2 und 3 usw. Sämtliche Zeitangaben werden in Minuten gespeichert.

4.1.3 Zeitrestriktionen der Fahrer

Zeitbedingungen bestehen nicht nur für die zu überwachenden Objekte, sondern auch für die Fahrer. Es müssen minimale und maximale Arbeitszeiten eingehalten, sowie frühest- bzw. spätmöglichster Arbeitbeginn und Arbeitsende festgelegt werden. Auch diese Zeitrestriktionen dürfen nach Möglichkeit nicht verletzt werden.

Es ergibt sich folgende Klasse für die Definition der Arbeitszeiten der Fahrer:

```
class FahrerClass
{
public:

   int Start;
   int EndeMin,EndeMax;
};
```

4.1.4 Streckenangaben zwischen den einzelnen Wachobjekten

Für die Bewertung der Touren ist es ist notwendig ungefähr zu wissen, welche Zeit man benötigt, um zwischen den einzelnen Objekten hin und her zu fahren. Diese Information stellt eine wichtige Voraussetzung zur Bestimmung einer möglichst effizienten Route dar und muss dem Rechner durch Angabe einer Entfernungsmatrix M_{ij} zur Verfügung gestellt werden.
Diese Matrix gibt die tatsächliche Fahrzeit bzw. die Kilometer von einem beliebigen Objekt i zu jedem anderen Objekt j an. Die Bereitstellung dieser Entfernungsmatrix stellt eine nicht triviale Aufgabe dar. Drei mögliche Ansätze dafür sind in Abschnitt 4.3 *Repräsentation des Straßennetzes* beschrieben.

4.2 Beschreibung eines GA für das Vehicle Routing Problem

Nachdem im Kapitel 3 die grundlegenden Mechanismen der Genetischen Algorithmen näher erläutert wurden, sollen in diesem Abschnitt die Erkenntnisse aus dem theoretischen Teil konkret in die Praxis (bezogen auf das Problem der Tourenplanung) angewendet werden. Dazu wird in Abbildung 4.3 zunächst der grobe Ablauf des GA dargestellt.

```
Start
Generiere zufällige Startpopulation
Wiederhole
     Bewerte alle Individuen anhand einer Fitnessfunktion
         Neue Population={}
         Solange neue Population noch unvollständig, wiederhole
             Selektion eines Individuums der alten Population
             Führe entweder durch
                 Replikation oder
                 Mutation oder
                 Crossover
             Gemäß den Variationsraten P_R, P_M, P_C
             Füge entstandene Nachkommen neuer Population hinzu
         Schleifenende
         Ersetze alte Population durch neue Population
         (*nächste Generation*)
     Bis Abbruchbedingung erfüllt ist
     Gib Ergebnisse aus
Stop
```

Abbildung 4.3: Allgemeines Ablaufschema des GA

Zunächst wird eine Anfangspopulation mit zufällig erzeugten Individuen generiert. Jedes Individuum stellt eine mögliche Lösungsrepräsentation und somit einen Tourenplan dar.

Die anfangs zufällig erzeugten Individuen sind bezüglich ihrer Lösungsgüte natürlich noch sehr schlecht. Allerdings werden sich diese verschiedenen Individuen stark voneinander unterscheiden. So wird es unter diesen vielen

Tourenplänen einige geben, die ein wenig besser sind als andere. Diese werden demnach auch einen besseren Fitnesswert besitzen als schlechtere Individuen.
In Abhängigkeit dieses Fitnesswertes werden nun im Selektionsprozess Individuen aus der aktuellen Population ausgewählt und im nachfolgenden Schritt eventuell einer Variation unterworfen. Dazu wird mittels den Variationsraten P_R,P_M,P_C bestimmt, ob das Individuum entweder unverändert in die nächste Generation übernommen (Reproduktion), mutiert oder mit einem weiteren Individuum der Crossover angewendet wird. Diejenigen Individuen, die einen höheren Fitnesswert besitzen als andere, haben eine höhere Wahrscheinlichkeit zu überleben und sich fortzupflanzen. Durch die geringfügigen Veränderungen der Individuen können Tourenpläne entstehen, die bezüglich der Zielfunktion besser sind als vorhergehende.

Ist die neue Population vollständig erzeugt, so ersetzt diese die alte Population und das Verfahren wird solange wiederholt bis ein bestimmtes Abbruchkriterium erreicht wurde.
Als Abbruchbedingung gilt neben der maximalen Anzahl an Generationen $g_{\max}$ auch das Drücken der ESC-Taste durch den Benutzer. Dadurch hat man die Möglichkeit, das Verfahren vorzeitig zu beenden, falls bereits eine brauchbare Lösung gefunden wurde. Dies setzt allerdings voraus, dass dem Benutzer der Verlauf des Optimierungsverfahrens auf Bildschirm angezeigt wird. Nachdem das Optimierungsverfahren beendet wurde, wird die bis dahin beste gefundene Lösung ausgegeben.

Die Anpassungen der einzelnen Genetischen Operationen und der Lösungsrepräsentation auf die Besonderheiten des Vehicle Routing Problems sind in den folgenden Abschnitten beschrieben.

4.2.1 Lösungsrepräsentation eines Chromosoms

Die Codierung des Problems ist von großer Bedeutung und hat großen Einfluß auf die Leistungsfähigkeit des GA. Mit der Lösungsrepräsentation wird schließlich festgelegt, wie der GA das gegebene Optimierungsproblem „sieht“.

Grundsätzlich lassen sich verschiedene Variablentypen und Probleme binär codieren, so z.B. auch reelle Zahlen oder kombinatorische Probleme. Jedoch ist dieser Vorteil der Anwendungsunabhängigkeit von binärer Codierung gleichzeitig ein Nachteil, da keine strukturellen Eigenheiten vieler Problem-

stellungen in angemessener Weise wiedergespiegelt werden. Das kann sich negativ auf Erfolgsaussichten und Effektivität eines GA auswirken. Ein weiterer Nachteil besteht in der rechenzeitintensiven Decodierung.

Nach Ansicht von Nissen [Ni94] sollten sich die Lösungsrepräsentationen und Codierungen möglichst direkt aus der gegebenen Problemstellung ableiten. In vielen praktischen Anwendungen hat man daher Repräsentationsformen höherer Abstraktionsebenen gewählt, wenn sie für die Problemstellung effizienter bzw. natürlicher erschienen. Dazu zählen z.B. reellwertige Lösungsdarstellungen sowie Permutationscodierungen.

Bei kombinatorischen Optimierungsproblemen lassen sich die einzelnen Lösungen als Permutationen darstellen. Im Fall des bekannten Traveling-Salesman-Problems, bei dem eine kostenminimale Rundreise durch n verschiedene Orte gesucht ist, bestehen mögliche Lösungen in unterschiedlichen Permutationen der n Orte. Diese sogenannte *Pfadrepräsentation* lässt sich unmittelbar auf einem eindimensionalen String abbilden (siehe Abbildung 4.4). Dabei braucht der Start- und Zielort A nicht explizit in den String mit aufgenommen zu werden.

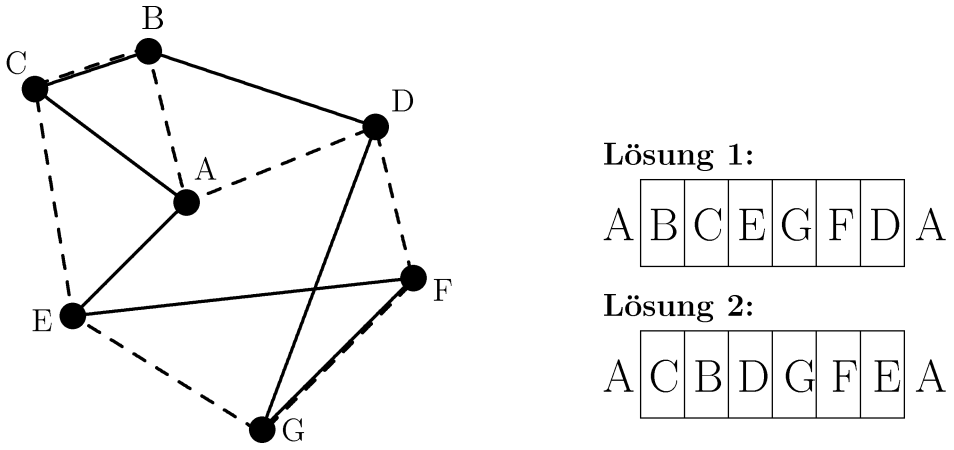


Abbildung 4.4: Pfadrepräsentation des TSP

Wie man sieht, können die Lösungen des TSP z.B. mit einem Buchstaben-String dargestellt werden. Dies ist für die Praxis allerdings ungeeignet, da hier die einzelnen Ortsbezeichnungen nicht nur aus einem Buchstaben bestehen. Dann müssten zur Speicherung der Orts- bzw. Kundennamen Zeichenketten

verwendet werden. Dies hat im Hinblick auf Populationsgrößen von mehr als 100 Individuen einen enormen Speicherplatzbedarf zur Folge.
Aus diesem Grund wird eine Lösungsrepräsentation in Form eines *Integer-Strings* (siehe Abbildung 4.5) verwendet, bei dem jedem Ort eindeutig eine natürliche Zahl zugeordnet ist ($A \mapsto 0, B \mapsto 1, C \mapsto 2, \ldots, G \mapsto 7$).

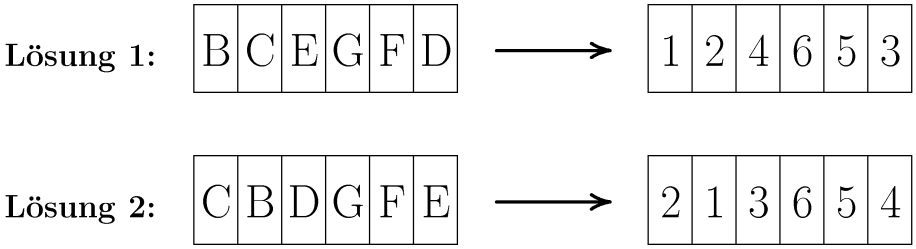


Abbildung 4.5: Lösungsrepräsentation in Form eines Integer-Strings

Man kann bei den Strings von Permutationscodierungen oder reellwertigen Lösungsrepräsentationen strenggenommen nicht mehr von Chromosomen (Genotyp-Ebene) sprechen, da der GA nun mehr auf phänotypischer Ebene arbeitet. Nach Nissen [Ni94] sollte die Wahl einer binären Repräsentationsform nicht zum entscheidenen Kriterium für das Prädikat „Genetischer Algorithmus“ werden.

Doch wie gestaltet sich nun die Lösungsrepräsentation für das Vehicle Routing Problem und somit auch für das Problem der Tourenplanung der Safe Wachschutz/Allservice GmbH?

Dort stehen zur Erfüllung der Kundenaufträge mehrere Fahrzeuge zur Verfügung. Die einzelnen Wachobjekte müssen also demzufolge auf mehrere Routen verteilt werden (siehe Abbildung 4.6). Doch wie soll nun eine geeignete Lösungscodierung für mehrere Routen aussehen?
Eine Möglichkeit besteht darin, für jedes Fahrzeug einen separaten Integer-String vorzusehen. Einem Individuum sind also mehrere Strings zugeordnet, wobei ein String genau eine spezielle Route für ein Fahrzeug beschreibt. Dabei können die Längen der einzelnen Strings variieren. Dies ist auch notwendig, denn man kann sich leicht vorstellen, dass eine Tour mehr Objekte enthalten kann als eine andere.

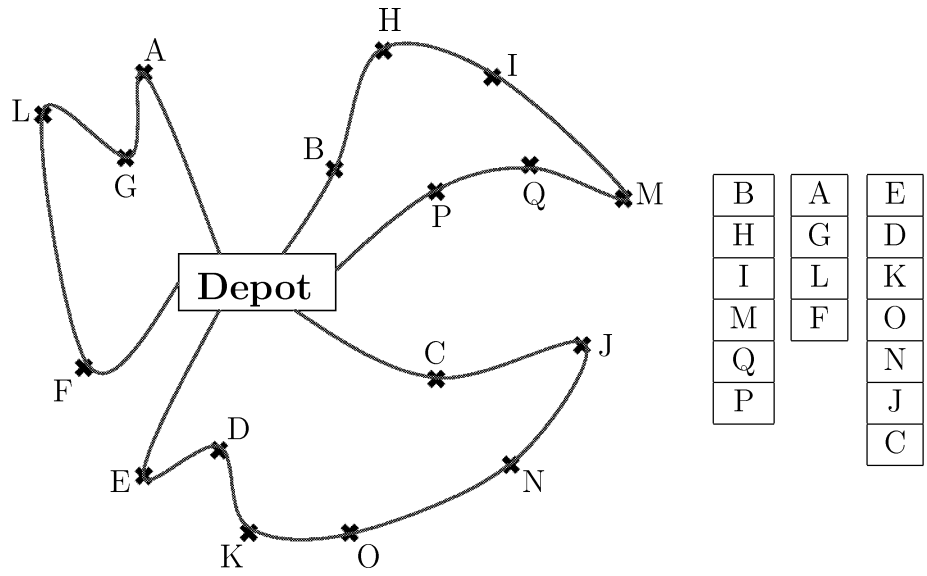


Abbildung 4.6: Verteilung der Objekte auf mehrere Routen

Die Verwendung von Individuen, die durch mehrere Strings mit variablen Stringlängen beschrieben werden, weicht erheblich von der allgemeinen Darstellung eines GA ab. Deshalb wurde in dieser Arbeit eine andere Repräsentationsform für das Vehicle Routing Problem entwickelt.

Diese Codierung basiert auf einem einzigen String zur Darstellung eines Individuums, wie schon beim TSP verwendet. Dieser String kann sämtliche Permutationen der anzufahrenden Objekte enthalten. Doch wie unterscheidet man dabei die einzelnen Routen der unterschiedlichen Fahrzeuge?

Dazu fügt man an beliebigen Stellen innerhalb des Strings ein sogenanntes Blankzeichen □ ein und unterteilt damit die Gesamtroute in mehrere Teilsequenzen (siehe Abbildung 4.7). Jedes Teilstück repräsentiert somit die Tour eines Wachangestellten.
Bei m Fahrzeugen müssen also $m-1$ Blanksymbole eingefügt werden, um den Gesamtstring in m Teilstücke zu unterteilen. Der Vorteil dieser Repräsentation eines Individuums liegt darin, dass zur Darstellung von m Routen verschiedener Fahrzeuge nur ein String verwendet werden muss. Außerdem sind die Stringlängen aller Individuen gleich lang, denn es ergibt sich für ein Problem mit k Kunden und m Fahrzeugen eine Stringlänge $l = k + (m - 1)$.

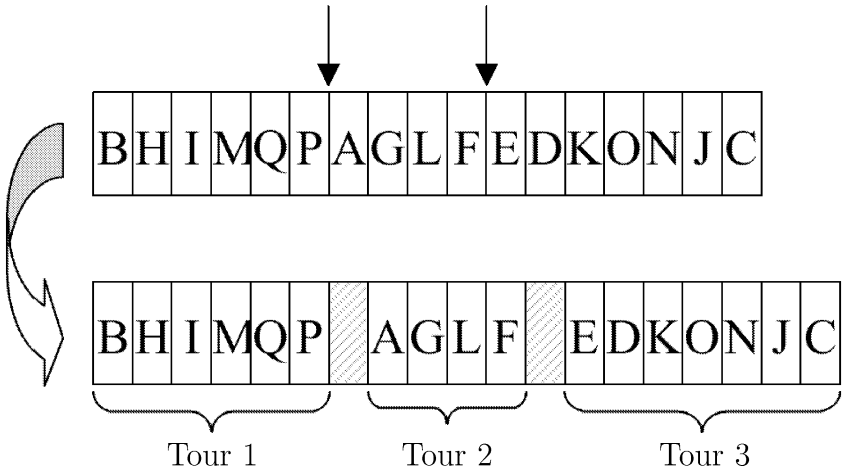
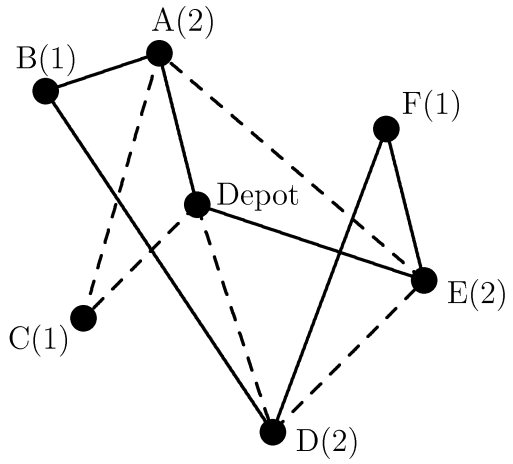


Abbildung 4.7: Einfügen von Blanksymbolen zur Unterscheidung von mehreren Routen innerhalb eines Strings

Bisher wurde allerdings nur der Spezialfall des *Vehicle Routing Problems* betrachtet, bei dem davon ausgegangen wird, dass jedes Objekt genau einmal besucht wird. Die Wachanweisungen sind aber so gestaltet, dass auf einem Objekt in der Regel mehrere Kontrollen stattfinden können.
Jede Kontrolle hat dabei wiederum ein individuelles Zeitfenster und eine eigene Kontrollzeit. Somit muss jede Kontrolle als ein separater Knoten aufgefasst werden und demzufolge auch in die Lösungspermutation mit aufgenommen werden.

Diese Problematik wird in dem Beispiel-Szenario in Abbildung 4.8 verdeutlicht. Eigentlich gibt es nur die 6 Wachobjekte A,B,C,D,E und F, jedoch gibt es zu jedem Objekt eine Anzahl von durchzuführenden Kontrollgängen, diese sind in den Klammern angegeben.
Jede Kontrolle muss als separater Knoten in die Permutation mit aufgenommen werden. Somit besteht das Problem nicht aus 6 Objekten, sondern genau genommen aus 9 Objekten.



Tour1:

A_1	B_1	D_1	F_1	E_2

Tour2:

C_1	A_2	E_1	D_2

Abbildung 4.8: Auf einigen Objekten können auch mehrere Kontrollen stattfinden

Den Kontrollen der einzelnen Objekte wird nacheinander eine eindeutige Knoten-Nummer zugewiesen:

Kontrolle	Nummer
A_1	1
A_2	2
B_1	3
C_1	4
D_1	5
D_2	6
E_1	7
E_2	8
F_1	9

Tabelle 4.1: Zuweisung einer eindeutigen Knoten-Nummer zu jeder Kontrolle

Damit die Information, auf welches Objekt sich der Knoten k bezieht nicht verloren geht, wird die Zuordnung Knoten$\mapsto$Objekt in einer Zuordnungstabelle `ObjektKnoten[]` gespeichert.

```
int KnotenObj[MAX_KNOTEN]; //Zuordnungstabelle
```

Für jede Knoten-Nummer wird die zugehörige Objekt-ID abgespeichert. Damit sind alle unterschiedlichen Kontrollen innerhalb des Chromosoms mit der eindeutigen Knotennummer beschrieben und dem entsprechenden Objekt zugeordnet.

Aus den vorherigen Betrachtungen ergibt sich die Klasse `ChromosomClass` zur Beschreibung der einzelnen Chromosomen. Den Kernpunkt dieser Klasse bildet ein `integer`-Feld `Gen[]`. Die einzelnen Feldelemente entsprechen dabei den Knoten-Nummern, welche die einzelnen Kontrollgänge repräsentieren.

```
class ChromosomClass
{
public:
    unsigned int Gen[MAX_GENE];
};
```

Folglich werden die Individuum in der Form

$$
\begin{array}{ll}
 & \overbrace{5,8,12,7,18,12,1}^{\text{Tour1}},0,\overbrace{2,3,14,6,7}^{\text{Tour2}},0,\overbrace{11,4,9,\ldots}^{\text{Tour3}} \\
\textit{Chromosom 1}: & \\
 & \overbrace{12,2,1,9,3,8,7}^{\text{Tour1}},0,\overbrace{15,4,11,13,\ldots}^{\text{Tour2}} \\
\textit{Chromosom 2}: & \\
 & \overbrace{4,13,10,7}^{\text{Tour1}},0,\overbrace{2,6,5,11,3,9,17,15}^{\text{Tour2}},0,\overbrace{1,8,\ldots}^{\text{Tour3}} \\
\textit{Chromosom 3}: & \\
\vdots & \vdots
\end{array}
$$

beschrieben, wobei das Blankzeichen $\square$, zum Trennen der unterschiedlichen Routen, durch das Auftreten einer 0 symbolisiert ist.

4.2.2 Mutationsoperator

Die Mutation ist ein 1-Elter-Operator[3], da er auf einem Elternindividuum ausgeführt wird. Mutation bedeutet eine zufällige Veränderung im Erbgut. Durch die Verwendung einer Permutationscodierung als Lösungsrepräsentation für das Vehicle Routing Problem besteht bei zufälliger Veränderung eines Gens die Gefahr ungültige Lösungen zu erzeugen, in denen einzelne Knoten mehrfach auftauchen, während andere fehlen. Abbildung 4.9 verdeutlicht diese Problematik.

[3]Ein Elternteil wird auch als „*Elter*“ bezeichnet.

Durch Mutation wird der Knoten „4“ aus Gen Nr. 4 in eine „3“ modifiziert. Die sich daraus ergebene Tour ist ungültig, da sie zum einen den Knoten 3 doppelt besucht, während der Knoten 4 nicht angefahren werden würde.

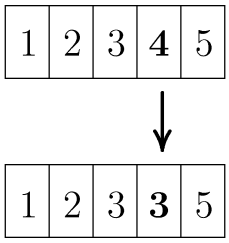


Abbildung 4.9: Zufällige Veränderung eines Gens (Kippen einer 4 in eine 3)

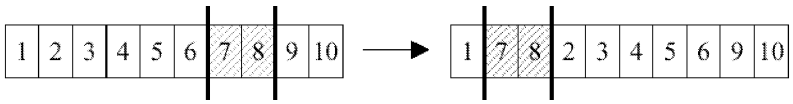
Durch zufällige Veränderung eines Knotens, ohne Berücksichtigung der zugrundeliegenden Permutations-Codierung, kommt es also zur Erzeugung von nicht konsistenten Lösungen. Prinzipiell könnte man sich daran nicht stören und diesen ungültigen Lösungen besonders schlechte Fitnesswerte zuweisen. Doch würde diese Vorgehensweise den ohnehin schon extrem großen Suchraum noch weiter aufblähen. Besser wäre es, die Erzeugung von inkonsistenten Touren generell auszuschließen, indem man den Mutationsoperator auf die Besonderheiten der Permutationscodierung anpasst. Für solche Reihenfolge-Probleme wurden spezielle Suchoperatoren entwickelt, die gemeinhin als Sequenzoperatoren bezeichnet werden.

Mögliche solcher 1-Elter-Sequenzoperatoren sind in [Ni94] dargestellt (siehe Abbildung 4.10). Dabei wird die Sequenz des Elternindividuums teilweise umgeordnet und auf die Nachkommen vererbt, wobei stets konsistente Lösungen erzeugt werden.

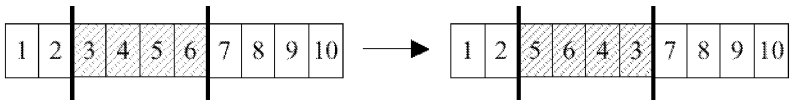
Das einfachste Beispiel für einen möglichen Mutationsoperator ist der sogenannte *Zweiertausch*. Hier werden stochastisch zwei Positionen innerhalb des Chromosoms bestimmt, wobei alle Positionen mit gleicher Wahrscheinlichkeit ausgewählt werden können. Anschließend werden die ausgewählten Elemente vertauscht.
Eine andere Möglichkeit besteht darin, Teilstücke aus der Sequenz herauszubrechen und an anderer Stelle wieder einzufügen. Dieser Vorgang wird als *Verschiebung eines Sequenz-Teilstückes* bezeichnet.

Beim *Scramble Sublist Operator* wird zunächst eine beliebige Teilsequenz aus dem Chromosom bestimmt und daraufhin die Elemente dieses Sequenzteilstücks zufällig permutiert.
Der *Inversionsoperator* schneidet eine beliebige Teilsequenz heraus, invertiert diese und setzt sie an der gleichen Stelle wieder ein.

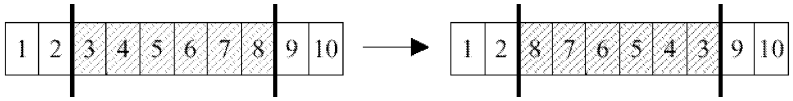
a.) Prinzip des Zweiertausches



b.) Verschiebung eines Sequenz-Teilstückes



c.) Scramble sublist - Operator



d.) Inversion

Abbildung 4.10: 1-Elter-Sequenzoperatoren (aus [Ni94])

Diese einzelnen Varianten des Mutationsoperators wurden entsprechend in das Programm implementiert und eignen sich sehr gut als lokaler Suchoperator. Im Falle der Mutation eines ausgewählten Individuums wird stochastisch und gleichverteilt eine dieser 4 Operatoren angewendet.

4.2.3 Der Crossover-Operator

Bei der Durchführung des Crossover auf Permutationscodierungen ist die gleiche Problematik zu beachten, wie bei der Mutation. Eine Vertauschung der Stringeinträge nach einem zuvor gewählten Crossover-Punkt kann zu inkonsistenten Nachkommen führen, so dass einige Knoten mehrmals auftauchen können, während andere aus der Tour verschwinden.

Beispielsweise erzeugt der 1-Punkt-Crossover im folgenden Beispiel unzulässige Lösungen:

E1:	**2**	**3**	**7**	**4**	|	**10**	**8**	**9**	**6**	**1**	**5**
E2:	5	6	3	8	|	2	1	7	10	4	9

N1:	**2**	**3**	**7**	**4**	|	2	1	7	10	4	9
N2:	5	6	3	8	|	**10**	**8**	**9**	**6**	**1**	**5**

Die Tatsache, dass es sich beim VRP um ein reines Abfolgeproblem handelt, d.h. alle Chromosomen tragen die gleichen Werte und unterscheiden sich nur in der Abfolge dieser Werte, macht spezielle Tauschoperationen notwendig.

Ein Crossover-Operator, welcher auf solche Permutationscodierungen angewendet werden kann, ist der *Partially matched crossover* (*kurz:* PMX-Crossover)(vgl. [Jo02]). Dieser führt auf zwei Individuen einen Crossover durch und garantiert die Erzeugung von konsistenten Nachkommen. Bezogen auf das Tourenplanungsproblem bedeutet dies, dass in den erzeugten Touren jeder Knoten genau einmal erhalten bleibt.

PMX-Crossover

Der PMX-Crossover arbeitet ähnlich wie der in Abschnitt 3.3.3 beschriebene 2-Punkt-Crossover. Zwei Elternindividuen E1 und E2 werden aus der alten Population ausgewählt und daraus zwei Nachkommen N1 und N2 erzeugt. Dabei werden zunächst zwei beliebige Crossover-Punkte bestimmt. Nun wählt man einen beliebigen Teilabschnitt aus, vertauscht ihn und kopiert ihn direkt in die Nachkommen.

E1:	2	3	7	4	|	**10**	**8**	**9**	|	6	1	5
E2:	5	6	3	8	|	**2**	**1**	**7**	|	10	4	9

N1:	x	x	x	x	|	2	1	7	|	x	x	x
N2:	x	x	x	x	|	10	8	9	|	x	x	x

Danach werden so viele Gene wie möglich von den Eltern übernommen, bis auf diejenigen, die bereits in der eingefügten Sequenz vorhanden sind.

N1:	x	**3**	x	**4**	|	2	1	7	|	**6**	x	**5**
N2:	**5**	**6**	**3**	x	|	10	8	9	|	x	**4**	x

Die restlichen Gene werden entsprechend der Zuordnung der Teilsequenzstücke 10↔2, 8↔1 und 9↔7 belegt. Das Ergebnis sind zwei konsistente Nachkommen, welche Eigenschaften (in Form von Teilstrecken) beider Elternteile aufweisen.

N1:	**10**	3	**9**	4	|	2	1	7	|	6	**8**	5
N2:	5	6	3	**1**	|	10	8	9	|	**1**	4	**7**

4.2.4 Die Fitnessfunktion

Die Bestimmung eines Fitnesswertes enspricht der Bewertung eines Individuums bezüglich der Lösungsgüte. Im Beispiel der Tourenplanung wäre dies zum einen die Gesamtlänge (in Kilometer) aller m Touren oder aber die Fahrzeit (in Minuten). Es erscheint sinnvoller die Fahrzeit (engl: *travel time*) als Optimierungskriterium zu betrachten, anstatt die Gesamtlänge zu verwenden. Denn die kürzeste Streckenlänge impliziert i.a. nicht die kürzeste Fahrzeit. So gelangt man beispielsweise über die Autobahn schneller zum Ziel, obwohl es eventuell eine Verbindung über Land gibt, die etwas kürzer ist. Das hängt ganz einfach damit zusammen, dass man auf der Autobahn eine höhere Durchschnittsgeschwindigkeit erreicht als auf der Landstraße, zumal man dort noch einige Ortschaften durchqueren muss.
Programmtechnisch wird für die Fahrzeit eine Variable t_{travel} angelegt, die die kummulierten Fahrzeiten (incl. Kontrollzeiten) der einzelnen Wachangestellten enthält. Für ein beliebiges Individuum kann diese Größe relativ einfach aus den Angaben der Entfernungsmatrix und den Kontrollzeiten der Wachanweisungen bestimmt werden. Allerdings stellt dies nur ein Optimierungskriterium dar, denn neben der Optimierung der Fahrzeit muss noch die Erfüllung sämtlicher Zeitrestriktionen und Bedingungen berücksichtigt werden. Es können folgende Arten von Nebenbedingungen verletzt werden:

- Die Kontrolle des Objekts wird außerhalb des Zeitfensters durchgeführt (*Zeitfensterverletzung*)
- Die Arbeitszeiten der Fahrer werden verletzt (z.B. Tour zu lang) (*Arbeitszeitverletzung*)
- Ein Objekt, für welches nur ein Schlüssel existiert, ist in mehreren Touren aufgeführt (*Schlüsselverletzung*)

Ein Tourenplan, welcher zwar sämtliche Objekte in optimaler Zeit besucht ist nutzlos, wenn die einzelnen Objekte nicht in den dafür vorgesehenen Zeitfenstern angefahren werden. Es treten also mehrere Zielsetzungen auf, welche es zu berücksichtigen gilt. Man spricht dabei auch von einer sogenannten *Mehrzieloptimierung.*
Die Entscheidung ist leicht, wenn es eine Lösung gibt, die hinsichtlich jedes Zielkriteriums alle anderen Lösungsalternativen übertrifft. Dies ist jedoch die Ausnahme. Mögliche Ansätze zur Mehrzieloptimierung mit einem GA sind der *Aggregationsansatz* und *Ansätze mit wechselnden Zielen* (vgl. [Ni97]).

Aggregationsansatz

Beim Aggregationsansatz werden die Funktionswerte $f(x)$ einer Lösung x bei den ξ verschiedenen Zielkriterien bestimmt und in gewichteter Form zu einem Gesamt-Zielfunktionswert $F(x)$ aggregiert, der folgende Form haben kann:

$$F(x) = w_1 \cdot f_1(x) + w_2 \cdot f_2(x) + \ldots + w_\xi \cdot f_\xi(x) \tag{4.1}$$

Dabei entsprechen die $w_j \in \mathbb{R}$ $(j = 1, 2, \ldots, \xi)$ den subjektiven Zielgewichtungen. Somit fließen die verschiedenen Größen direkt in die Bewertung der Tourenpläne ein und das Optimierungsproblem mit mehreren Teilzielen kann dank der Aggregatfunktion ohne spezielle Modifikationen am GA wie jedes Einzelproblem behandelt werden.

Ansätze mit wechselnden Zielen

Auch bei diesem Ansatz werden zunächst die Funktionswerte einer Lösung hinsichtlich der unterschiedlichen ξ Zielkriterien ermittelt. Die Idee hinter dem Ansatz mit wechselnden Zielen ist die, dass man die Selektion in ξ Teilschritte zerlegt. In jedem Teilschritt wählt man nach Maßgabe des betrachteten Zielkriteriums Individuen aus der Gesamtpopulation aus. So können Individuen, die in mehreren Optimierungszielen gute Werte erreichen, mehr Nachkommen erzeugen als andere.

In dieser Arbeit wird der Aggregationsansatz näher verfolgt, welcher äquivalent zur sogenannten *Strafterm-Methode* ist. Die Idee bei der Methode der Strafterme besteht darin, Terme zu der Zielfunktion *F(x)* dazu zu addieren, wenn Randbedingungen verletzt werden, so dass illegale Individuen schlechter bewertet werden als legale.

Auf diese Weise steigen mit jeder nicht erfüllten Zeitrestriktion die Kosten eines Individuums. Je höher der Wert der Kostenfunktion einer Lösungsrepräsentation desto geringer ist die Wahrscheinlichkeit den Selektionsprozess zu überstehen.

Die nachfolgende Tabelle 4.2 enthält eine Übersicht aller Größen, welche in die Berechnung des Fitnesswertes eines Individuums mit einfließen:

Variable	Beschreibung
t_{travel}	Gesamtzeit aller m Touren
z_{fail}	Anzahl der Zeitfensterverletzungen
z_{failAbs}	Summe der Abweichungen von den Zeitfenstern
a_{fail}	Anzahl der Arbeitszeitverletzungen
a_{failAbs}	Summe der Abweichungen von den Arbeitszeiten
key_{fail}	Anzahl der Schlüsselverletzungen

Tabelle 4.2: Sämtliche Größen zur Berechnung des Fittnesswertes einer Tour

Eine dieser Größen ist die bereits angesprochene Fahrzeit t_{travel}, welche die kumulierten Fahrzeiten aller Fahrer (in Minuten) enthält. Weiterhin werden die Verletzungen der Nebenbedingungen in den dafür vorgesehenen Variablen z_{fail}, a_{fail} und key_{fail} gezählt. Diese geben die Anzahl der Zeitfensterverletzungen, Arbeitszeitverletzungen sowie der Schlüsselverletzungen an.
Zusätzlich zu den nichterfüllten Zeitfenster- und Arbeitszeitbedingungen werden noch die Variablen z_{failAbs} und a_{failAbs} ermittelt. Diese enthalten die Summe der Abweichungen (in Minuten) von den dafür vorgesehenen Zeitfenstern. Eine Abweichung gibt an, wie weit eine nicht erfüllte Kontrolle vom eigentlichen Zeitfenster entfernt ist. Durch Angabe dieses Wertes hat man eine feinere Granularität und leitet das Optimierungsverfahren eher in die richtige Richtung, anstatt nur Angaben darüber zu erhalten wie viele Zeitbedingungen verletzt sind.
Diese Größen betrachtet man für jedes beliebige Individuum und berechnet daraus den Fitnesswert. Dabei ergibt sich nach dem Aggregationsansatz folgende allgemeine Darstellung der Fitnessfunktion F:

$$F(i) = w_1 \cdot t_{\text{travel}} + w_2 \cdot z_{\text{fail}} + w_3 \cdot z_{\text{failAbs}} + w_4 \cdot a_{\text{fail}} + w_5 \cdot a_{\text{failAbs}} + w_6 \cdot key_{\text{fail}}$$

Offensichtlich handelt es sich bei der Fitnessfunktion F um eine Kostenfunktion, welche es zu minimieren gilt. In die Berechnung dieser Kostenfunktion gehen neben den Kosten für die Fahrzeit auch sämtliche Restriktionen mit einer entsprechenden Gewichtung ein. Die Verletzung einer Zeitrestriktion führt demnach zu höheren Kosten und verringert die Überlebenswahrscheinlichkeit eines Individuums. Eine geschickte Wahl der Gewichte $w_1, w_2, \ldots w_6$ ist Voraussetzung für eine gute Leistungsfähigkeit des Genetischen Algorithmus und wird im Kapitel 6 näher analysiert.

4.3 Repräsentation des Straßennetzes

Wie bereits in Kapitel 2.3.3 beschrieben, ist die Angabe der Entfernungen zwischen den einzelnen Wachobjekten eine wichtige Grundlage für die rechnergestützte Tourenplanung. Dies ist verständlich, denn der Rechner benötigt in irgendeiner Art und Weise Informationen über das vorhandene Straßennetz oder zumindest näherungsweise Angaben zu den Entfernungen und Fahrzeiten zwischen allen Objekten.
Ein geeignetes Lösungsverfahren braucht schließlich Kenntnis darüber, welche Objekte dicht beieinander liegen oder weit entfernt sind. Dieses Wissen wird z.B. dazu verwendet, um benachbarte Objekte möglichst von ein und demselben Fahrer nacheinander kontrollieren zu lassen und somit lange Fahrzeiten zu vermeiden.

Zur Lösung des Problems benötigt man demzufolge die kürzesten Entfernungen zwischen den einzelnen Wachobjekten in Form eines vollständigen Graphen[4]. Dabei werden alle Wachobjekte als Knoten betrachtet, wobei jeder Knoten mit jedem anderen durch genau eine Kante verbunden ist. Die Kantengewichte w_{ij} entsprechen jeweils den kürzesten Entfernungen zwischen einem beliebigen Objekt i zu jedem anderen Objekt j. Es ergibt sich auf diese Weise eine Entfernungsmatrix M_{ij}, die sämtliche Distanzen und Fahrzeiten zwischen allen n Wachobjekten und dem Fahrzeugdepot (0) enthält.

[4]siehe Kapitel 2.2

$$M_{i,j} = \begin{pmatrix} w_{0,0} & w_{0,1} & \dots & w_{0,n} \\ w_{1,0} & w_{1,1} & \dots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n,0} & w_{n,1} & \dots & w_{n,n} \end{pmatrix}$$

Die Elemente der Matrix sind die Kantengewichte $w_{ij} = (d_{ij}, t_{ij})$, wobei d_{ij} die Entfernung (*distance*) und t_{ij} die benötigte Fahrzeit (*travel time*) zwischen dem Knoten i und j angibt.
Zur Vereinfachung geht man von einem ungerichteten Graphen aus, bei dem nicht unterschieden wird zwischen den Strecken $\vec{AB}$ und $\vec{BA}$. Die Entfernungsmatrix ist demzufolge symmetrisch. Außerdem entspricht jedes Element der Hauptdiagonalen dem Wertepaar $(0, 0)$:

$$\forall i, j : (i = j) \Longrightarrow w_{ij} = (0, 0),$$

da die Entfernung von einem Objekt zu sich selbst null beträgt.

Zur Implementierung der Entfernungsmatrix in C++ wurde eine Klasse `StreckenClass` definiert. Diese enthält zwei Arrays `min[][]` und `km[][]` der Größe 100×100, darin können die Entfernungen zwischen maximal 100 verschiedenen Objekten aufgenommen werden.

```
class StreckenClass
{
public:
    unsigned int min[100][100];
    double km[100][100];
};
```

Die Bereitstellung der Entfernungen zwischen den Objekten stellt einen wichtigen Kernpunkt in der Modellierung der vorhandenen Problemstellung dar und ist unabdingbar für die Generierung und Evaluierung der Tourenpläne. Ein Wunsch wäre es natürlich automatisiert auf einen der unzähligen Routenplaner zuzugreifen, die bereits kostenlos im Internet zur Verfügung stehen. Doch leider unterstützen diese lediglich eine manuelle Eingabe von einem Start- und Zielort und berechnen dann die optimale Route zwischen beiden. Auch kommt man nicht so einfach an die dahinterliegenden Datenbanken heran, um sie für eigene Zwecke nutzen zu können. Darum sollen in den nächsten

Abschnitten drei mögliche Ansätze dargestellt werden, wie man dennoch eine Entfernunsmatrix M_{ij} aufstellen kann.

4.3.1 Ermittlung der Luftlinienentfernung

Die einfachste Möglichkeit die Entfernungen zwischen den einzelnen Wachobjekten zu ermitteln ist, die *Luftlinienentfernung* zu messen. Dazu trägt man zunächst alle zu bewachenden Kunden in eine Landkarte ein und nimmt den euklidischen Abstand zwischen den einzelnen Objekten (dies entspricht der Luftlinienentfernung).

Zur Veranschaulichung ist die kürzeste Entfernung zwischen zwei gegebenen Objekten in Abbildung 4.11 angegeben. Dabei ist die Luftlinienentfernung links und die tatsächlich benötigte Entfernung, unter Berücksichtigung des zugrundeliegenden Straßennetzes, rechts dargestellt.

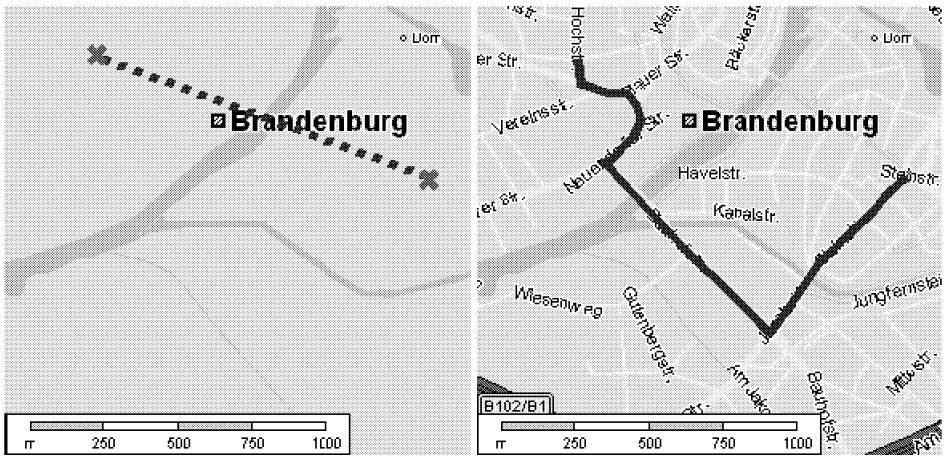


Abbildung 4.11: Luftlinienentfernung (links) und tatsächliche Entfernung (rechts) zwischen zwei Objekten

Diese Vorgehensweise erfordert natürlich die Abspeicherung der X- bzw. Y-Koordinate eines jeden Objekts. Jedoch entspricht diese Methode lediglich einer sehr groben Schätzung der tatsächlichen Distanz. Dadurch ergeben sich einige Nachteile und Ungenauigkeiten, denn Informationen über das vorhandene Straßennetz (Landstraße, Autobahn, Ortschaften) werden nicht berücksichtigt.

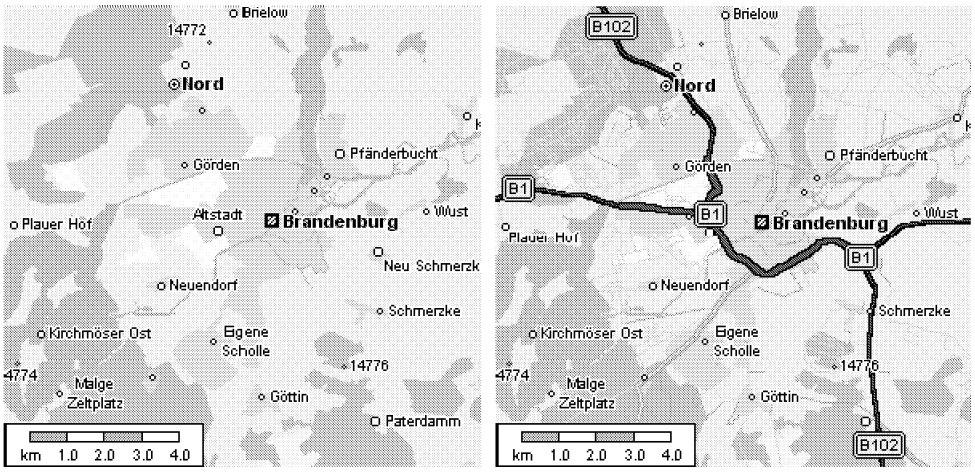


Abbildung 4.12: Darstellung einer Landkarte ohne Berücksichtigung des Straßennetzes (links) und mit Straßennetz (rechts)

Deshalb wird der euklidische Abstand eher bei theoretischen Arbeiten verwendet und ist für den praktischen Einsatz ungeeignet. Abbildung 4.12 macht die Problematik dieses Ansatzes noch einmal deutlich.

4.3.2 Eingabe der gesamten Entfernungsmatrix

Ein weiterer Ansatz zur Bereitstellung der Entfernungen zwischen allen Objekten, ist die Aufstellung eines vollständigen Graphen mit Hilfe eines Routenplaners. Dabei werden die kürzesten Entfernungen d_{ij} und Fahrzeiten t_{ij} zwischen jedem beliebigen Objekt i zu allen anderen Objekten j unter Verwendung eines Routenplaners (z.B. *www.reiseplanung.de*) oder intuitiv ermittelt. Dies entspricht der Eingabe der gesamten Entfernungsmatrix M_{ij}.

Der Vorteil liegt gegenüber dem ersten Ansatz klar auf der Hand: Man kann auf die tatsächlichen Entfernungen und Fahrzeiten zugreifen, in denen die Informationen des Straßennetzes berücksichtigt sind (die Genauigkeit ist viel größer). Der Nachteil dieses Ansatzes wird deutlich, wenn man sich die Formel 2.1 für die Berechnung der Anzahl der Bögen in einem vollständigen Graphen nochmals vor Augen hält (siehe Abschnitt 2.2).

Für n Kunden plus Depot ergibt sich ein vollständiger Graph mit $k = n + 1$ Knoten. Wie zeitaufwendig dieser Vorgang bei wachsenden Knoten k werden kann, veranschaulicht die Darstellung dieser Funktion in Abbildung 4.13.

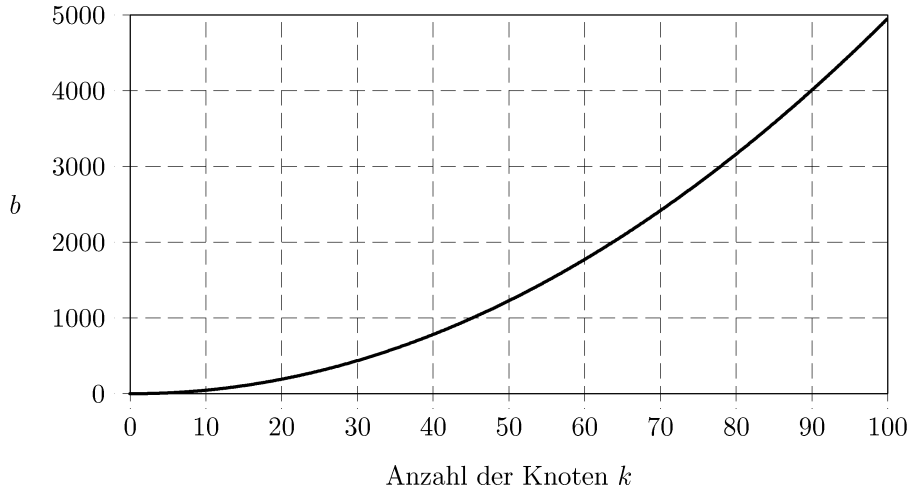
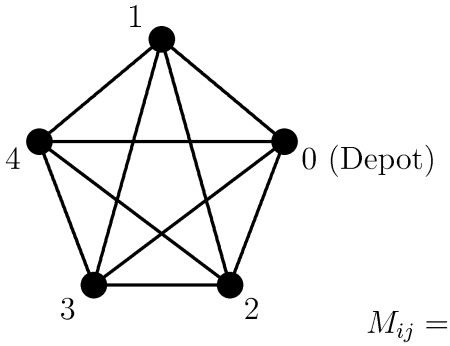


Abbildung 4.13: Anstieg der Verbindungen b in einem vollständigen Graphen, bei wachsender Anzahl der Knoten k

Ein Beispiel für ein Kundennetz mit 4 Kunden und der dazugehörigen Entfernungsmatrix M_{ij} ist in der Abbildung 4.14 dargestellt.



$$M_{ij} = \begin{pmatrix} (0,0) & w_{0,1} & w_{0,2} & w_{0,3} & w_{0,4} \\ w_{1,0} & (0,0) & w_{1,2} & w_{1,3} & w_{1,4} \\ w_{2,0} & w_{2,1} & (0,0) & w_{2,3} & w_{2,4} \\ w_{3,0} & w_{3,1} & w_{3,2} & (0,0) & w_{3,4} \\ w_{4,0} & w_{4,1} & w_{4,2} & w_{4,3} & (0,0) \end{pmatrix}$$

Abbildung 4.14: Ein vollständiger Graph mit 4 Kunden plus einem Fahrzeugdepot und der dazugehörigen Entfernungsmatrix M_{ij}

Aufgrund der Symmetrie der Entfernungsmatrix genügt es, nur die Elemente oberhalb oder unterhalb der Hauptdiagonalen anzugeben und diese dann entsprechend zu spiegeln. Dadurch braucht man nur knapp die Hälfte der Entfernungsmatrix einzugeben, trotzdem ist der Arbeitsaufwand bei einer größeren Anzahl an Kunden schon sehr hoch.

Bei dem zu untersuchenden Nachtrevier mit 56 Wachobjekten (plus ein Depot) ergibt sich also ein vollständiger Graph mit 57 Knoten und 1596 Bögen! Das entspricht schon einem erheblichen Zeitaufwand.
Beim Hinzufügen eines neuen Knotens, muss ausgehend von diesem zu jedem bereits bestehenden Knoten die kürzeste Verbindung ermittelt und eingetragen werden.

4.3.3 Abbildung des Straßennetzes von Brandenburg und Umgebung

Natürlich sollte im Rahmen dieser Diplomarbeit das „Rad nicht völlig neu erfunden“ und eigenhändig das Straßennetz auf dem Rechner abgebildet werden, zumal es bereits unzählige Routenplaner auf dem Markt gibt, die genau das schon implementiert haben. Doch wie bereits erwähnt, kann man auf diese Daten nicht in der Weise zugreifen, wie man es gerne für seine eigenen Zwecke hätte. Leider erlauben all diese Programme nur eine manuelle Eingabe zwischen Start- und Zielort und liefern daraufhin die kürzeste Entfernung. Auf dieser Methode basiert Ansatz 2, welcher zunächst in das Programm implementiert wurde.
Doch die Erfahrungen und Gespräche mit den Mitarbeitern der Safe Wachschutz/Allservice GmbH haben gezeigt, dass es unzumutbar ist, die Entfernungsmatrix manuell aufzustellen, zu erweitern und zu pflegen.
Die Dynamik der Wachschutzaufträge ist sehr hoch, so ändert sich fast wöchentlich (manchmal sogar täglich) die Anzahl der zu kontrollierenden Wachobjekte. Besonders das Hinzufügen eines neuen Wachobjekts kann aufgrund der Angabe sämtlicher Entfernungen zu allen bereits vorhandenen Objekten sehr langwierig werden. Hier besteht die große Gefahr, dass die Mitarbeiter vor solchen immensen Aufwand zurückschrecken und wieder ihren althergebrachten Weg gehen würden.
Deshalb wurde (nach gründlichen Überlegungen) entschieden, zumindest das Straßennetz der Stadt Brandenburg a. d. Havel mit den umliegenden Städten und Gemeinden in vereinfachter Form auf dem Rechner nachzubilden. Ähnlich wie beim ersten Ansatz platziert der Benutzer das Objekt in einer Landkarte, diese entspricht einer Repräsentation des vorhandenen Straßennetzes in Form eines *gerichteten bewerteten Graphen*[5]. Dabei beschreiben die Knoten des Graphen Landmarken, wie z.B. Kreuzungen oder Kurven. Die Streckenabschnitte zwischen diesen Knoten werden durch die Kanten beschrieben.

[5]siehe Abschnitt 2.2

Eine Kante besitzt folgende Eigenschaften:

- Id's der beiden zugehörigen Straßenknoten
- durchschnittliche Geschwindigkeit (wichtig zur Ermittlung der Fahrzeit)
- gerichtet / ungerichtet (um auch Einbahnstraßen berücksichtigen zu können)
- Farbe (für graphische Darstellung, um z.B. Autobahn, Landstraße, etc. zu unterscheiden)
- Fahrzeit (in Minuten)
- Länge (in Kilometer)

Zur Beschreibung des Straßennetzes als Graph werden die folgenden zwei Klassen `StrassenKantenClass` und `StrassenKnotenClass` definiert:

```
class StrassenKantenClass
{
public:

    unsigned long K1,K2; //Id's der Knoten
    unsigned int V; //Geschwindigkeit
    unsigned int F; //Farbe
    bool R; //Richtung
    double min,km; //Fahrzeit u. Länge
};

class StrassenKnotenClass
{
public:

    long X,Y; //Position
    double dist_min,dist_km; //Distanz zum Startknoten
    int parent; //Vorgängerknoten
};
```

Ein Straßenknoten enthält neben seiner Position (`X,Y`) auch noch die Einträge `dist_min`, `dist_km` und `parent`, die für den Algorithmus zur

Bestimmung der kürzesten Wege innerhalb dieses Graphen benötigt werden. Eine genauere Beschreibung dieses Algorithmus erfolgt im nachfolgenden Abschnitt.

Die Farben und Beschreibungen der Streckenabschnitte sind nach Tabelle 4.3 festgelegt:

Wert(`F`)	Beschreibung	Farbe
0	*Weg*	*grau*
1	*normale Straße*	*weiß*
2	*Hauptstraße*	*gelb*
3	*Bundesstraße*	*rot*
4	*Autobahn*	*gelb*

Tabelle 4.3: Festlegung der Streckenabschnitte

Beschreibung der Schnittstellen

Die entsprechenden Daten und Informationen des Straßennetzes, wie z.B. Knoten, Kanten und Bezeichnungen, liegen in unterschiedlichen Textdateien vor. Jede einzelne Datei hat ein fest vorgegebenes Format. Diese Dateien bilden die Schnittstelle mit dem Programm und können beliebig erweitert werden, um beispielsweise eventuelle Veränderungen des Straßennetzes zu berücksichtigen oder aber völlig neue Umgebungen zu erschaffen.

Das Straßennetz wird durch folgende Dateien beschrieben (siehe Tabelle 4.4), dessen Aufbau sich übrigens im Anhang A einsehen lässt.

Dateiname	Beschreibung
Knoten.dat	enthält sämtliche Straßenknoten
Kanten.dat	Angabe und Beschreibung sämtlicher Streckenabschnitte
Bezeichnung.dat	Angabe aller Orts- bzw. Straßennamen
BRB.dat	Stadtgrenzen von Brandenburg a.d. Havel in Form eines Polygons

Tabelle 4.4: Dateien zur Beschreibung des Straßennetzes

Ausgehend von diesen Betrachtungen ergibt sich für das Straßennetz von Brandenburg und Umgebung ein Graph mit ca. 2000 Knoten und 3000 Straßenabschnitten (siehe Abbildung 4.15).

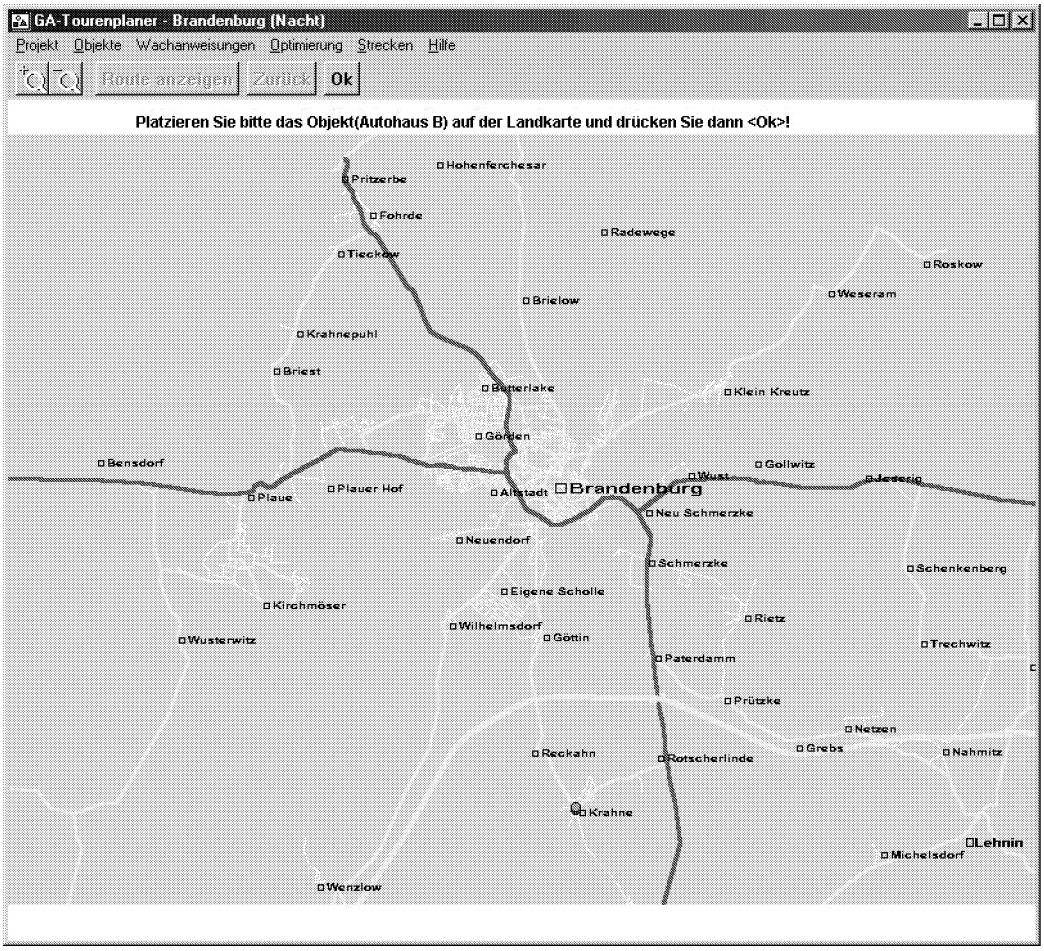


Abbildung 4.15: Straßennetz von Brandenburg und Umgebung

Dadurch wurde für den Anwender eine komfortable Eingabemöglichkeit geschaffen, um neue Objekte anzulegen oder zu bearbeiten. Die Standorte werden dazu einfach in die generierte Landkarte eingezeichnet und daraufhin wird der Straßenknoten bestimmt, der am dichtesten am eingetragenen Objekt liegt.

Von diesem Straßenknoten ausgehend, sind nun die kürzesten Entfernungen und benötigten Fahrzeiten zu allen anderen Objekten gemäß den vorhan-

denen Teilstrecken und angegebenen Durchschnittsgeschwindigkeiten zu bestimmen. Da sich das Straßennetz allgemein als bewerteter Graph darstellt, kann man auf bekannte Probleme der Graphentheorie zurückgreifen.

Man unterscheidet:

- *single-pair-shortest-path Probleme*: Suchstrategien für den kürzesten Weg von einem festen Start- zu einem festen Zielknoten (Tiefensuche, Breitensuche, Gleiche-Kosten Suche, A*-Algorithmus (vgl. [HS99]))
- *single-source-shortest-paths Probleme*: finden für einen Graphen mit bewerteten Kanten von einem gegebenen Knoten aus die kürzesten Wege zu allen anderen Knoten (z.B. Dijkstra-Algorithmus, Algorithmus von Moore und Ford (vgl. [Zh01])
- *all-pairs-shortest-paths Probleme*: bei denen die kürzesten Verbindungen zwischen allen Knoten des Graphen bestimmt werden (z.B. Floyd-Warshall-Algorithmus (vgl. [Cr99]))

Bei dem Algorithmus, welcher in dieser Arbeit implementiert wurde, handelt es sich um den Algorithmus von Dijkstra, welcher im nachfolgenden Abschnitt näher beschrieben wird.

Algorithmus von Dijkstra

Professor Edsger Wybe Dijkstra, Professor für Mathematik und Computer-Wissenschaften, fand 1959 einen Algorithmus, welcher das kürzeste Weg Problem (shortest path problem) löst (vgl. [Be01]). Dieser bestimmt ausgehend von einem Startknoten die kürzesten Verbindungen zu allen mit ihm verbundenen Knoten des Graphen. Angefangen mit dem Startknoten werden dazu die einzelnen Knoten sukzessive expandiert. Dabei führt man eine Liste der Knoten (Agenda), welche gerade in Bearbeitung sind. Beim Dijkstra-Algorithmus wird in jedem Knoten die zurückgelegte Distanz zum Startknoten sowie ein Verweis zum Vorgänger-Knoten (parent) gespeichert. Durch letzteren kann man dem Weg zum Startknoten zurückverfolgen. Für jeden Nachfolger-Knoten wird verglichen, ob die Distanz über den gerade expandierten Knoten kürzer ist, als eine vorher ermittelte. Ist dies der Fall, so wird die alte Distanz überschrieben und der aktuelle Knoten als parent des Nachfolger-Knotens gesetzt.

Die Funktionsweise von Dijkstra's Algorithmus kann formal wie folgt beschrieben werden:

Gegeben ist ein Graph $G = (V, E)$, wobei

$$V = \{v_i | i = 1, \ldots, n\}$$

die Knoten und

$$E = \{e_{ij} = \langle v_i, v_j \rangle | v_i, v_j \in V\}$$

die Kanten repräsentieren. Dabei besitzt jede Kante $e_{ij} \in E$ eine Wichtung $w_{ij} := w(e_{ij}) \in \mathbb{R}$, welche z.B. die Entfernung oder Fahrzeit zwischen den beiden Knoten v_i und v_j angibt.
Ausgehend von einem Startknoten v_{Start} seien nun die kürzesten Wege zu allen anderen Knoten v_k des Graphen gesucht. Eine Menge $A \subset V$ (Agenda) enthält die Knoten, welche sich gerade in Bearbeitung befinden. Für jeden Knoten v wird in $v.pre \in V$ der Vorgänger und in $v.dist \in \mathbb{R}$ die Distanz zum Startknoten v_{Start} gespeichert. Die Menge $v.Adj \subset V$ enthält sämtliche Nachfolger eines Knoten v.

Die Funktionsweise des Algorithmus ist im folgenden Pseudo-Code dargestellt (vgl. [Be01]).

PseudoCode:

gegeben:

$v_{\text{Start}} \in V$	der Startknoten
$w_{ij} \in \mathbb{R}$	Kantengewichte
$v.Adj \subset V$	Menge der Nachfolger eines Knoten v
$v.dist \in \mathbb{R}$	Distanz zum Startknoten
$v.pre \in V$	Vorgänger-Eintrag
$v, v_k \in V$	bestimmte Knoten
$d \in \mathbb{R}$	lokale Variable zur Berechnung der neuen Distanz
$A \subset V$	die Agenda

```
Initialisierung:
    FOREACH v ∈ V DO
        v.pre ← NULL                          initialisiere Vorgängereinträge
        v.dist ← ∞                            und Distanzen
    END FOREACH
    v_Start.pre ← v_Start                     Initialisierung des Startknotens
    v_Start.dist ← 0
    A ← {v_Start}                             Agenda initialisieren
Hauptteil:
    WHILE A ≠ ∅ DO
        v_k ← wähle v ∈ A ∧ v.dist = min       Knoten aus Agenda wählen
        A ← A \ {v_k}                         und aus der Agenda entfernen
        expand(v_k)                           expandiere Knoten v_k
        FOREACH v ∈ v_k.Adj DO
            d ← v_k.dist + w_{v_k,v}          Berechne neue Distanz
            IF v.pre = NULL DO                Wenn Nachfolger neu
                v.pre ← v_k                   initialisiere Nachfolger
                v.dist ← d
                A ← A ∪ {v}                   Knoten in Agenda aufnehmen
            ELSE DO                           Nachfolger bereits besucht
                IF d < v.dist DO              Wenn neuer Weg kürzer
                    v.pre ← v_k               Nachfolger reinitialisieren
                    v.dist ← d
                    A ← A ∪ {v}               erneut in Agenda aufnehmen
                END IF
            END IF
        END FOREACH
    END WHILE
```

Zunächst einmal werden in der Initialisierungsphase die Distanzen sämtlicher Knoten auf den Wert ∞ gesetzt sowie die Vorgänger-Einträge auf `NULL`. Einzige Ausnahme bildet der Startknoten v_{Start}, welcher sich selbst als Vorgänger besitzt und somit die Distanz „0“ hat. Mit diesem Startknoten wird weiterhin die Agenda A initialisiert.
Nun folgt der Hauptteil des Algorithmus, bei dem solange eine Schleife durchlaufen wird, bis schließlich die Agenda leer ist. Ist die Agenda nicht leer, so wird ein Knoten v_k aus der Liste ausgewählt, welcher die kleinste Distanz zum Startknoten besitzt. Dieser Knoten v_k wird daraufhin aus der Agenda entfernt und expandiert. Dabei werden alle seine Nachfolger-Knoten bestimmt

und in eine Nachfolger-Liste $v_k.Adj$ gespeichert. Für jeden Nachfolgerknoten $v \in v_k.Adj$ wird nun zunächst die Distanz $d \in \mathbb{R}$ vom Startknoten über den Knoten v_k bestimmt. Danach wird für den Nachfolger-Knoten v überprüft, welcher der beiden folgenden Fälle auftritt:

1. Der Knoten v wurde noch nicht besucht und hat noch keinen Vorgänger ($v.pre = \texttt{NULL}$, $v.dist = \infty$), dann erhält dieser Knoten als Vorgänger-Eintrag den aktuellen Knoten ($v.pre \leftarrow v_k$) und die sich darüber ergebende Distanz ($v.dist \leftarrow d$).

2. Der Nachfolgerknoten v hat bereits einen Vorgänger-Eintrag mit einer entsprechenden Distanzangabe ($v.pre \neq \texttt{NULL} \wedge v.dist \neq \infty$). In diesem Fall muss der alte Distanzeintrag $v.dist$ verglichen werden mit der aktuellen Distanz d über den Knoten v_k. Ist die Verbindung über den aktuellen Knoten kürzer als die bisher ermittelte, so werden die alten Einträge für Distanz und Vorgänger mit den aktuellen Werten überschrieben.

In beiden Fällen wird daraufhin der Nachfolgerknoten v in die Agenda A eingefügt. So kann es vorkommen, dass ein Knoten mehr als einmal expandiert wird, wenn später eine kürzere Verbindung über einen anderen Knoten gefunden wird. Dies ist auch notwendig, denn die Veränderung der Einträge des Knotens v haben wiederum Einfluss auf all seine Nachfolgerknoten.

Am Ende des Algorithmus besitzt jeder Knoten v einen Wert $v.dist$, welcher die Distanz des kürzesten Weges vom Startknoten v_{Start} angibt. Aufgrund der Vorgänger-Verweise $v.pre$ kann man für jeden Knoten v_{Ziel} den Weg bis zum Startknoten zurückverfolgen.

```
v ← v_Ziel
DO
   gib aus v
   v = v.pre
WHILE v ≠ v.pre DO
```

Auf diese Weise kann man nun die Entfernungsmatrix zwischen allen k Objekten bestimmen, indem man den Algorithmus k-mal durchläuft. Dabei wird jedes Objekt genau einmal als Startknoten v_{Start} gesetzt und nach Ende jedes Durchlaufs die Entfernungen zu allen anderen Objekten in die Entfernungsmatrix M eingetragen.

4.4 Entwicklungsumgebung Borland C++

Im Rahmen dieser Diplomarbeit wurde die Entwicklungsumgebung von Borland C++ 5.0 verwendet, um ein Programm zur automatischen Tourenplanung mit einer benutzerfreundlichen Oberfläche zu erstellen.
Borland C++ ist ein sehr leistungsfähiger Compiler, mit dem man Programme für DOS, Windows 9x, 2000, XP & NT erstellen kann. Mit der *Object Windows Library* (OWL) stellt Borland eine leistungsstarke Klassenbibliothek zur Verfügung. Sie deckt weitgehend alle Bereiche der Windows-Programmierung ab und abstrahiert beinahe die gesamte Windows-API (*Application Programming Interface*).
Durch die Verwendung der objektorientierten Programmierung, ist es möglich auf relativ einfache Art und Weise Windows-Anwendungen zu erstellen. Dies geschieht durch Verwendung der bereitgestellten Klassen und Objekte aus den Bibliotheken, die entweder direkt benutzt oder nach eigenen Bedürfnissen erweitert werden. Die folgende Abbildung 4.16 zeigt den Aufbau einer Windows-Anwendung in vereinfachter Form.

Abbildung 4.16: Aufbau einer Windows-Anwendung in vereinfachter Form

4.5 Der allgemeine Programmablauf

Der allgemeine Programmablauf ist anhand eines *Funktionsbaums* (vgl. [Ba01]) in Abbildung 4.17 dargestellt. Ein Funktionsbaum gibt einen Überblick über die verwendeten Funktionen des Programms.

Abbildung 4.17: Darstellung eines Funktionsbaums

Zunächst einmal besteht die Möglichkeit, den gesamten Wachschutzprozess in mehrere *Projekte* zu unterteilen. Ein Projekt stellt eine abgeschlossene Einheit dar, welches die Aufgabe hat, zusammengehörige Wachaufträge(z.B. das Nachtrevier, Tagrevier etc.) zu vereinen.
Dadurch sollen inhaltlich und zeitlich nicht zusammengehörige Wachanweisungen voneinander getrennt werden können. Das später aufsetzende Optimierungsverfahren wird sämtliche definierte Wachanweisungen eines Projekts berücksichtigen, deshalb macht es Sinn von vornherein die Reviere in unterschiedliche Projekte zu fassen.

Die Verwaltung der einzelnen Projektdaten erfolgt in den 5 Unterpunkten:

- Projektdaten
- Wachobjekte
- Wachanweisungen
- Optimierung
- Streckenangaben

Im Unterpunkt *Projektdaten* erfolgt die Verwaltung der Projektdaten. So kann hier beispielsweise ein neues Projekt angelegt, ein bereits bestehendes Projekt geladen bzw. das aktuelle Projekt gespeichert werden.

Im zweiten Auswahlpunkt *Wachobjekte* besteht die Möglichkeit, die Wachobjekte eines bestimmten Projekts zu verwalten. Dazu gehören das Anlegen neuer Objekte, das Aktualisieren bestehender Objekte sowie das Löschen von Objekten.

Die Verwaltung der Wachanweisungen erfolgt im Punkt drei *Wachanweisungen*. Hier können zum einen neue Wachanweisungen definiert, aber auch bereits vorhandene angepasst oder gelöscht werden.

Der Unterpunkt *Optimierung* dient dazu, das Optimierungsverfahren (*Genetischer Algorithmus*) zu starten. Außerdem können hier die Einstellungen des Optimierungsverfahrens, z.B. Gewichtungen der Zielkriterien, Anzahl der Generationen, Reproduktions-, Mutations- und Crossoverraten usw. verändert werden. Nach dem Ablauf des Verfahrens wird anschließend die beste gefundene Lösung auf dem Bildschirm ausgegeben.

Im letzten Punkt *Streckenangaben* können die Einträge der Entfernungsmatrix M_{ij} eingegeben oder verändert werden.

4.6 Das Klassendiagramm

In diesem Abschnitt soll eine kurze Übersicht über die verwendeten Klassen erfolgen. Dabei werden lediglich nur kurz einige Auszüge beschrieben (eine komplette Beschreibung des gesamten Quelltextes würde den Rahmen dieser Diplomarbeit sprengen).
Ziel war es, das Programm in Form einer Windows-Anwendung zu erstellen. Dazu stellt Borland die *Object Windows Library* (*kurz*: OWL) zur Verfügung. Die OWL ist zu vergleichen mit den *Foundation Classes* von Microsoft und enthält eine Vielzahl an Klassen, mit denen sich Windows-Anwendungen relativ einfach programmieren lassen.

Alle Windows-Programme sind in weiten Teilen gleich aufgebaut. Eine Windows-Oberfläche besteht zum einen aus dem sogenannten *Rahmen* (Frame), dieser enthält z.B. die Titelzeile, Menü- und Symbolleisten etc. und zum anderen aus dem eigentlichen *Fensterinhalt* (Window). Für diese Komponenten stellt die OWL die beiden Basisklassen `TDecoratedFrame` und `TWindow` zur Verfügung.

Die Klasse `TDecoratedFrame` stellt Methoden zur Erzeugung und Darstellung eines Rahmenfensters bereit. Sie ist u.a. zuständig für das Setzen und Manipulieren von Menüeinträgen sowie zur Erstellung von Werkzeug- und Statusleisten.

Die Fensterklasse `TWindow` ist u.a. zuständig für das Zeichnen des Fensterinhalts. So enthält diese Klasse z.B. die `Paint`-Methode welche für das Neuzeichnen des Fensters verantwortlich ist. Hier wird auch die *Anworttabelle* für die Behandlung der Fensternachrichten hinterlegt.

Zur Erzeugung einer Windows-Applikation werden ausgehend von einem sogenannten *Anwendungsgerüst* Instanzen dieser beiden Klassen angelegt. Die Klasse `TApplication` stellt ein solches Anwendungsgerüst bereit, auf welches man als Programmierer zugreifen und individuell nach eigenen Bedürfnissen anpassen kann. Die Anwendungsklasse `TApplication` repräsentiert praktisch schon ein voll lauffähiges Windows-Programm, jedoch ohne weitere Funktionalität.
Um das Anwendungsgerüst nach seinen Bedürfnissen anzupassen, erstellt man eine von `TApplication` abgeleitete Klasse `TMyApp` und erweitert das von `TApplication` gelieferte Standardgerüst.

```
class TMyApp : public TApplication
{
public:
   TMyApp() : TApplication() {}
   TControlBar* ControlBar;
   void InitMainWindow()
   {
     TWindow* testWindow = new TMyWindow;
     TMyFrameWindow* frame =
            new TMyFrameWindow(0, APP_NAME, testWindow, true);
     ControlBar = new TControlBar(frame);
     ControlBar->Insert(*new TButtonGadget(IDB_ZOOMPLUS, IDB_ZOOMPLUS));
     ControlBar->Insert(*new TButtonGadget(IDB_ZOOMMINUS, IDB_ZOOMMINUS));
     ControlBar->Insert(*new TSeparatorGadget(10));
     ControlBar->Insert(*new TButtonGadget(IDB_ROUTE_ANZ, IDB_ROUTE_ANZ));
     ControlBar->Insert(*new TSeparatorGadget(6));
     ControlBar->Insert(*new TButtonGadget(IDB_ZURUECK, IDB_ZURUECK));
     ControlBar->Insert(*new TSeparatorGadget(6));
     ControlBar->Insert(*new TButtonGadget(IDB_OK, IDB_OK));
     frame->Insert(*ControlBar, TMyFrameWindow::Top);
     SetMainWindow(frame);
     GetMainWindow()->AssignMenu(MENU_1);
   }
};
```

Zuerst wird der Konstruktor angegeben. Dieser kann leer bleiben, da alle notwendigen Schritte zur Erzeugung des Anwendungsgerüstes bereits vom Konstruktor der Basisklasse `TApplication` ausgeführt werden. Da die Anwendung über eine Symbolleiste verfügen soll, wird daraufhin ein Objekt der Klasse `TControlBar` angelegt.

Die wichtigste Klasse des Programms ist die Fensterklasse `TMyWindow`, welche von der Klasse `TWindow` abgeleitet ist und entsprechend den Anforderungen angepasst wurde. Eine weitere wichtige Klasse ist die Klasse `TMyFrameWindow`, welche von `TDecoratedFrame` abgeleitet ist. Sie beschreibt neben der Rahmenart auch die anfängliche Position und Größe der Anwendung auf dem Bildschirm.

```
class TMyFrameWindow : public TDecoratedFrame
{
public:
   TMyFrameWindow (TWindow* parent, const char far *title = 0,
            TWindow *clientWnd = 0,
            BOOL shrinkToClient = FALSE)
      : TDecoratedFrame (parent, title, clientWnd, FALSE)
      {
         Attr.W = GetSystemMetrics(SM_CXSCREEN);
         Attr.H = GetSystemMetrics(SM_CYSCREEN);
         Attr.X = (GetSystemMetrics(SM_CXSCREEN) - Attr.W);
         Attr.Y = (GetSystemMetrics(SM_CYSCREEN) - Attr.H);
      }
};
```

Innerhalb der Klasse `TMyApp` werden nun Instanzen von `TMyWindow` und `TMyFrameWindow` angelegt. Damit ist im Grunde das Programmgerüst fertig, bleibt nur noch die Belegung der Symbolleiste (`TControlBar`) mit entsprechenden Objekten der Klassen `TButtonGadget` und `TSeperatorGadget`. Dabei beschreibt die Klasse `TButtonGadget` Schaltelemente (vergleichbar mit Buttons) in Werkzeugleisten, während `TSeperatorGadget` Abstände zwischen diesen Schaltelementen realisiert.

Zum Schluss wird der Anwendung noch das Menü zugewiesen, welches durch den Bezeichner `MENU_1` innerhalb einer Ressourcen-Datei festgelegt ist.
Die Ressourcen-Datei `GA-Tourenplaner.rc` enthält neben der Beschreibung des Menüs, auch Bilder, Mauszeiger und sämtliche Dialoge der Anwendung. Neben dieser Ressourcen-Datei gehören auch noch das Hauptprogramm `GA-Tourenplaner.cpp` sowie zahlreiche Headerdateien, welche überwiegend die Klassenbeschreibungen enthalten, zur Projektdatei `GA-Tourenplaner.ide` von Borland C++ . Eine Übersicht aller Dateien ist in der folgenden Abbildung 4.18 dargestellt.

Abbildung 4.18: Überblick der verwendeten Projektdateien

Der Einstiegspunkt des Programms befindet sich am Ende der Datei `ga-tourenplaner.cpp`. Dort wird innerhalb der `OwlMain`-Funktion eine Instanz der Klasse `TMyApp` erzeugt und ausgeführt.

```
//########################################
// M A I N
//########################################
int OwlMain(int, char**)
{
    return TMyApp().Run();
}
```

Wie bereits oben angedeutet, handelt es sich bei der Klasse `TMyWindow` um das Kernstück des Programms. Diese enthält u.a. auch eine angepasste Antworttabelle für das Auftreten von Windows-Nachrichten. Windows ist ein ereignisorientiertes Betriebsystem, d.h. für jedes Fenster können verschiedene Ereignisse, wie z.B. die Auswahl eines bestimmten Menüpunktes, die Betätigung der Maustasten oder aber das Vergrößern- und Verkleinern des Fensters, auftreten.

Sobald ein bestimmtes Ereignis in einem Fenster auftritt, wird eine Windows-Nachricht an das jeweilige Fenster geschickt. Das Fenster muss dann bereit sein, auf solche Nachrichten zu reagieren. Dazu dient die sogenannte Antworttabelle.

In der Antworttabelle ist genau festgelegt, welche Methode bei welcher Nachricht auszuführen ist.

```
// Definition der Antworttabelle
DEFINE_RESPONSE_TABLE1(TMyWindow,TWindow)
    EV_WM_TIMER,
    EV_WM_KEYDOWN,
    EV_WM_MOUSEMOVE,
    EV_COMMAND(CM_PROJEKT_NEU,CmProjektNeu),
    EV_COMMAND(CM_PROJEKT_OEFFNEN,CmProjektOeffnen),
    EV_COMMAND(CM_PROJEKT_SPEICHERN,CmProjektSpeichern),
    EV_COMMAND(CM_PROJEKT_SPEICHERNUNTER,CmProjektSpeichernUnter),
    EV_COMMAND(CM_ROUTE_SPEICHERN,CmRouteSpeichern),
    EV_COMMAND_ENABLE(CM_ROUTE_SPEICHERN,CmRouteSpeichernEnable),
    EV_COMMAND(CM_OBJEKT_NEU,CmObjektNeu),
    EV_COMMAND(CM_OBJEKT_AENDERN,CmObjektAendern),
    EV_COMMAND(CM_OBJEKT_LOESCHEN,CmObjektLoeschen),
    EV_COMMAND(CM_ANWEISUNG_NEU,CmAnweisungNeu),
    EV_COMMAND(CM_ANWEISUNG_AENDERN,CmAnweisungAendern),
    EV_COMMAND(CM_ANWEISUNG_LOESCHEN,CmAnweisungLoeschen),
    EV_COMMAND(CM_STRECKEN,CmStrecken),
    EV_COMMAND(CM_OPTIMIERUNG1,CmOptimierung1),
    EV_COMMAND_ENABLE(CM_OBJEKT_LOESCHEN,CmEnablerObjektExists),
    EV_COMMAND_ENABLE(CM_PROJEKT_SPEICHERN,CmEnablerProjekt),
    EV_COMMAND_ENABLE(CM_PROJEKT_SPEICHERN,CmDateiname),
    EV_COMMAND_ENABLE(CM_PROJEKT_SPEICHERNUNTER,CmEnablerProjekt),
    EV_COMMAND_ENABLE(CM_OBJEKT_NEU,CmEnablerProjekt),
    EV_COMMAND_ENABLE(CM_OBJEKT_AENDERN,CmEnablerObjektExists),
    EV_COMMAND_ENABLE(CM_STRECKEN,CmEnablerStreckenExists),
    EV_COMMAND_ENABLE(CM_ANWEISUNG_NEU,CmEnablerObjektExists),
    EV_COMMAND_ENABLE(CM_ANWEISUNG_AENDERN,CmEnablerAnweisungExists),
    EV_COMMAND_ENABLE(CM_ANWEISUNG_LOESCHEN,CmEnablerAnweisungExists),
    EV_COMMAND(CM_WICHTUNGEN,CmWichtungen),
    EV_COMMAND_ENABLE(CM_OPTIMIERUNG1,CmEnablerAnweisungExists),
    EV_COMMAND(CM_ABOUT, CmAbout),
    EV_COMMAND(IDB_ZOOMPLUS,EvZoomPlusGadget),
    EV_COMMAND(IDB_ZOOMMINUS,EvZoomMinusGadget),
    EV_COMMAND(IDB_ROUTE_ANZ,EvRouteAnzGadget),
    EV_COMMAND(IDB_ZURUECK,EvZurueckGadget),
    EV_COMMAND(IDB_OK,EvOkGadget),
```

```
    EV_COMMAND_ENABLE(IDB_ZOOMPLUS,CmEnablerAnzeige3v4),
    EV_COMMAND_ENABLE(IDB_ZOOMMINUS,CmEnablerAnzeige3v4),
    EV_COMMAND_ENABLE(IDB_ROUTE_ANZ,CmEnablerAnzeige1),
    EV_COMMAND_ENABLE(IDB_OK,CmEnablerAnzeige4),
    EV_COMMAND_ENABLE(IDB_ZURUECK,CmEnablerAnzeige3),
    EV_WM_LBUTTONDOWN,
    EV_WM_PAINT,
    EV_WM_SIZE,
END_RESPONSE_TABLE;
```

Für jedes Ereignis, welches behandelt werden soll, existiert also in der Klasse `TMyWindow` eine entsprechende Methode. Jede dieser *Ereignismethoden* führt eine Reihe von Anweisungen aus, um auf das vorhergehende Ereignis zu reagieren.

So wird beispielsweise durch den Eintrag

```
EV_COMMAND(CM_OBJEKT_NEU,CmObjektNeu),
```

festgelegt, dass beim Eintreten des Ereignisses `CM_OBJEKT_NEU` die Methode `CmObjektNeu()` ausgeführt werden soll. Das Ereignis tritt übrigens immer dann ein, wenn in der Menüleiste der Eintrag zum Erstellen eines neuen Wachobjekts ausgewählt wurde.

Die Ereignismethode `CmObjektNeu()` ist in der Klasse `TMyWindow` definiert und sieht folgendermaßen aus:

```
void TMyWindow::CmObjektNeu()
{
    TDlgObjektNeu dlg(this);

    if (dlg.Execute() == IDOK)
    {
       Auswahl_Objekt=Anz_Objekte-1;
       Invalidate();
    }
    else {}
}
```

Demnach wird nach dem Drücken des Menüeintrages „Objekte\Neues Objekt erstellen...“ ein Objekt der Klasse `TDlgObjektNeu` erzeugt. Diese

ist wiederum von der Basisklasse `TDialog`, welche für die Darstellung von Dialogfenstern zuständig ist, abgeleitet und entsprechend den individuellen Gegebenheiten angepasst.
Dialogfenster sind eine sehr häufig unter Windows verwendete Fensterart. Sie werden immer dann eingesetzt, wenn zusammengehörige Daten vom Benutzer eingegeben werden müssen. Auf den folgenden Seiten soll die Vorgehensweise zur Definition der einzelnen Dialogfenster exemplarisch anhand der Klasse `TDlgObjektNeu` demonstriert werden.

Wie bereits erwähnt, wird die Klasse `TDlgObjektNeu` von der Basisklasse `TDialog` abgeleitet. Im Konstruktor der abgeleiteten Klasse wird für den Parameter `resId` der Bezeichner einer Dialogvorlage (`DLG_OBJEKTNEU`) aus der Ressourendatei als Standard-Parameter übergeben.

```
class TDlgObjektNeu : public TDialog
{
public:
    TDlgObjektNeu (TWindow* parent, TResId resId = DLG_OBJEKTNEU,
                   TModule* module=0);
```

Die Klasse `TDlgObjektNeu` überschreibt die beiden virtuellen Funktionen `SetupWindow()` und `CanClose()`. Die Funktion `SetupWindow()` nimmt die Initialisierung des Dialogfensters vor. Erst hier werden die Kontrollelemente des Dialogfensters, welche zwar schon durch den Konstruktor angelegt werden, erzeugt und dargestellt. Die Funktion `CanClose()` wird immer dann aufgerufen, wenn der „Ok“-Button (Bezeichnung `IDOK`) gedrückt wird (in diesem Dialogfenster hat dieser Button jedoch die abweichende Beschriftung „Erstellen“). Diese Funktion wird meistens zur Überprüfung von Dialogfenstern verwendet. Liefert die Funktion `false`, kann der Dialog nicht geschlossen werden.
In dieser, von `TDialog` abgeleiteten Klasse, wird diese Funktion überschrieben, um das Schließen des Dialogs bei ungültigen Eingaben zu verhindern.

```
protected:
    virtual void SetupWindow();
    virtual bool CanClose();
```

Durch die Dialogvorlage `DLG_OBJEKTNEU` (siehe Abbildung 4.19) ist zunächst das äußere Erscheinungsbild des Dialogfensters festgelegt.

Abbildung 4.19: Dialogvorlage `DLG_OBJEKTNEU`

Dieser Dialog enthält 9 Kontrollelemente: vier statische Textfelder, drei Editfelder und zwei Schaltflächen. Jedoch hätte das Dialogfenster so noch keinerlei Funktionalität. Aufgabe der Klasse `TDlgObjektNeu` ist es, auf die Eingaben, die der Benutzer in den Textfeldern vornimmt, zuzugreifen und durch Betätigen des „Erstellen“-Buttons ein neues Objekt zu erstellen.

Um auf die Kontrollelemente der Dialogvorlage zugreifen zu können, benötigt man sogenannte *Schnittstellenobjekte*. Da es sich um Editfelder handelt, müssen die Objekte von der Klasse `TEdit` sein.

```
// Schnittstellen für die Editfelder
 TEdit *m_EditText,*m_EditText2,*m_EditText3;
```

Die Zuordnung, welches Textfeld von welchem Objekt beschrieben wird, ist in der nachfolgenden Tabelle 4.5 festgelegt.

Objekt	Textfeld
`m_EditText`	*Bezeichnung*
`m_EditText2`	*Adresse*
`m_EditText3`	*Anzahl der Schlüssel*

Tabelle 4.5: Zuordnung der Schnittstellenobjekte

Da auch die Klasse `TDialog` von `TWindow` abgeleitet ist, wird an dieser Stelle wiederum eine Antworttabelle erwartet. In dieser abgeleiteten Klasse werden jedoch keine Einträge in der Antworttabelle vorgenommen.

```
    DECLARE_RESPONSE_TABLE(TDlgObjektNeu);
};
// Definition der Antworttabelle
DEFINE_RESPONSE_TABLE1(TDlgObjektNeu,TDialog)
//Leer
END_RESPONSE_TABLE;
```

Im Konstruktor der Klasse `TDlgObjektNeu` werden zunächst die Dialogvorlage und die sich darauf beziehenden Schnittstellenelemente angelegt.

```
TDlgObjektNeu::TDlgObjektNeu (TWindow* parent, TResId resId,
      TModule* module) : TDialog(parent,resId,module)
{
    // Erzeugen der Schnittstellenelemente
    m_EditText = new TEdit(this,IDC_EDIT_TEXT1,255); //Bezeichnung
    m_EditText2 = new TEdit(this,IDC_EDIT_TEXT2,255); //Adresse
    m_EditText3 = new TEdit(this,IDC_EDIT6,255); //Anzahl der Schlüssel
}
```

Zu diesem Zeitpunkt sind die einzelnen Kontrollelemente zwar angelegt, jedoch noch nicht dargestellt. Diese Aufgabe der Initialisierung übernimmt die Funktion `SetupWindow()`. Diese ruft zunächst die Funktion des Vorfahrens `TDialog::SetupWindow()` auf, um alle allgemeinen Initialisierungen, die zur Darstellung eines Dialogfensters nötig sind, durchzuführen.
Danach wird der Fenstertitel auf „Neues Objekt erstellen...“ sowie der Standardwert für die Anzahl der Schlüssel für das Objekt auf „1“ gesetzt.

```
void TDlgObjektNeu::SetupWindow()
{
    TDialog::SetupWindow();

    //Fensterkopf
    SetCaption('Neues Objekt erstellen...');
    m_EditText3->SetWindowText('1'); //Standardmäßig ein Schlüssel
}
```

Wie bereits erwähnt, wird die Funktion `CanClose()` immer dann aufgerufen, wenn der „Erstellen“-Button gedrückt wird. Nun kann man die Funktion dazu verwenden, die Eingaben des Benutzers zu überprüfen. Bei einer fehlerhaften Eingabe (z.B. Objektbezeichnung schon vergeben) wird dann ein Hinweistext auf dem Bildschirm ausgegeben und das Schließen des Dialoges durch Rückgabe des Funktionswertes `false` verhindert.

```
bool TDlgObjektNeu::CanClose()
{
    if (!TDialog::CanClose()) return false ;

    char buf[255];
    m_EditText->GetWindowText(buf,sizeof(buf));

    // Überprüfen, ob Bezeichnung eindeutig ist...
    bool test = true;

    if (strlen(buf) == 0) test=false;

    for(unsigned int i=0;i<Anz_Objekte;i++)
       if (strcmp(buf,Objekt[i].Bezeichnung) == 0) test=false;

    if (!test)
    {
       // Die Bezeichnung des Objekts ist bereits vergeben!
       MessageBeep(MB_ICONEXCLAMATION);
       MessageBox('Die Bezeichnung des Objekts ist bereits für ein
          anderes Objekt vergeben, bitte ändern Sie Ihren
          Eintrag dahingehend, dass die Bezeichnung eindeutig ist.',
          'Bezeichnungsfeld nicht eindeutig!',MB_ICONEXCLAMATION | MB_OK);
    }
    else
    {
       // Die Bezeichnung des Objekts ist noch nicht vergeben!
       // Deshalb wird nun ein neues Objekt mit den entsprechenden
       // Daten angelegt.
       Auswahl_Objekt=Anz_Objekte;
       Anzeige=4;
       strcpy(Objekt[Anz_Objekte].Bezeichnung,buf); //Bezeichnung
```

```
        m_EditText2->GetWindowText(buf,sizeof(buf)); //Adresse
        strcpy(Objekt[Anz_Objekte].Adresse,buf);
        m_EditText3->GetWindowText(buf,sizeof(buf)); //Anzahl der Schlüssel
        Objekt[Anz_Objekte].Anz_Key=(unsigned char)(atoi(buf));
        Anz_Objekte++;
    }

    return test;
}
```

Ist die Objektbezeichnung eindeutig (`test==true`), so wird das neue Objekt mit den entsprechenden Daten angelegt. Das Auslesen der eingegebenen Texte erfolgt über die Schnittstellenobjekte.

Analog dazu sind die übrigen Antwortfunktionen und Dialogklassen implementiert. Die vollständigen *Listings*[6] sind im Anhang und auf CD beigefügt. Einen Überblick über die verwendeten Klassen bietet das Klassendiagramm aus Abbildung 4.20.

[6] Quellcodedateien

Abbildung 4.20: Klassendiagramm

Kapitel 5

Applikation

Nach dem Starten des Programms, durch den Aufruf von GA-Tourenplaner.exe, erscheint die Benutzeroberfläche in Form eines Fensters im bekannten Windows-Look&Feel (siehe Abbildung 5.1). Zunächst sieht man lediglich die Menüleiste sowie die Angabe des aktuellen Projekts.
Das Programm wird hauptsächlich über die Menüeinträge gesteuert, welche in 5 Kategorien Projekt, Objekte, Wachanweisungen, Optimierung und Strecken unterteilt sind. Eine nähere Beschreibung der einzelnen Menüpunkte erfolgt in den nachfolgenden Abschnitten.

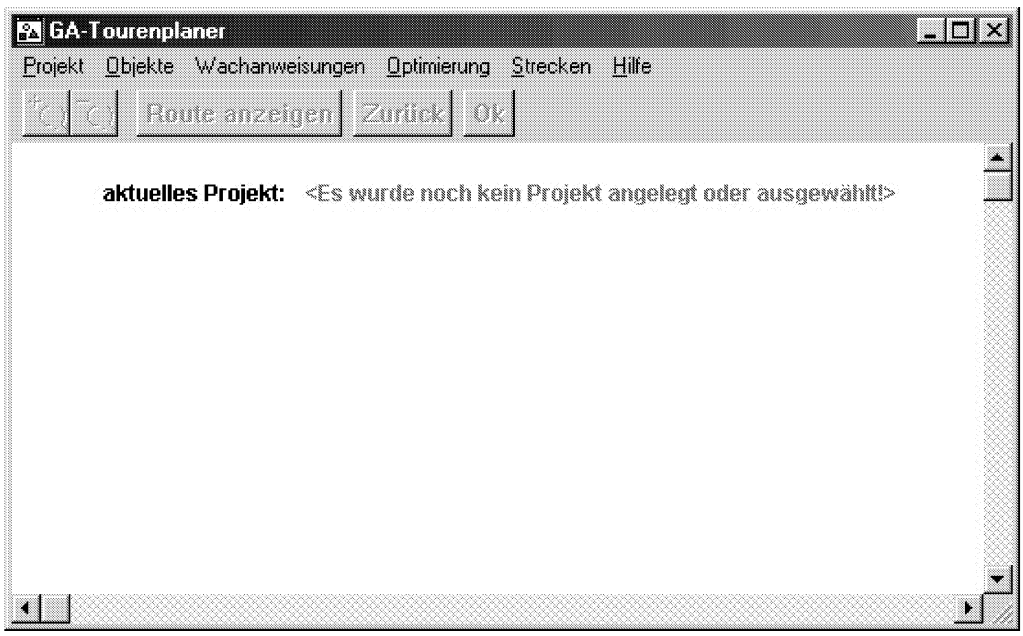


Abbildung 5.1: Programmoberfläche nach dem Start des Programms

5.1 Projekte

Ein *Projekt* stellt eine abgeschlossene Einheit dar, welches die Aufgabe hat, zusammengehörige Wachaufträge z.B. das Nachtrevier, Tagrevier etc. zu vereinen. Dadurch sollen inhaltlich, zeitlich und örtlich nicht zusammengehörige Wachanweisungen und Objekte voneinander getrennt und in verschiedenen Projekten bearbeitet werden können.

Um mit dem Programm arbeiten zu können, muss man zunächst ein Projekt anlegen. Dies erreicht man, indem man über den Menüpunkt Projekt→Neu... das Dialogfeld Neues Projekt anlegen... öffnet und dort den gewünschten Projektnamen eingibt. Daraufhin wird dieser Name in der Projektanzeige und im Fenstertitel angezeigt.
Zum Laden und Speichern von Projekten verwendet man die entsprechenden Menüeinträge Projekt→Öffnen... bzw. Projekt→Speichern.... Daraufhin erscheinen die standardmäßigen Windows-Dateiauswahlfenster, in denen man Dateinamen eingeben oder auswählen kann. Nach Bestätigung des gewünschten Dateinamens, wird die entsprechende Datei dann entweder geladen oder gespeichert.

Das Programm wird beendet, indem man den Menüpunkt Projekt→Beenden auswählt.

5.2 Objekte

Objekte beschreiben die einzelnen Wachobjekte, auf denen die Kontrollgänge durchgeführt werden sollen. Sie spezifizieren den Ort, an dem die Kontrolle stattfindet.
Dem Programm müssen sämtliche Wachobjekte bekannt gemacht werden. Dies ist notwendig für die spätere Optimierung der Tourenplanung.
Damit das Projekt auch ständig auf dem neuesten Stand gehalten und aktualisiert werden kann, für den Fall dass neue Kunden hinzukommen oder abspringen, stehen folgende Funktionen zur Verfügung:

1. neues Objekt anlegen
2. vorhandenes Objekt ändern
3. bestehendes Objekt löschen

Diese werden in den nachfolgenden Abschnitten kurz beschrieben.

5.2.1 Neues Objekt anlegen

Um ein neues Objekt anzulegen, wählt man im Menüpunkt Objekte den Eintrag Neues Objekt erstellen.... Es öffnet sich ein Dialog:

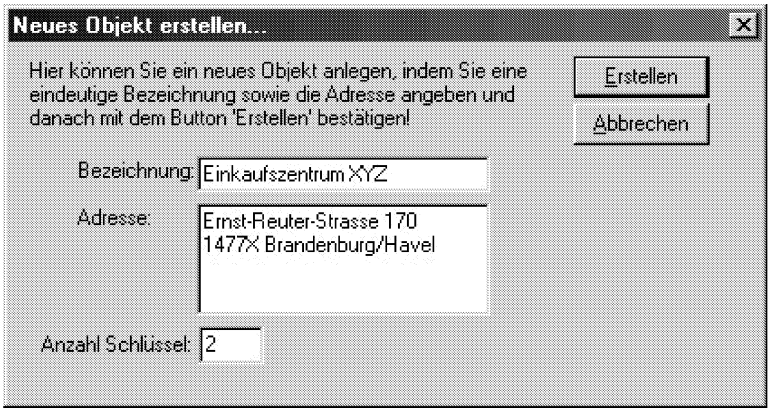


Abbildung 5.2: Dialog zum Anlegen eines neuen Wachobjekts

Hier trägt man neben der Bezeichnung und der Adresse des Wachobjekts auch die Anzahl der vorhandenen Schlüssel für dieses Objekt ein. Diese Angabe ist wichtig, da die einzelnen Kontrollen des Objekts maximal nur auf so viele Wachangestellte verteilt werden können, wie Schlüssel zu diesem Objekt vorhanden sind. Existiert nur ein Schlüssel für ein bestimmtes Objekt, so sind alle Kontrollen von ein und demselben Wachangestellten durchzuführen. Um später die einzelnen Objekte eindeutig zu unterscheiden, verwendet man keine Bezeichnungen doppelt.
Sind die entsprechenden Informationen eingegeben, so wird durch Drücken des Buttons „Erstellen“ das neue Wachobjekt angelegt und dem aktuellen Projekt hinzugefügt. Daraufhin erscheint eine digitale Straßenkarte von Brandenburg und Umgebung (siehe Abbildung 5.3). Hier gilt es nun, das eben erstellte Objekt punktgenau in diese Karte einzutragen, dazu stehen Navigations- und Zoomfunktion in der Symbolleiste zur Verfügung.

Genau an dieser Stelle wird der Straßenknoten ausgewählt, welcher am dichtesten am Objekt liegt. Dieser Knoten referenziert das Objekt innerhalb des Graphen und wird dazu verwendet, automatisch die Entfernungen vom neuen Objekt zu allen anderen Straßenknoten der bereits bestehenden Wachobjekte zu ermitteln[1].

[1] siehe Kapitel 4.3.3

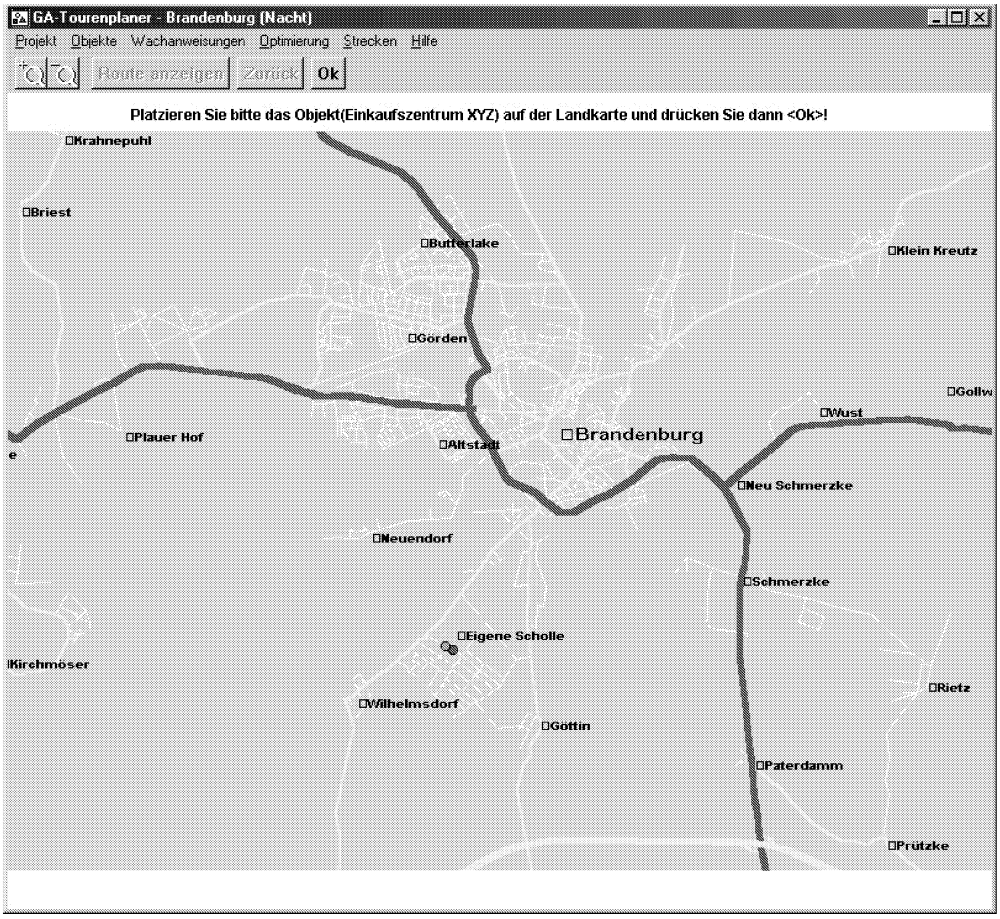


Abbildung 5.3: Die Straßenkarte dient zum Platzieren des Objekts

Nachdem die Berechnung der Entfernungen abgeschlossen ist, öffnet sich das Dialogfenster Streckenangaben (siehe Abbildung 5.4). Dieses Dialogfenster dient dazu, die ermittelten Distanzen und Fahrzeiten anzuzeigen, so dass der Anwender eine Vergleichsmöglichkeit hat und eventuell Änderungen vornehmen kann. Dabei werden die Fahrzeiten in Minuten und die Strecken in Kilometern angegeben.

5.2.2 Vorhandenes Objekt ändern

Möchte man ein bereits bestehendes Objekt ändern (z.B. den Namen, die Adresse oder die Anzahl der vorhandenen Schlüssel zum Objekt), so kann man dies tun, indem man den Menüpunkt Objekt→vorhandenes Objekt ändern... wählt.

Abbildung 5.4: Entfernungen zwischen dem neuen Objekt und allen anderen Objekten

Im nachfolgenden Dialog hat man die Möglichkeit die gewünschten Änderungen vorzunehmen. Dazu wählt man zunächst in der Auswahlliste das Objekt aus, welches verändert werden soll. Daraufhin wird in den dafür vorgesehenen Textfeldern die bisherige Bezeichnung, die Adresse und die Anzahl der Schlüssel des Objekts angezeigt. In diesen Feldern können eventuelle Änderungen vorgenommen und diese durch Drücken des „Aktualisieren“-Buttons übernommen werden. Nun öffnet sich erneut die Straßenkarte, in der auch die bisherige Position des Objekts angezeigt wird. Diese kann nun ebenfalls verändert werden, wodurch sich natürlich auch die Entfernungen zu allen anderen Objekten ändert. Aus diesem Grund müssen diese neu berechnet und die Ergebnisse erneut in der Maske Streckenangaben angezeigt werden.

5.2.3 Bestehendes Objekt löschen

Es kann natürlich auch der Fall eintreten, dass ein Objekt nicht mehr benötigt wird, weil dieser Kunde nicht mehr überwacht werden möchte. Dazu kann man das Wachobjekt des Kunden ganz einfach aus dem Projekt entfernen.

Hierzu wählt man einfach im Menü Objekt den Eintrag Objekt löschen... und es öffnet sich ein Dialogfenster. Darin wählt man in der Auswahlliste das Objekt aus, welches gelöscht werden soll. Als Kontrolle werden vor dem Löschvorgang nochmals alle Daten des Objekts (Bezeichnung, Adresse und Schlüsselanzahl) angezeigt. Erst durch Drücken des „Löschen“-Buttons erfolgt das endgültige Löschen des Objekts.

5.3 Wachanweisungen

Eine Wachanweisung definiert den Wachauftrag und die Anzahl der Kontrollen, die bezüglich eines bestimmten Objekts durchgeführt werden müssen. In der Wachanweisung wird angegeben an welchen Wochentagen diese gültig ist, wie viele Kontrollen durchzuführen sind, die Zeitfenster (Zeitspannen) in denen die Kontrollgänge stattfinden müssen sowie die eigentliche Dauer der Kontrollen (Kontrollzeit).

Die Definition der Wachanweisungen ist ein wichtiger Bestandteil des Programms. Mit der Definition der Wachanweisungen werden die Wachaufträge der Kunden in geeigneter Weise modelliert. Dieses Modell ist Grundlage für das darauf aufsetzende Optimierungsverfahren.

5.3.1 Eine Wachanweisung erstellen

Nachdem man im Menü Wachanweisungen den Unterpunkt Neue Wachanweisung... geöffnet hat, erscheint das Dialogfenster Neue Wachanweisung definieren.... (siehe Abbildung 5.5).
Hier wählt man zunächst das zu bewachende Objekt aus, auf welches sich die neue Wachanweisung schließlich beziehen soll. Danach werden die Wochentage angeklickt, an denen die Wachanweisung gültig ist. Daraufhin wird die Anzahl der Kontrollgänge eingegeben, die einzelnen Zeitspannen der Kontrollgänge definiert sowie die Kontrollzeiten (in Minuten) angegeben.
Zwischen zwei oder mehreren Kontrollen ist es notwendig noch eine Differenzzeit mit anzugeben, welche die Zeitspanne angibt, die mindestens zwischen zwei aufeinanderfolgenden Kontrollgängen vergehen muss. Damit wird u.a. die Situation ausgeschlossen, dass zwei Kontrollgänge an ein und demselben Objekt unmittelbar nacheinander stattfinden.

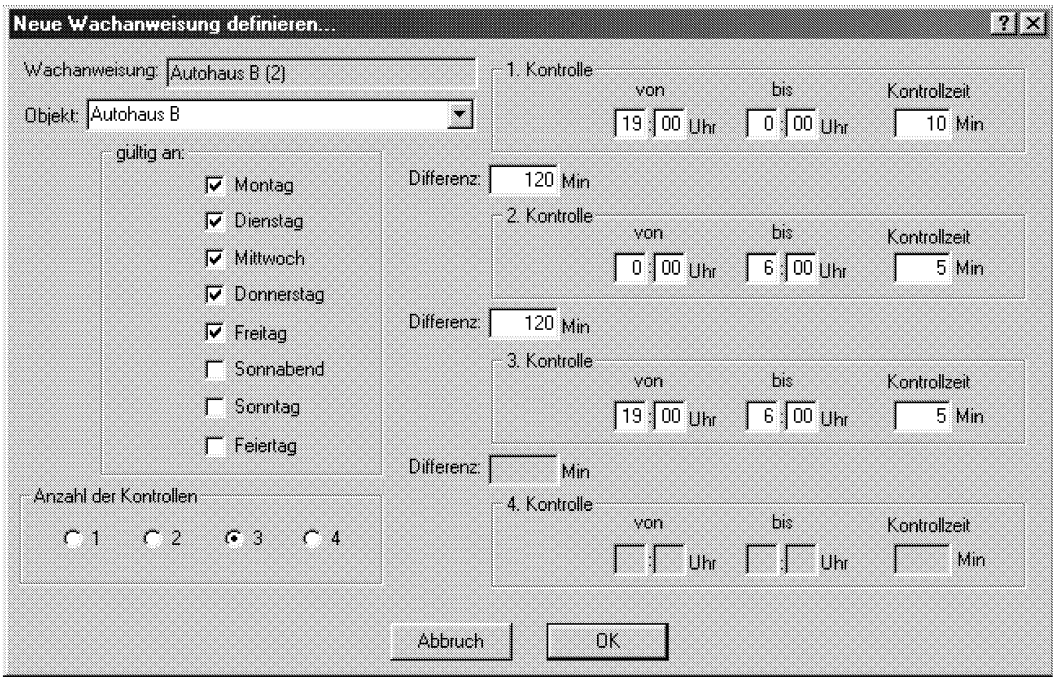


Abbildung 5.5: Formular zum Definieren der Wachanweisungen

Sind alle Angaben über die Wachanweisung gemacht worden, so wird die Wachanweisung durch Drücken des „OK“-Buttons angelegt.

5.3.2 Ändern einer bestehenden Wachanweisung

Sollen an einer bereits angelegten Wachanweisung Änderungen vorgenommen werden, so öffnet man über den Menüpunkt Wachanweisungen→Wachanweisung ändern... das Formular zum Ändern einer Wachanweisung.
Hier muss zunächst die Wachanweisung ausgewählt werden, an der Änderungen vorgenommen werden sollen. Die Auswahl der Wachanweisung erfolgt durch Angabe des zu bewachenden Objekts. Sind für ein Objekt mehrere Wachanweisungen definiert, so existiert eine fortlaufende Nummerierung ((1),(2),(3),...) um die einzelnen Wachanweisungen zu unterscheiden.

Wurde die gewünschte Wachanweisung ausgewählt, so werden die entsprechenden Daten dieser Wachanweisung angezeigt und können daraufhin den entsprechenden Änderungen angepasst werden.

Durch Betätigung des „Ändern“-Buttons wird die ausgewählte Wachanweisung mit den neuen Werten überschrieben.

5.3.3 Wachanweisung löschen

Wachanweisungen, die nicht mehr gültig sind, können gelöscht werden, indem man das Formular Wachanweisung löschen über den Menüpunkt Wachanweisungen→Wachanweisung löschen... aufruft. Darin wählt man die entsprechende Wachanweisung aus und bestätigt den Löschvorgang durch Drücken der „Löschen“-Taste.

5.4 Strecken

Normalerweise werden bereits beim Anlegen bzw. Ändern von Objekten die kürzesten Verbindungen zwischen allen Wachobjekten ermittelt und ggf. vom Anwender korrigiert. Man hat allerdings auch im Nachhinein die Möglichkeit diese Entfernungen einzusehen und zu verändern.
Dazu wählt man den Menüpunkt Strecken→Streckenangaben ändern... und kann sich die Einträge der Entfernungsmatrix für jedes Objekt anzeigen lassen.

5.5 Optimierung

Das Kernstück des Programms bildet das eigentliche Optimierungsverfahren. Die bisher beschriebenen Objekte, Strecken und Wachanweisungen dienen lediglich der Modellierung der Problemstellung. Dieses Modell wird benötigt, um den Prozess der Wachanweisung in geeigneter Art und Weise dem Rechner verständlich zu machen. Auf Basis eines erstellten Projekts setzt ein Optimierungsverfahren auf und bestimmt eine möglichst optimale Lösung.

Bevor man das eigentliche Optimierungsverfahren startet, hat man die Möglichkeit die Wichtungsfaktoren $w_1, w_2, \ldots, w_6$ für die Zielkriterien des Optimierungsverfahrens einzusehen und zu verändern. Die Gewichtungen beschreiben, wie stark sich jedes der Zielkriterien auf die Zielfunktion des Optimierungsverfahrens auswirkt.

Um zu dem Dialog zu gelangen, wo die Zielgewichtungen verändert werden können, wählt man im Menüpunkt Optimierung den Unterpunkt Einstellungen.... Daraufhin öffnet sich der *Einstellungsdialog* (siehe Abbildung 5.6).

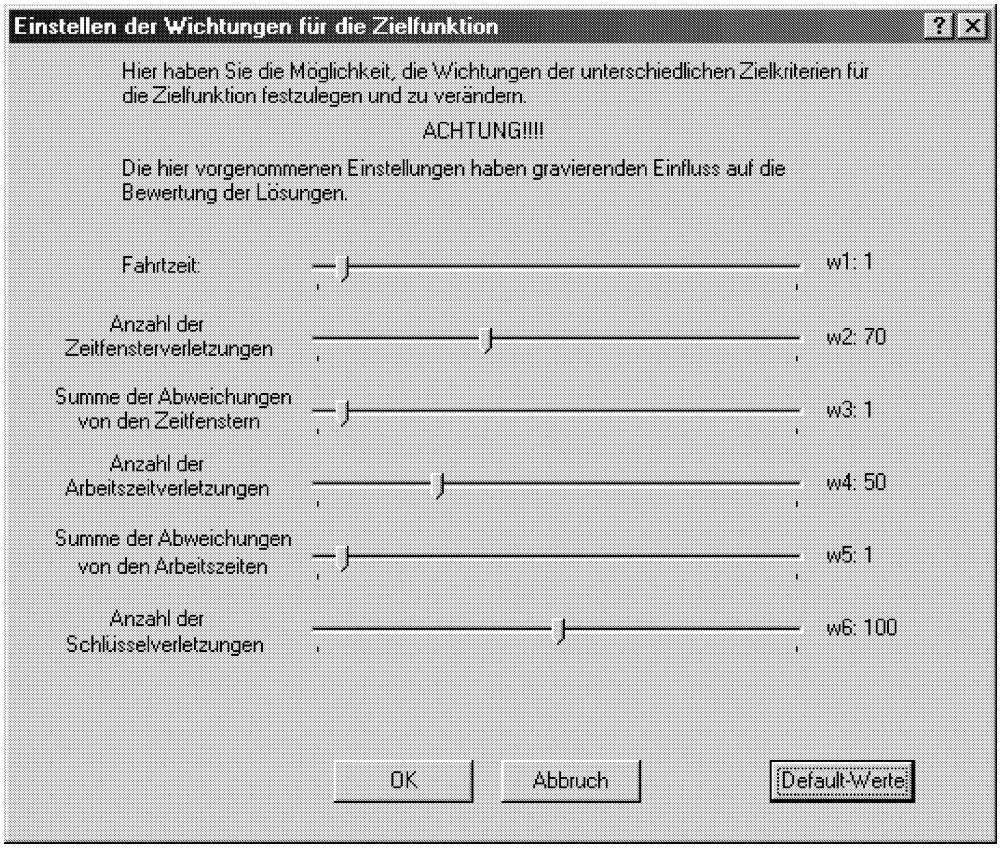


Abbildung 5.6: Dialog zum Verändern der Zielgewichtungen

In dem Dialogfenster findet man für jedes der angesprochenen Zielkriterien einen Schieberegler, mit dessen Hilfe sich nun nahezu jede beliebige Einstellung vornehmen lässt. Hier hat man nun die Chance eventuell einige Zielkriterien stärker mit in die Zielfunktion F mit einfließen zu lassen. Dabei haben die Wichtungsfaktoren $w_1, w_2, \ldots, w_6$ verschiedene Wertebereiche.

Es gilt folgende Festlegung:

$$w_1, w_3, w_5 \in \{1, 2, 3, \ldots, 20\}$$
$$w_2, w_4, w_6 \in \{n \in \mathbb{N}^* | n \leq 200\}$$

Diese wurden, ebenso wie die eingestellten *default*-Werte, intuitiv nach zahlreichen Testdurchläufen (siehe Kapitel 6) bestimmt. Die Zielgewichtungen sind zu Beginn auf diese *Standard*-Werte eingestellt.

Diese Standard-Einstellungen können jederzeit wieder hervorgerufen werden, indem man den Button „Default-Werte“ drückt. Erst durch Drücken des „OK“-Buttons werden die aktuell eingestellten Gewichtungen vom Programm übernommen.

Start des Optimierungsverfahrens

Zum Start der Optimierung der Tourenplanung wählt man im Menüfeld Optimierung den Unterpunkt Starten.... Es erscheint das Dialogfeld zur Auswahl einiger Parameter und Einstellungen (Abbildung 5.7). Hier gibt man neben dem Wochentag, für den die Tourenoptimierung gestartet werden soll, auch die Anzahl der Fahrer und die *GA-Parameter* ein.

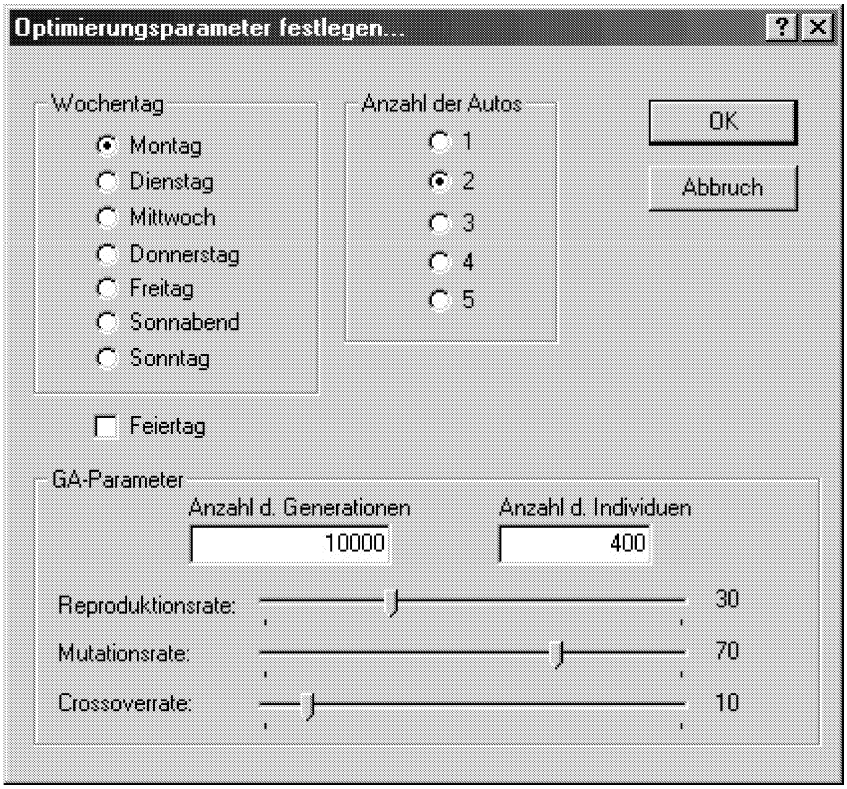


Abbildung 5.7: Einstellen der Parameter des GA

Bei den *GA-Parametern* handelt es sich um die Anzahl der Generationen, die Anzahl der Individuen sowie Reproduktions-, Mutations- und Crossoverraten. Auch diese Parameter sind standardmäßig auf Ihre *default*-Werte eingestellt.

Wurden alle Einstellungen vorgenommen, so bestätigt man diese mit „OK“ und es erscheint ein weiteres Formular.

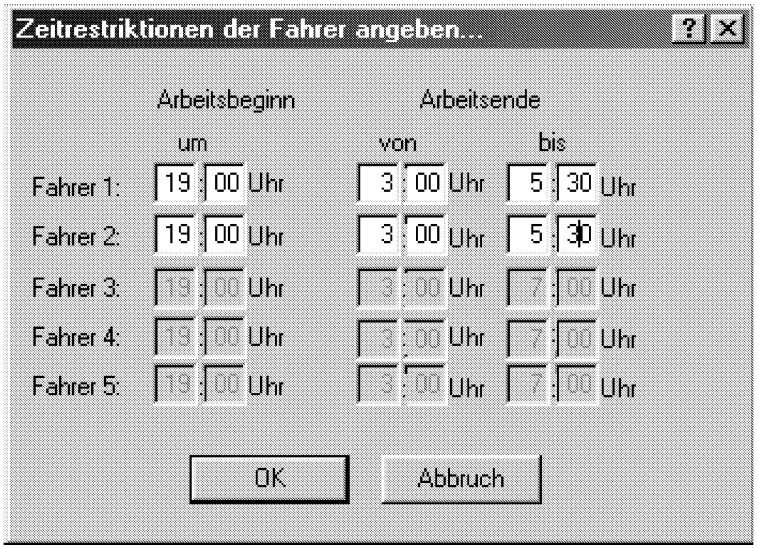


Abbildung 5.8: Festlegung der Zeitrestriktionen der Fahrer

In diesem Formular sind die Zeitrestriktionen der Einsatzfahrer[2] anzugeben:

- Arbeitsbeginn
- Arbeitsende

Sind auch diese Angaben gemacht worden, kann der genetische Algorithmus gestartet werden, indem man erneut „OK“ drückt.

5.5.1 Verlauf des Optimierungsverfahrens

Nachdem alle GA-Parameter und Einstellungen vorgenommen wurden, beginnt der eigentliche Lauf des Genetischen Algorithmus. Hierbei wird

[2]Das Programm ist derzeit auf höchstens 5 Fahrer ausgelegt. Dies ist dadurch begründet, dass die maximale Anzahl der Einsatzfahrer der Safe Wachschutz/Allservice GmbH derzeit bei drei je Revier liegt.

zunächst eine zufällige Anfangspopulation erzeugt und im Anschluss daran die einzelnen Schritte des GA durchlaufen, wie im Kapitel 4 ausführlich beschrieben. Als Abbruchkriterium gilt neben der maximalen Anzahl der Generationen $g_{\max}$ auch das Drücken der „ESC“-Taste.
Während des Verlaufs der Evolution wird im Hauptfenster angezeigt, wie weit das Optimierungsverfahren fortgeschritten ist. Dazu werden neben der aktuellen Generation und der bisher benötigten Programmlaufzeit (in Prozent) auch die Werte für die bisher beste gefundene Lösung ausgegeben.

Diese Werte sind:

- Zielfunktionswert
- Fahrzeit
- Anzahl der Zeitfensterverletzungen
- Anzahl der Arbeitszeitverletzungen
- Anzahl der Schlüsselverletzungen

Dadurch hat der Anwender die Möglichkeit den Verlauf des Optimierungsverfahrens am Bildschirm mitzuverfolgen und kann entscheiden, ob die bisher gefundene Lösung seinen Anforderungen genügt. Im letzteren Fall kann er das Optimierungsverfahren durch Drücken der „ESC“-Taste vorzeitig beenden. Ansonsten endet das Verfahren nach Ablauf aller Generationen.

5.5.2 Ausgabe der gefundenen Lösung

Nachdem das Optimierungsverfahren beendet wurde, wird die beste gefundene Lösung im Hauptfenster angezeigt (siehe Abbildung 5.9). Hier werden die einzelnen Routen der Fahrer mit den entsprechenden Objekten aufgelistet. Zu jedem Objekt ist neben der berechneten Ankunftszeit auch die Kontrollzeit angegeben. Am Ende jeder Route finden sich die kumulierten Kilometer und Fahrzeiten.
Die Gesamtfahrzeit sowie Kilometer sind weiter oben aufgelistet. Außerdem findet man dort die Information, ob alle Bedingungen erfüllt werden.
Entspricht dieser Routenvorschlag den Anforderungen, so kann er entweder abgespeichert (Menü Projekt→Route speichern...) oder direkt ausgedruckt und den Einsatzfahrern vorgelegt werden.

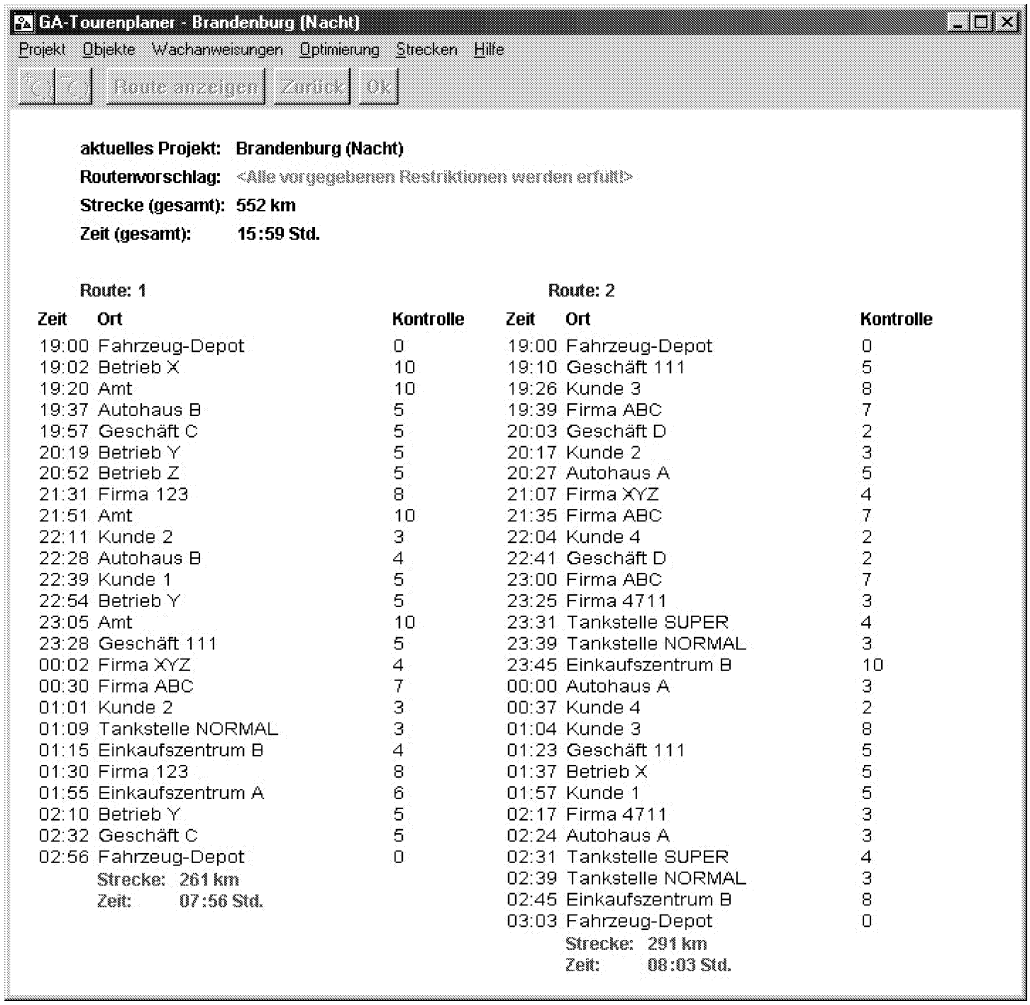


Abbildung 5.9: Ausgabe der besten gefundenen Lösung

Eine nützliche Funktionalität ist nun, sich die kürzeste Verbindung zwischen zwei aufeinanderfolgenden Objekten einer Route anzeigen zu lassen. Dazu wählt man zunächst das Objekt im Tourenplan aus (es wird daraufhin blau markiert) und klickt daraufhin den Button „Route anzeigen“ in der Symbolleiste. Nun öffnet sich erneut die Straßenkarte und es wird der Anfahrtsweg vom vorherigen Objekt zum ausgewählten Objekt blau dargestellt (siehe Abbildung 5.10).

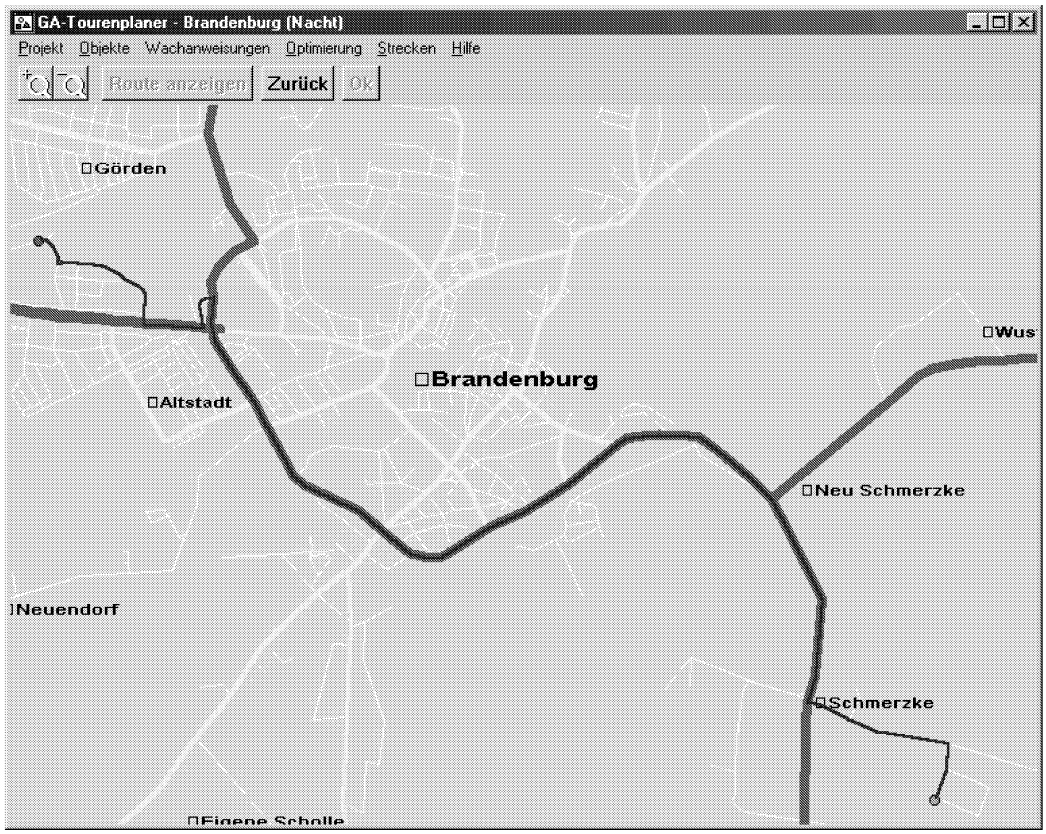


Abbildung 5.10: Anzeige der kürzesten Verbindung zwischen zwei Objekten

Kapitel 6

Tests und Auswertung

Dieses Kapitel dient dazu, den Einsatz des zuvor beschriebenen Programms zu untersuchen und erste Ergebnisse zu präsentieren.
Dazu wurde das Programm anhand des Nachtrevierplans von Brandenburg getestet. Die einzelnen Wachobjekte und Wachanweisungen können aufgrund von Geheimhaltungsvorschriften leider nicht mit aufgeführt werden. Die Definition der Zeitfenster und Angabe der Kontrollzeiten erfolgte in Absprache mit dem Einsatzleiter.
Das Nachtrevier besteht insgesamt aus 56 Wachobjekten, welche auf drei Wachangestellte verteilt werden. Hierbei ergab sich die Schwierigkeit, dass ein Einsatzfahrer als *ZBV* (*zur Besonderen Verfügung*) eingesetzt wird. Dieser nimmt eine Sonderstellung im Bereich des Wachschutzmodells ein. So hat er beispielsweise im Falle eines Alarms das entsprechende Gebäude zu besuchen, in dem die Alarmanlage aktiviert wurde.
Der ZBV kann aufgrund seiner unvorhersehbaren Einsätze nicht in die Berechnung einer optimalen Route aufgenommen werden. Dadurch reduzierte sich die Tourenplanung nun auf zwei Einsatzfahrer. Dennoch war die Vorstellung des Einsatzleiters, wenigstens einige Wachobjekte aus der Tour des Alarmfahrers mit auf die restlichen Fahrer zu verteilen, um diesen etwas zu entlasten. Daraufhin wurde das Optimierungsverfahren mit den entsprechenden Vorgaben gestartet.
Dies war ein mit Spannung erwarteter Moment, denn schließlich ist das Programm bisher nur an einigen Testszenarien hinsichtlich der Leistungsfähigkeit untersucht worden. Hier und dort musste zwar noch etwas an einigen Parametereinstellungen gefeilt werden, aber die ersten Ergebnisse waren sehr zufriedenstellend. Natürlich lieferte das Optimierungsverfahren mehrere verschiedene Lösungsvorschläge, was ja bekanntlich auch

üblich für einen Genetischen Algorithmus ist. Dabei ist auch nicht jeder Lösungsvorschlag zu gebrauchen, da mitunter nicht alle Restriktionen erfüllt werden. Es hat sich gezeigt, dass in etwa jeder fünfte Lauf einen Lösungsvorschlag hervorbringt, der die gegebenen Nebenbedingungen berücksichtigt.

Zur Berechnung wurden die Parametereinstellungen experimentell untersucht. Dabei wurden unterschiedliche Populationsgrößen von 10 bis 1 000 verwendet. Eine gute Einstellung scheinen Werte zwischen 100 und 400 zu sein. Größere Populationen bringen anscheinend keine merklichen Verbesserungen und verlängern nur unnötig die Rechenzeit. Die maximale Anzahl der Generationen $g_{\max}$ wurde anfangs auf Werte zwischen 20 000 und 100 000 gesetzt. Es hat sich allerdings gezeigt, dass nach der 50 000. Generation keine Verbesserungen mehr bezüglich der besten gefundenen Lösung auftreten. Die Laufzeit des Verfahrens mit Populatonsgrößen von 200 und 50 000 Generationen beträgt auf einem Pentium III mit 500 MHz ca. zwanzig Minuten. Laufzeiten dieser Größenordnung sind schon sensationell, damit kann der Einsatzleiter auch auf kurzfristige Kundenaufträge reagieren und sich einen entsprechenden Tourenplan berechnen lassen. Ein Beispiel für den Verlauf des Evolutionsverfahrens ist in den Abbildungen 6.1 und 6.2 dargestellt.

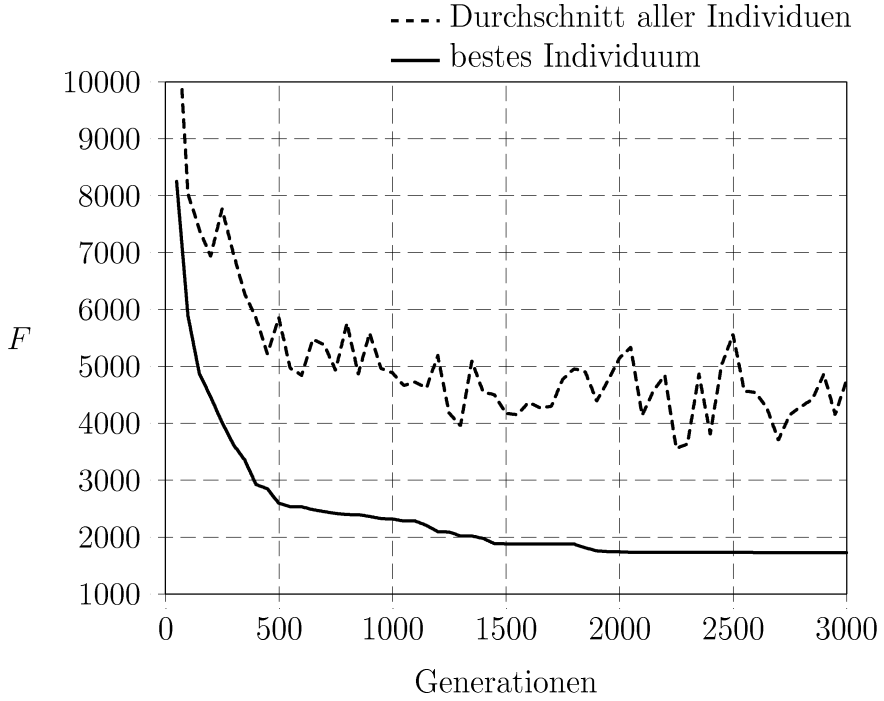


Abbildung 6.1: Verlauf der Evolution: Kostenfunktion

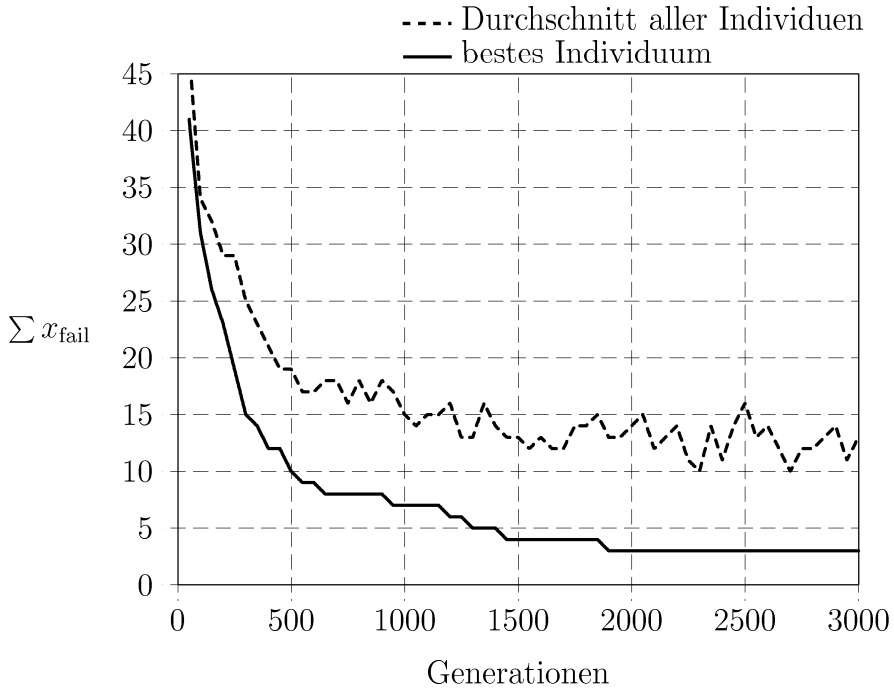


Abbildung 6.2: Verlauf der Evolution: Anzahl nicht erfüllter Restriktionen

Die Darstellung beschränkt sich auf den Verlauf während der ersten 3 000 Generationen. Darauf ist zu erkennen, wie aus anfangs noch sehr schlechten Tourenplänen allmählich immer bessere Lösungsvorschläge entstehen.

Durch die experimentellen Untersuchungen anhand des Nachtreviers, wurden auch intuitiv geeignete Einstellungen der anderen Parameterwerte, wie die Variationsraten (P_M, P_C, P_R) und Zielgewichtungen ($w_1, w_2, \ldots w_6$) , bestimmt[1]. In Tabelle 6.3 sind geeignete Wertebereiche für die einzelnen Parameter angegeben. Dabei gibt der *default*-Wert an, welche Einstellung standardmäßig im Programm voreingestellt ist.

[1]Eine umfassende Analyse sämtlicher relevanter Parameterwerte würde den Rahmen dieser Arbeit sprengen.

Parameter	geeignete Werte	default-Wert
P_M	$60 \leq P_M \leq 80$	70
P_C	$10 \leq P_C \leq 30$	30
P_R	$5 \leq P_R \leq 30$	10
w_1	$1 \leq w_1 \leq 3$	1
w_2	$10 \leq w_2 \leq 150$	70
w_3	$1 \leq w_3 \leq 2$	1
w_4	$30 \leq w_4 \leq 100$	50
w_5	$1 \leq w_5 \leq 3$	1
w_6	$50 \leq w_6 \leq 150$	100

Abbildung 6.3: Geeignete Parametereinstellungen

Doch nun zurück zu der eigentlichen Untersuchung. Die gefundenen Lösungsvorschläge zeigten, dass die Objekte, welche aus dem Revierplan des ZBV herausgenommen wurden, auch von den anderen beiden Fahrern übernommen werden können, ohne dass sich deren Arbeitszeit verlängern würde. Die Arbeitszeit der Einsatzfahrer im Nachtrevier Brandenburg beginnt um 19.00 Uhr und endet ca. um 6.00 Uhr.
Aus den verschiedenen gefundenen Tourenplänen wurde schließlich die beste Lösung herausgesucht und den Einsatzfahrern vorgelegt. Damit änderte sich erheblich der Arbeitsablauf der Fahrer. Während zuvor jeder Wachangestellte lediglich eine Liste der abzufahrenden Objekte erhielt und sich die Abfahrtsreihenfolge individuell einteilen konnte, wird ihm nun ein genauer Ablaufplan vorgelegt. Dieser schreibt genau vor, in welcher Reihenfolge die einzelnen Kontrollen durchzuführen sind. Dabei sind sogar die Ankunfts- und Kontrollzeiten eines jeden Objekts vorgegeben.
Es war interessant zu erfahren, wie die Wachangestellten darauf reagieren und ob sie die vorgesehenen Zeiten einhalten würden. Die vorgeschlagene Route wurde für drei Tage von den Einsatzfahrern abgefahren. Das Ergebnis dieser ersten Untersuchung war zunächst einmal, dass nicht alle Kontrollen bis 6.00 Uhr erledigt wurden. Weiterhin stellte sich heraus, dass einige Zeitfenster, obwohl mit dem Einsatzleiter abgesprochen, fehlerhaft definiert wurden. So wurde beispielsweise in einer Firma schon gearbeitet, als das Wachpersonal vor Ort war.
Daraus folgte eine Anpassung des Projekts, wobei zum einen die Zeitfenster berichtigt und zum anderen die Wachobjekte, die ursprünglich dem Alarmfahrer zugewiesen waren, herausgenommen wurden. Ausgehend von diesen Änderungen mussten nun erneut, durch mehrmaliges Starten des Optimie-

rungsverfahrens, mögliche Lösungsvorschläge gefunden werden. Die beste Lösung wurde dann wiederum von den Einsatzfahrern getestet. Das Ergebnis war nun schon deutlich besser. Diesmal wurden alle Kontrollgänge vom Wachpersonal durchgeführt, jedoch zeigte sich auch hier, dass die vorgesehenen Zeiten nicht immer eingehalten wurden, so dass die berechnete Zeitersparnis von ca. einer Stunde nicht erzielt wurde. Woran dies nun im Einzelnen liegt, muss näher untersucht werden.
Die Fahrzeiten zwischen den Objekten, welche schließlich automatisch ermittelt werden, sind eigentlich sehr großzügig bestimmt und vergleichbar mit den Angaben eines herkömmlichen Routenplaners.
Die tatsächliche Fahrzeit sollte sich demzufolge auch nicht merklich von der berechneten unterscheiden. Voraussetzung dafür ist allerdings, dass die kürzeste Route zwischen zwei Objekten auch wirklich gefahren wird. Die Einsatzfahrer erhalten schließlich nur die Abfahrtsreihenfolge. Wie der konkrete Weg zwischen den einzelnen Objekten aussieht, den sie fahren müssten, ist daraus nicht zu entnehmen. So ist davon auszugehen, dass eben nicht wirklich die kürzeste Route abgefahren wird. Hierzu müssten die kürzesten Routen, entsprechend der Abfahrtsreihenfolge, direkt vom Programm ausgedruckt und den Fahrern mitgegeben werden.
Man muss sich außerdem versuchen vorzustellen, welche Auswirkungen die neue Situation auf die Fahrer hat. Bisher waren sie es gewohnt, ihre Arbeitszeit, welche mit 11 bis 12 Stunden nicht gerade unerheblich ist, relativ frei einzuteilen. Wenn man sich vor Augen hält, dass ein Wachangestellter die ganze Zeit über an sein Auto gebunden ist, so ist es verständlich, dass sich die Objektbesuche auch nach persönlichen Bedürfnissen richten (z.B. wo kann man etwas essen oder welches Objekt besitzt einen Kaffeeautomaten).
Es bleibt festzuhalten, dass eine konsequente Umsetzung der berechneten Touren die täglichen Arbeitsbedingungen der Einsatzfahrer nicht gerade erhöht.

Kapitel 7

Zusammenfassung und Ausblick

Die Arbeit stellte eine große Herausforderung dar. Die Ziele waren sehr hoch gesteckt. Zunächst wurden verschiedene Tourenplanungsprobleme untersucht und eine Einordnung der Routenplanung, welche bei der Safe Wachschutz/Allservice GmbH vorliegt, vorgenommen. Diese lässt sich in Form eines *Vehicle Routing Problems* beschreiben.
Dabei gilt es, neben einer optimalen Aufteilung aller Kunden auf verschiedene Routen, auch die vorgeschriebenen Nebenbedingungen einzuhalten. Bei Problemen dieser Art handelt es sich um „schwierige“ Probleme[1], welche auch mit den heute schon sehr leistungsfähigen Rechenanlagen nicht in annehmbarer Zeit vollständig gelöst werden können. Das liegt einfach daran, dass die Anzahl der möglichen Anordnungen, die sich für die Aufteilung der Kunden auf die verschiedenen Fahrer ergibt, exponentiell wächst und schon für relativ kleine Probleme (mit 20 bis 30 Kunden) die dafür benötigten Rechenzeiten schnell bis auf einige Jahre ansteigen. Dies ist einfach nicht mehr akzeptabel.
Aus diesem Grund wurde versucht, alternative Lösungsmethoden für die gegebene Problemstellung zu finden. Dabei handelt es sich um sogenannte heuristische Lösungsverfahren, welche zwar nicht das globale Optimum im Lösungsraum finden, aber wenigstens in die Nähe einer optimalen Lösungsqualität gelangen. Die verschiedenen Verfahren wurden hinsichtlich einiger Kriterien untersucht und sich für die Implementierung eines *Genetischen Algorithmus* entschieden.
Es erfolgte daraufhin eine Beschreibung der zugrundeliegenden Mechanismen und eine Anpassung an die Gegebenheiten des Wachschutzprozesses. Dies erforderte zum einen die Verwendung einer speziellen Lösungsrepräsenta-

[1] im Sinne der Komplexitätstheorie

tion und zum anderen die Modifikation der genetischen Operatoren. Ein weiterer Schwerpunkt war die Aufstellung eines Modells zur Abbildung des gesamten Wachschutzprozesses. Damit können die einzelnen Kunden, Wachanweisungen, Kontrollzeiten etc. in angemessener Weise vom Benutzer beschrieben und somit auch vom Computer „verstanden“ werden. Das größte Problem tauchte dabei bei der Repräsentation des Straßennetzes auf. Grundlage für die Berechnung eines optimalen Tourenplans ist natürlich die Information darüber, welche Distanzen und Fahrzeiten zwischen den einzelnen Wachobjekten zurückgelegt werden müssen.

Hierzu wurden die drei mögliche Ansätze

- Verwendung der Luftlinienentfernungen
- Eingabe der gesamten Entfernungsmatrix
- Abbildung des Straßennetzes von Brandenburg und Umgebung

untersucht.

Der erste Ansatz wurde nicht näher verfolgt, da er keine Berücksichtigung des zugrundeliegenden Straßennetzes vornimmt und die so ermittelten Fahrzeiten einfach zu ungenau wären.
Der zweite Ansatz hingegen wurde zunächst in das Programm implementiert. Jedoch hat sich in Gesprächen mit den Mitarbeitern der Safe Wachschutz/Allservice GmbH gezeigt, dass es die manuelle Pflege der Entfernungsmatrix durch Hinzufügen neuer Wachobjekte einfach unzumutbar ist.
Dies führte zur Verfolgung des dritten Ansatzes. Hierbei wurde das Straßennetz von Brandenburg und Umgebung als gerichteter bewerteter Graph auf dem Rechner abgebildet. Der Aufwand des Benutzers zur Erstellung eines neuen Objekt ist somit auf dessen Positionierung innerhalb des Straßennetzes reduziert worden. Die Berechnung der kürzesten Entfernungen und Fahrzeiten zu den anderen Objekten erfolgt automatisch durch das Programm.

Der Genetische Algorithmus setzt schließlich auf das beschriebene Wachschutz- und Straßenmodell auf und liefert verschiedene Lösungsvorschläge.
Um diese Funktionalitäten für die Mitarbeiter bereitzustellen, wurde eine graphische Benutzeroberfläche für das Betriebssystem Windows geschaffen.

Hiermit kann der Benutzer auf recht einfache und komfortable Weise die unterschiedlichen Reviere in separate Projekte erfassen und verwalten. Sind alle wachspezifischen Daten eingegeben worden, so startet er das Optimierungsverfahren.
Für das Nachtrevier Brandenburg wurden Zeitersparnisse bis zu einer Stunde ermittelt. Schade ist nur, dass die Einsatzfahrer noch nicht über eine Anzeigemöglichkeit in ihren Fahrzeugen verfügen, welche ihnen die kürzesten Verbindung zwischen zwei Objekten anzeigt. Hierzu muss eine konsequentere Umsetzung seitens der Safe Wachschutz/Allservice GmbH erfolgen, wobei sich beispielsweise die Ausstattung der Einsatzfahrer mit GPS oder Handheld-Computern anbietet.
Auch bleibt abzuwarten, wie sich die Akzeptanz der Einsatzfahrer gegenüber dem neuen Verfahren entwickelt. Fest steht jedenfalls, dass das Potential des Programms noch nicht vollständig ausgeschöpft wird.

Mit der Erstellung dieses Programms ist eine sehr schöne und nützliche Anwendung geschaffen worden. Der Entwurf und die Implementierung stellten eine sehr anspruchsvolle Aufgabe dar. Es hat sich gezeigt, dass die Verwendung eines Genetischen Algorithmus erstaunlich gut zum Lösen größerer und schwieriger Tourenplanungsprobleme geeignet ist.
Einziger Knackpunkt des Programms ist und bleibt die Repräsentation des Straßennetzes. Obwohl dieser Ansatz im Vergleich zu den beiden vorhergehenden schon erheblich besser ist, kann man an dieser Stelle noch nicht zufrieden sein. Das liegt einfach daran, dass keine Objekte außerhalb von Brandenburg angelegt werden können (wie z.B. Berlin, Potsdam, etc.). Zwar können die entsprechenden Schnittstellendateien dahingehend erweitert werden, doch die manuelle Abbildung des Straßennetzes ist extrem arbeits- und zeitaufwendig. Wünschenswert wäre hier eine Anbindung an einen der unzähligen Tourenplaner, welche sich bereits auf dem Markt befinden und auch regelmäßig gepflegt werden (z.B. übers Internet).

Ansonsten sind lediglich nur noch ein paar Kleinigkeiten am Programm zu verbessern. Schön wäre es, nach dem Abbruch des GA einige Parameter zu verändern und diesen wieder fortsetzen zu können. Nützlich wäre auch die Möglichkeit einer manuellen Nachbearbeitung der gefundenen Route. Hinzu kommt das Laden von bereits gespeicherten Tourenplänen. Auch erfolgt bisher noch keine Überprüfung auf korrekte Eingabe der Zeitfenster. Wünschenswert ware auch, die Optimierung anhand einer bereits bestehenden Route vornehmen zu können.

Anhang A

Beschreibung der Schnittstellen

A.1 Die Datei Knoten.dat

Zeile	Beschreibung	Datentyp	Kurzbeschreibung
1	Anzahl der Datensätze (Knoten)	`int`	*Header*
2	X-Koordinate von Knoten 1	`long`	*Datensatz 1*
3	Y-Koordinate von Knoten 1	`long`	*Datensatz 1*
4	X-Koordinate von Knoten 2	`long`	*Datensatz 2*
5	Y-Koordinate von Knoten 2	`long`	*Datensatz 2*
6	X-Koordinate von Knoten 3	`long`	*Datensatz 3*
7	Y-Koordinate von Knoten 3	`long`	*Datensatz 3*
.	...	...	
.	...	...	
$2n$	X-Koordinate von Knoten n	`long`	*Datensatz n*
$2n+1$	Y-Koordinate von Knoten n	`long`	*Datensatz n*

Tabelle A.1: Aufbau der Datei `Knoten.dat`

A.2 Die Datei Kanten.dat

Zeile	Beschreibung	Datentyp	Kurzbeschreibung
1	Anzahl der Datensätze (Kanten)	`int`	*Header*
2	Id von Knoten 1	`long`	*Datensatz 1*
3	Id von Knoten 2	`long`	*Datensatz 1*
4	Durchschnittsgeschwindigkeit	`unsigned int`	*Datensatz 1*
5	Farbe (0 bis 4)	`unsigned int`	*Datensatz 1*
6	gerichtet/ungerichtet (0 oder 1)	`bool`	*Datensatz 1*
7	Strassenname	`char*`	*Datensatz 1*
8	Id von Knoten 1	`long`	*Datensatz 2*
9	Id von Knoten 2	`long`	*Datensatz 2*
10	Durchschnittsgeschwindigkeit	`unsigned int`	*Datensatz 2*
11	Farbe (0 bis 4)	`unsigned int`	*Datensatz 2*
12	gerichtet/ungerichtet (0 oder 1)	`bool`	*Datensatz 2*
13	Strassenname	`char*`	*Datensatz 2*
.	...	...	
.	...	...	
$6n-4$	Id von Knoten 1	`long`	*Datensatz n*
$6n-3$	Id von Knoten 2	`long`	*Datensatz n*
$6n-2$	Durchschnittsgeschwindigkeit	`unsigned int`	*Datensatz n*
$6n-1$	Farbe (0 bis 4)	`unsigned int`	*Datensatz n*
$6n$	gerichtet/ungerichtet (0 oder 1)	`bool`	*Datensatz n*
$6n+1$	Strassenname	`char*`	*Datensatz n*

Tabelle A.2: Aufbau der Datei `Kanten.dat`

A.3 Die Datei Bezeichnung.dat

Zeile	Beschreibung	Datentyp	Kurzbeschreibung
1	Anzahl der Datensätze	`int`	*Header*
2	X-Koordinate der Bezeichnung	`long`	*Datensatz 1*
3	Y-Koordinate der Bezeichnung	`long`	*Datensatz 1*
4	Schriftgröße	`unsigned int`	*Datensatz 1*
5	Bezeichnung/Ortsname	`unsigned int`	*Datensatz 1*
6	Winkel/Ausrichtung	`bool`	*Datensatz 1*
7	X-Koordinate der Bezeichnung	`long`	*Datensatz 2*
8	Y-Koordinate der Bezeichnung	`long`	*Datensatz 2*
9	Schriftgröße	`unsigned int`	*Datensatz 2*
10	Bezeichnung/Ortsname	`unsigned int`	*Datensatz 2*
11	Winkel/Ausrichtung	`bool`	*Datensatz 2*
.	...	...	
.	...	...	
$5n-3$	X-Koordinate der Bezeichnung	`long`	*Datensatz n*
$5n-2$	Y-Koordinate der Bezeichnung	`long`	*Datensatz n*
$5n-1$	Schriftgröße	`unsigned int`	*Datensatz n*
$5n$	Bezeichnung/Ortsname	`unsigned int`	*Datensatz n*
$5n+1$	Winkel/Ausrichtung	`bool`	*Datensatz n*

Tabelle A.3: Aufbau der Datei `Bezeichnung.dat`

A.4 Die Datei BRB.dat

Zeile	Beschreibung	Datentyp	Kurzbeschreibung
1	Anzahl der Datensätze (Punkte)	`int`	*Header*
2	X-Koordinate von Punkt 1	`long`	*Datensatz 1*
3	Y-Koordinate von Punkt 1	`long`	*Datensatz 1*
4	X-Koordinate von Punkt 2	`long`	*Datensatz 2*
5	Y-Koordinate von Punkt 2	`long`	*Datensatz 2*
6	X-Koordinate von Punkt 3	`long`	*Datensatz 3*
7	Y-Koordinate von Punkt 3	`long`	*Datensatz 3*
.	...	...	
.	...	...	
$2n$	X-Koordinate von Punkt n	`long`	*Datensatz n*
$2n+1$	Y-Koordinate von Punkt n	`long`	*Datensatz n*

Tabelle A.4: Aufbau der Datei `BRB.dat`

Anhang B

Listings

Hier werden lediglich einige Auszüge aus der Quellcodedatei `GA-Tourenplaner.cpp` aufgeführt. Dabei handelt es sich neben der Klassendeklaration von `TMyWindow` auch um einige ausgewählte Methoden dieser Klasse.
Der vollständige Quelltext und die restlichen Headerdateien zur Beschreibung der Dialogklassen finden sich auf der beiliegenden CD.

B.1 Klassendeklaration TMyWindow

```
/**
  *     T M y W i n d o w
  *
  *     Klassendeklaration
  */
class TMyWindow : public TWindow
{ public:
    TMyWindow();
    void SetupWindow();
    void EvTimer(uint timerId);
    virtual ~TMyWindow();
    LOGFONT MainFontRec;
    TScroller *ScrollerT;
    int CursorId;

protected:
    virtual void Paint(TDC &dc,bool, TRect &);
    void EvSize(uint , TSize& );

private:
    /*Bildschirmspeicher zum Switchen von Grafikinhalten (für Strassennetz)*/
    TMemoryDC* memoryDC;
    void ReadFile(FILE *in,char *string); //Methode zum Lesen von Dateiformat

    /* Antwortfunktionen (Menü)*/
    void CmProjektNeu();
```

```
void CmProjektOeffnen();
void CmProjektSpeichern();
void CmRouteSpeichern();
void CmProjektSpeichernUnter();
void CmProjektBeenden();
void CmWichtungen();
void teste_Constraints();
void sichere_Route();
void CmObjektNeu();
void CmObjektAendern();
void CmObjektLoeschen();
void CmAnweisungNeu();
void CmAnweisungAendern();
void CmAnweisungLoeschen();
void CmStrecken();
void CmOptimierung1();
void CmOptimierungX();
void CmEnablerAnweisungExists(TCommandEnabler &enabler);
void CmEnablerStreckenExists(TCommandEnabler &enabler);
void CmEnablerObjektExists(TCommandEnabler &enabler);
void CmDateiname(TCommandEnabler &enabler);
void CmEnablerProjekt(TCommandEnabler &enabler){ enabler.Enable(Projekt); };
void CmRouteSpeichernEnable(TCommandEnabler &enabler)
  {enabler.Enable(Chromosomlaenge>0);};
void CmAbout();
void EvLButtonDown(uint , TPoint& point);
void EvKeyDown(uint key, uint , uint );
void EvZoomPlusGadget();
void EvZoomMinusGadget();
void EvOkGadget();
void EvZurueckGadget();
void EvRouteAnzGadget();

void CmEnablerAnzeige3v4(TCommandEnabler &enabler)
{
  enabler.Enable((Anzeige==3)||(Anzeige==4));
}
void CmEnablerAnzeige3(TCommandEnabler &enabler)
{
  enabler.Enable(Anzeige==3);
}
void CmEnablerAnzeige4(TCommandEnabler &enabler)
{
  enabler.Enable(Anzeige==4);
}
void CmEnablerAnzeige1(TCommandEnabler &enabler)
{
  enabler.Enable((Anzeige==1)&&(Tourenplan.AuswahlX!=-1));
}
/*bei Mouse-Bewegungen*/
void EvMouseMove(uint , TPoint& point)
{
  TRect rc;
  GetClientRect(rc); /*ermittle Größe des Fensters*/
  int breiteX,breiteY;

  if ((Anzeige==3)||(Anzeige==4))
  {
    breiteX=min((int)rc.right,(int)800);
```

```
        breiteY=min((int)rc.bottom,(int)600);

        if ((point.x<(rc.right/2-breiteX/2+50))&&(CursorId!=IDC_LEFT))
           {SetCursor(GetModule(),IDC_LEFT);CursorId=IDC_LEFT;}
        if ((point.x>(rc.right/2+breiteX/2-50))&&(CursorId!=IDC_RIGHT))
           {SetCursor(GetModule(),IDC_RIGHT);CursorId=IDC_RIGHT;}
        if ((point.y<(rc.bottom/2-breiteY/2+30))&&(CursorId!=IDC_UP))
           {SetCursor(GetModule(),IDC_UP);CursorId=IDC_UP;}
        if ((point.y>(rc.bottom/2+breiteY/2-30))&&(CursorId!=IDC_DOWN))
           {SetCursor(GetModule(),IDC_DOWN);CursorId=IDC_DOWN;}
        if ((point.x>=(rc.right/2-breiteX/2+50))&&
         (point.x<=(rc.right/2+breiteX/2-150))&&
         (point.y>=(rc.bottom/2-breiteY/2+30))&&
         (point.y<=(rc.bottom/2+breiteY/2-30))&&
         (CursorId!=0))
            {SetCursor(GetModule(),IDC_ARROW);CursorId=0;}
      }
   }

   DECLARE_RESPONSE_TABLE(TMyWindow);
};
```

B.2 Die Methode EvRouteAnzGadget()

```
/**
  *    EvRouteAnzGadget
  *
  *    Methode zur Berechnung der kürzesten Verbindungen von
  *    einem Startknoten zu allen anderen Knoten des Graphen
  *    Imlementierung des Dijkstra-Algorithmus
  */
void TMyWindow::EvRouteAnzGadget()
{
   int B[MAX_STRASSEN_KNOTEN],B_index;

   Anzeige=3; /*Anzeige-Modus auf 3*/

   /*initialisiere alle Strassenknoten*/
   for(unsigned int i=1;i<=Anz_StrassenKnoten;i++)
   {
      StrassenKnoten[i].parent=-1;
   }

   /*Startknoten initialisieren*/
   start=Objekt[Route1].Knoten;
   StrassenKnoten[start].parent=start;
   StrassenKnoten[start].dist_km=0;
   StrassenKnoten[start].dist_min=0;

   B[0]=start; /*Agenda initialisieren*/
   B_index=0;

   while (B_index>=0)
   {
      unsigned int k,kindex;
      /*wähle Element k aus der Menge B mit minimaler Distanz*/
```

```
double dist_k=1000000; //sehr groß
for(int i=0;i<=B_index;i++)
{
   if (StrassenKnoten[B[i]].dist_min<dist_k)
   {
      dist_k=StrassenKnoten[B[i]].dist_min;
      k=B[i];kindex=i;
   }
}
/*entferne das Element k aus der Menge B*/
for(int i=kindex;i<B_index;i++)
{
   B[i]=B[i+1];
}
B_index--;

/*expandiere Knoten k -> ermittle alle Nachfolgerknoten*/

/*für alle Strassenkanten*/
for(unsigned int i=0;i<Anz_StrassenKanten;i++)
{
   int NF=-1;
   /*betrachte zunächst alle Kanten (gerichtet und ungerichtet),
   wo Knoten k an 1.Stelle steht*/
   if (StrassenKanten[i].K1==k) NF=StrassenKanten[i].K2; //NF = Nachfolger-Knoten

   /*bei ungerichteten Kanten kann Knoten k auch an 2.Stelle stehen*/
   if ((StrassenKanten[i].K2==k)&&(!StrassenKanten[i].R)) NF=StrassenKanten[i].K1;

   if (NF!=-1)
   {
      /*Berechne Zeit über k*/
      double dist_min=StrassenKnoten[k].dist_min+StrassenKanten[i].min;
      /*Berechne neue Distanz*/
     double dist_km=StrassenKnoten[k].dist_km+StrassenKanten[i].km;

      /*wenn Nachfolger neu (besitzt noch keinen parent)*/
      if (StrassenKnoten[NF].parent==-1)
      {
         StrassenKnoten[NF].parent=k;
         StrassenKnoten[NF].dist_min=dist_min;
         StrassenKnoten[NF].dist_km=dist_km;
         /*Nachfolger-Knoten in Menge B aufnehmen*/
         B_index++;
         B[B_index]=NF;
      }
      else
      {
         /*wenn neuer Weg über Knoten k kürzer*/
         if (dist_min<StrassenKnoten[NF].dist_min)
         {
            bool isElement=false;
            StrassenKnoten[NF].parent=k;
            StrassenKnoten[NF].dist_min=dist_min;
            StrassenKnoten[NF].dist_km=dist_km;
            /*wenn Knoten NF noch nicht in Menge B, dann einfügen*/
            for(int i=0;i<=B_index;i++) if (B[i]==NF) isElement=true;
            if (!isElement) {B_index++;B[B_index]=NF;}
         }//end if
```

```
            } //end else
          }//end if
        }//end for
    }//end while
    Invalidate();
}
```

B.3 Die Methode teste_constraints()

```
/**
  *     teste_constraints
  *
  *     Diese Methode überprüft, welche Knoten
  *     die vorgegebenen Zeitbedingungen nicht erfüllen.
  *     Knoten, die die Zeitrestriktionen verletzten,
  *     erhalten einen entsprechenenden Eintrag und werden
  *     im Tourenplan rot dargestellt.
  */
void TMyWindow::teste_Constraints()
{
    unsigned int i,travel;

    /*initialisiert alle Knotenfarben mit 0 (blau)*/
    for(i=0;i<=Knoten;i++) Knotenfarbe[i]=0;
    travel=0;
    Auto=1;
    Fahrt=false;
    RouteStart=Fahrer[Auto].Start;
    a_fail=0;
    afailAbs=0;

    for(i=0;i<Chromosomlaenge;i++)
    {
       if (ChromosomZ.Gen[i]==0) /*Blankzeichen -> andere Tour*/
       {
          if (travel>0) travel+=Strecken.min[KnotenObj[ChromosomZ.Gen[i-1]]][0];

          /*Sind alle Zeitschranken für Tour erfüllt?*/
          if (RouteStart>Fahrer[Auto].Start)
             {a_fail++;afailAbs+=RouteStart-Fahrer[Auto].Start;}
          if ((int)(travel+RouteStart)<Fahrer[Auto].EndeMin)
             {a_fail++;afailAbs+=Fahrer[Auto].EndeMin-(travel+RouteStart);}
          if ((int)(travel+RouteStart)>Fahrer[Auto].EndeMax)
             {a_fail++;afailAbs+=(travel+RouteStart)-Fahrer[Auto].EndeMax;}

          travel=0;
          Auto++;
          Fahrt=false;
          RouteStart=Fahrer[Auto].Start;
       }


       if (ChromosomZ.Gen[i]!=0)
       {
          /*vom Depot zu 1.Knoten*/
          if (!Fahrt) travel+=Strecken.min[0][KnotenObj[ChromosomZ.Gen[i]]];
```

```
        if (Fahrt)
        travel+=Strecken.min[KnotenObj[ChromosomZ.Gen[i-1]]][KnotenObj[ChromosomZ.Gen[i]]];
        Zeit[ChromosomZ.Gen[i]]=travel+RouteStart;

        travel+=Knotenzeit[ChromosomZ.Gen[i]];
        Fahrt=true;
      } //End if

   } //Next i

   /*vom letzten Knoten zum Depot*/
   if (ChromosomZ.Gen[i]!=0)
   {
     travel+=Strecken.min[KnotenObj[ChromosomZ.Gen[i-1]]][0];
   }

   /*Sind alle Zeitschranken für Tour erfüllt?*/
   if (RouteStart>Fahrer[Auto].Start) {a_fail++;afailAbs+=RouteStart-Fahrer[Auto].Start;}
   if ((int)(travel+RouteStart)<Fahrer[Auto].EndeMin)
     {a_fail++;afailAbs+=Fahrer[Auto].EndeMin-(travel+RouteStart);}
   if ((int)(travel+RouteStart)>Fahrer[Auto].EndeMax)
     {a_fail++;afailAbs+=(travel+RouteStart)-Fahrer[Auto].EndeMax;}

   z_fail=0; /*Anzahl der Constraintverletzungen ist zu Beginn null*/
   zfailAbs=0; /*Summe der Abweichungen der Constraints*/

   for(i=0;i<Anz_C;i++)
   {
     if (Constraint[i].Funktion==1)
     {
       if (Zeit[Constraint[i].Op1] < (Zeit[Constraint[i].Op2] + Constraint[i].k))
       {
         Knotenfarbe[Constraint[i].Op1]=1; /*Knoten wird rot dargestellt*/
         z_fail++; /*z_fail erhöhen*/
         zfailAbs+=abs(Zeit[Constraint[i].Op1]-(Zeit[Constraint[i].Op2]+Constraint[i].k));
       }

     }//End If

     if (Constraint[i].Funktion==2)
     {
       if (Zeit[Constraint[i].Op1] > (Zeit[Constraint[i].Op2] + Constraint[i].k))
       {
         Knotenfarbe[Constraint[i].Op1]=1; /*Knoten wird rot dargestellt*/
         z_fail++; /*z_fail erhöhen*/
         zfailAbs+=abs(Zeit[Constraint[i].Op1]-(Zeit[Constraint[i].Op2]+Constraint[i].k));
       }
     } //End If
   }//Next i
}
```

B.4 Die Methode CmOptimierungX()

```
/**
  *     CmOptimierungX()
  *
  *     Diese Methode wird bei jedem Timer-Interrupt aufgerufen
```

```
  *     und führt dazu, dass stets 10 Generationen durchlaufen werden.
  *     Am Ende des Durchlaufs wird überprüft, ob die max. Anzahl
  *     der Generationen bereits überschritten wurde.
  *     dadurch wird ggf. der Evolutionsprozess beendet und
  *     das beste Ergebnis ausgegeben.
  */
void TMyWindow::CmOptimierungX()
{
   unsigned int gen;
  double cost[MAX_IND+1],max,min;
  unsigned int t_travel[MAX_IND+1],travel;
  unsigned int Mutationsart,z1,z2,z3,i;

  KillTimer(1504); /*erstmal den Timer beenden*/
  for(gen=1;(gen<=10)&&((g+gen)<=Anz_Generationen);gen++) //10 Generationen durchlaufen
   {
     /*Fitness aller Individuen bestimmen*/
     sum=0;
     min=1000000000;

     for(unsigned int k=1;k<=Anz_Ind;k++)
     {
       t_travel[k]=0;
       travel=0;

       unsigned char Maske = 00000001;
       Auto=1;
       Fahrt=false;
       RouteStart=Fahrer[Auto].Start;
       a_fail=0;
       afailAbs=0;

       /*noch keiner Tour zugeordnet*/
       for(i=1;i<=Anz_Objekte;i++) ObjMaske[i]=00000000;

       for(unsigned int i=0;i<Chromosomlaenge;i++)
       {

         if (Chromosom[k].Gen[i]==0) /*Blankzeichen??? - dann andere Tour*/
         {
           if (travel>0) travel+=Strecken.min[KnotenObj[Chromosom[k].Gen[i-1]]][0];

           /*die Zeiten aller Routen zusammen addieren ergibt t_travel des Individuums*/
           t_travel[k]+=travel;

           /*Sind alle Zeitschranken für Tour erfüllt?*/
           if (RouteStart>Fahrer[Auto].Start)
             {a_fail++;afailAbs+=RouteStart-Fahrer[Auto].Start;}
           if ((int)(travel+RouteStart)<Fahrer[Auto].EndeMin)
             {a_fail++;afailAbs+=Fahrer[Auto].EndeMin-(travel+RouteStart);}
           if ((int)(travel+RouteStart)>Fahrer[Auto].EndeMax)
             {a_fail++;afailAbs+=(travel+RouteStart)-Fahrer[Auto].EndeMax;}

           travel=0;
           /*Bit in der Maske eins weiter nach links schieben (nächstes Auto)*/
           Auto++;Maske=Maske<<1;
           Fahrt=false;
           RouteStart=Fahrer[Auto].Start;
```

```
   }
   /*kein Blankzeichen*/
   if (Chromosom[k].Gen[i]!=0)
   {
      /*Objekt wird vom aktuellen Fahrzeug besucht
      deshalb entsprechendes Bit in der Maske setzen*/
      ObjMaske[KnotenObj[Chromosom[k].Gen[i]]]|=Maske;

      /*vom Depot zu 1.Knoten*/
      if (!Fahrt) travel+=Strecken.min[0][KnotenObj[Chromosom[k].Gen[i]]];
      if (Fahrt)
         travel+=Strecken.min[KnotenObj[Chromosom[k].Gen[i-1]]↪
         [KnotenObj[Chromosom[k].Gen[i]]];
      Zeit[Chromosom[k].Gen[i]]=travel+RouteStart;

      travel+=Knotenzeit[Chromosom[k].Gen[i]];
      Fahrt=true;
   } //End if

} //Next i

/*vom letzten Knoten zum Depot*/
if (Chromosom[k].Gen[i]!=0)
{
   travel+=Strecken.min[KnotenObj[Chromosom[k].Gen[i-1]]][0];
}

/*die Zeiten aller Routen zusammen addieren ergibt t_travel des Individuums*/
t_travel[k]+=travel;

/*Sind alle Zeitschranken für Tour erfüllt?*/
if (RouteStart>Fahrer[Auto].Start) {a_fail++;afailAbs+=RouteStart-Fahrer[Auto].Start;}
if ((int)(travel+RouteStart)<Fahrer[Auto].EndeMin)
   {a_fail++;afailAbs+=Fahrer[Auto].EndeMin-(travel+RouteStart);}
if ((int)(travel+RouteStart)>Fahrer[Auto].EndeMax)
   {a_fail++;afailAbs+=(travel+RouteStart)-Fahrer[Auto].EndeMax;}

z_fail=0; //Anzahl der Zeitverletzungen ist zu Beginn null
zfailAbs=0; //Summe der Abweichungen der Zeitfenster
key_fail=0; //Anzahl der Schlüsselverletzungen

/*sind alle Schlüsselbedingungen erfüllt?*/
for(unsigned int i=1;i<=Anz_Objekte;i++)
{
   unsigned char zaehler=0;
   Maske=00000001; //starte bei Tour 1
   for(unsigned char bit=0;bit<=7;bit++)
   {
      if ((ObjMaske[i]&Maske)!=0) zaehler++;
      Maske=Maske<<1;
   }
   /*Wenn nicht so viele Schlüssel vorhanden sind,
   wie das Objekt in Touren auftaucht*/
   if (Objekt[i].Anz_Key<zaehler) key_fail+=zaehler-Objekt[i].Anz_Key;
}
/*untersuche alle Zeitrestriktionen (Constraints)*/
for(i=0;i<Anz_C;i++)
{
   if (Constraint[i].Funktion==1) // >=
```

```
      {
         /*Kontrolle findet vor dem Zeitfenster statt*/
         if (Zeit[Constraint[i].Op1] < (Zeit[Constraint[i].Op2] + Constraint[i].k))
         {                  z_fail++; //erhöhe z_fail
            zfailAbs+=abs(Zeit[Constraint[i].Op1]-(Zeit[Constraint[i].Op2]+Constraint[i].k));
         }
      } //End If
      if (Constraint[i].Funktion==2) // <=
      {
         /*Kontrolle findet nach dem Zeitfenster statt*/
         if (Zeit[Constraint[i].Op1] > (Zeit[Constraint[i].Op2] + Constraint[i].k))
         {
            z_fail++; //erhöhe z_fail
            zfailAbs+=abs(Zeit[Constraint[i].Op1]-(Zeit[Constraint[i].Op2]+Constraint[i].k));
         }
      } //End If
   } //next i

   /*Berechnung der Kostenfunktion*/
   cost[k]=w1*t_travel[k]+(double)(w2*z_fail+w3*zfailAbs+w4*a_fail+w5*afailAbs+w6*key_fail);

   if (t_travel[k]>max) max=t_travel[k];
   if (t_travel[k]<min) min=t_travel[k];

   /*wenn aktuelles Individuum besser als bisher bestes
   if (best>cost[k])
   {
      best_t_travel=t_travel[k];
      best=cost[k];
      best_z_fail=z_fail;
      best_a_fail=a_fail;
      best_key_fail=key_fail;
      best_zfailAbs=zfailAbs;
      best_afailAbs=afailAbs;

      /*bestes Ind. sichern*/
      for(unsigned int m=0;m<Chromosomlaenge;m++)
      {               ChromosomZ.Gen[m]=Chromosom[k].Gen[m];
      } //next m

   } //End if best-Belegung ändern

} //Next k (nächstes Individuum der Population bewerten)

/* 0. Element=Bestes Element
for(unsigned int j=0;j<Chromosomlaenge;j++) Chromosom[0].Gen[j]=ChromosomZ.Gen[j];

/*********************************************************
// Selektionsprozess wird gestartet
// übernehme Individuen in Abhängigkeit ihrer Kostenfkt. in die neue Generation
for(unsigned int i=1;i<=Anz_Ind;i++)
{
   int el,el1,el2,el3;

   //Tournament selection
   el1=random(Anz_Ind)+1;
   el2=random(Anz_Ind)+1;
   el3=random(Anz_Ind)+1;
```

```
if (cost[el3]<cost[el2]) el2=el3;
if (cost[el2]<cost[el1]) el1=el2;
el=el1;

for(unsigned int j=0;j<Chromosomlaenge;j++) //for j=0 to k+m-2
{
   ChromosomN[i].Gen[j]=Chromosom[el].Gen[j];
}

//das Individuum el wurde aus der Population ausgewählt
//und wird nun entweder reproduziert,mutatiert oder
//rekombiniert mit einem anderen Individuum(Crossover)
//dazu dienen die jeweiligen Wahrscheinlichkeiten Pr,Pm,Pc

//Auswahl der Variationsart Rep,Mut,Cross entsprechend den jeweiligen Raten
int Variation;
int Zufall=random(P_Rep+P_Mut+P_Cross)+1;

if (Zufall<=P_Rep) Variation=0;
else if (Zufall<=P_Rep+P_Mut) Variation=1;
else Variation=2;

/*führe Mutation am Individuum durch*/
if (Variation==1)
{
   Mutationsart=random(5)+1;
   if (Mutationsart==1)
   {
      /*zufällige Positionen wählen*/
      z1=random(Chromosomlaenge); // 0¡=z1¡=k+m-2
      z2=random(Chromosomlaenge); // 0¡=z2¡=k+m-2

      //tausche
      ChromosomN[i].Gen[z1]=Chromosom[el].Gen[z2];
      ChromosomN[i].Gen[z2]=Chromosom[el].Gen[z1];

   }//Ende Mutationsart 1

   /*lokal begrenzeter Tausch zweier Elemente*/
   if (Mutationsart==2)
   {
      //zufällige Position wählen
      z1=random(Chromosomlaenge-1); // 0¡=z1¡=k+m-3

      //tausche
      ChromosomN[i].Gen[z1]=Chromosom[el].Gen[z1+1];
      ChromosomN[i].Gen[z1+1]=Chromosom[el].Gen[z1];

   } //Ende Mutationsart 2

   /*Inversion of sublist-Operator*/
   if (Mutationsart==3)
   {
      /*zufällige Positionen wählen*/
      z1=random(Chromosomlaenge); // 1.Stelle
      z2=random((Chromosomlaenge-1)-(z1))+1; // Länge der Subsequenz

      for(unsigned int j=0;j<z2;j++)
      {
```

```
      ChromosomN[i].Gen[z1+j]=Chromosom[el].Gen[z1+z2-j-1];
   }

} //Ende Mutationsart 3

/*Scramble sublist-Operator*/
if (Mutationsart==4)
{
   unsigned int n;
   /*zufällige Positionen wählen*/
   z1=random(Chromosomlaenge); //1.Stelle
   z2=random((Chromosomlaenge-1)-(z1))+1; //Länge der Subsequenz

   for(unsigned int j=1;j<=z2;j++) Markiert[j]=0;
   summe=0;
   n=z1;
   while (summe<z2)
   {
      z3=random(z2)+1; //Position
      if (Markiert[z3]==0)
      {
         summe++;
         Markiert[z3]=1;
         ChromosomN[i].Gen[n]=Chromosom[el].Gen[z1+z3-1];
         n++;
      }
   }

} //Ende Mutationsart 4

/*Verschiebung eines Sequenz-Teilstückes*/
if (Mutationsart==5)
{
   unsigned int n;
   /*zufällige Positionen wählen*/
   z1=random(Chromosomlaenge); //1.Stelle
   z2=random((Chromosomlaenge-1)-(z1))+1; //Länge der Subsequenz
   z3=random(Chromosomlaenge-z2+1); //Einfügeposition

   /*1. Löschen des Sequenz-Teilstückes*/
   n=0;
   for(unsigned int j=0;j<Chromosomlaenge;j++)
   {
      ChromosomN[i].Gen[n]=Chromosom[el].Gen[j];
      if ((j<z1)||(j>=z1+z2)) n++;
   } // Next j

   /*2. Einfügen des Sequenz-Teilstückes*/
   n=0;
   for(unsigned int j=Chromosomlaenge-1;j>z3;j--)
   {
      ChromosomN[i].Gen[j]=ChromosomN[i].Gen[j-z2];
   }

   for(unsigned int j=0;j<z2;j++)
   {
      ChromosomN[i].Gen[z3+j]=Chromosom[el].Gen[z1+j];
   }
} //Ende Mutationsart 5
```

```
   } //Ende Mutation

   /*führe durch PMX-Crossover*/
   if ((Variation==2)&&(i<Anz_Ind))
   {
      int el1,el2,el3;

      /*Tournament selection - wähle 2. Elternteil aus*/
      el1=random(Anz_Ind)+1;
      el2=random(Anz_Ind)+1;
      el3=random(Anz_Ind)+1;

      if (cost[el3]<cost[el2]) el2=el3;
      if (cost[el2]<cost[el1]) el1=el2;
      el2=el1;

      for(unsigned int j=0;j<Chromosomlaenge;j++)
      {
         ChromosomN[i+1].Gen[j]=Chromosom[el2].Gen[j];
      }

      /*zufällige Positionen wählen*/
      z1=random(Chromosomlaenge); //1.Stelle
      z2=random((Chromosomlaenge-1)-(z1))+1; //Länge der Subsequenz

      for(unsigned int n=z1;n<z1+z2;n++)
      {
         for(unsigned int j=0;j<Chromosomlaenge;j++)
         {
            if (ChromosomN[i+1].Gen[j]==ChromosomN[i].Gen[n])
            {
               ChromosomN[i+1].Gen[j]=ChromosomN[i+1].Gen[n];
               break;
            }
         }
         ChromosomN[i+1].Gen[n]=ChromosomN[i].Gen[n];
      }

      for(unsigned int n=z1;n<z1+z2;n++)
      {
         for(unsigned int j=0;j<Chromosomlaenge;j++)
         {
            if (ChromosomN[i].Gen[j]==Chromosom[el].Gen[n])
            {
               ChromosomN[i].Gen[j]=ChromosomN[i].Gen[n];
               break;
            }
         }
         ChromosomN[i].Gen[n]=Chromosom[el].Gen[n];
      }
      i++;
   }//Ende Crossover
} //Next i
/*Ende Selektion+Variation*/

// ******************************************************
//neue Generation belegen
for(unsigned int k=1;k<=Anz_Ind;k++)
{
```

```
        for(unsigned int j=0;j<Chromosomlaenge;j++) Chromosom[k].Gen[j]=ChromosomN[k].Gen[j];
      }// Next k

   } //Next g

   g=g+10;
   if (g<Anz_Generationen) //wenn g kleiner als g_max, dann
   {
      SetTimer(1504, 200, 0); //setze erneut Timer auf 200ms
      Invalidate(); //neu zeichnen
   }

   if (g>=Anz_Generationen) //wenn g größer gleich g_max, dann
   {
      unsigned char buf[255];
      teste_Constraints();
      sichere_Route();
      Anzeige=1; //Anzeige-Modus 1 (Ausgabe des Tourenvorschlags)
      Tourenplan.AuswahlX=-1;Tourenplan.AuswahlY=-1;
      Invalidate(); //Fenster neuzeichnen
      sprintf(buf,'Das Optimierungsverfahren ist beendet.\n\nBeste gefundene Lösung:\n\n
       Anzahl der Zeitfensterverletzungen:\t%d\nAnzahl der Arbeitszeitverletzungen: \t%d\n
       Anzahl der Schlüsselverletzungen:\t%d\n\nDer Lösungsvorschlag wird nun
       auf dem Bildschirm ausgegeben!',best_z_fail,best_a_fail,best_key_fail);
      MessageBox(buf,'Optimierung beendet!',MB_OK);
   }
} //Ende Cm_OptimierungX()
```

Anhang C

CD-Inhalt

Verzeichnis	Inhalt
Dokumentation	Dokumentation zum Programm
GA-Tourenplaner	lauffähige Programmversion
Literatur	Literatur zu den Literaturangaben, die aus dem Internet stammen
TeX	LaTeX-, PostScript- und Pdf-Dateien der vorliegenden Diplomarbeit
Quelltexte	Borland C++ Projektdatei Quellcodedateien Ressourcendatei Headerdateien

Literaturverzeichnis

[Ba01] Helmut **Balzert**. *Lehrbuch der Software-Technik*. Spektrum, Akad. Verlag, 2001.

[BK89] Horst **Bayrhuber** und Ulrich **Kull**. *Linder Biologie: Lehrbuch für die Oberstufe*. Schroedel Schulbuchverlag GmbH, 1989.

[Be01] M. **Bernreuther**. *Kürzster-Weg-Suche mit dem Dijkstra - Algorithmus*. Script zur Vorlesung *Einführung in die Informatik* an der Universität Stuttgart, 2001. www.uni-stuttgart.de/iv-kib/generic/download/lehre/einfuehrung/ws00_01/vorlesung/mb_dijkstra.pdf (Stand 01/2003).

[Bo01] Ingo **Boersch**. *Simulierte Evolution*. Script zur Vorlesung Wissensverarbeitung an der Fachhochschule Brandenburg, 2001. ots.fh-brandenburg.de/mod/ams/data/wva/unterlagen/evo.pdf (Stand 01/2003).

[Cr99] Reiner **Creutzburg**. *Algorithmen und Datenstrukturen*. Script zur gleichnamigen Vorlesung an der Fachhochschule Brandenburg, 1999.

[DSW99] Gunter **Dueck**, Tobias **Scheuer** und Hans-Martin **Wallmeier**. *Toleranzschwelle und Sintflut: neue Ideen zur Optimierung*. Spektrum der Wissenschaft, (Heft 2/1999):22-31, 1999.

[GP99] Martin **Grötschel** und Manfred **Padberg**. *Die optimierte Odyssee*. Spektrum der Wissenschaft, (Heft 2/1999):32-41, 1999.

[HS99] Jochen **Heinsohn** und Rolf **Socher-Ambrosius**. *Wissenverarbeitung : Eine Einführung*. Spektrum, Akad. Verlag, 1999.

[HH72] Ulrich **Hoffmann** und Hanns **Hofmann**. *Einführung in die Optimierung*. Weinheim: Akademische Verlagsgesellschaft, 1972.

[Jo02] Ivan **Jordanov**. *Intro to Genetic Algorithms. GA Terminology and Operators* Vorlesungscript an der De Montfort University - Milton Keynes, 2002. www.mk.dmu.ac.uk/∼ijordanov/NN&GA/Lectures/Lecture7.pdf (Stand 01/2003)

[Ko93] John R. **Koza**. *Genetic Programming.* Cambridge: MIT Press, 1993.

[Kr98] Sven Oliver **Krumke**. *Graphentheoretische Konzepte und Algorithmen.* Vorlesungsscript an der Ludwigs-Maximilians-Universität Würzburg, 1998. www.zib.de/krumke/Postscript/graphen-skript.pdf (Stand 01/2003).

[LC01] Uwe **Laemmel** und Jürgen **Cleve**. *Künstliche Intelligenz.* Weinheim: Fachbuchverlag Leipzig, 2001.

[Mu97] Friedhelm. **Mündemann**. *Formale Sprachen und Automatentheorie.* Script zur gleichnamigen Vorlesung an der Fachhochschule Brandenburg, 1997.

[Ni94] Volker **Nissen**. *Evolutionäre Algorithmen - Darstellung, Beispiele, betriebswirtschaftliche Anwendungsmöglichkeiten.* Wiesbaden: Deutscher Universitätsverlag GmbH, 1994

[Ni97] Volker **Nissen**. *Einführung in Evolutionäre Algorithmen - Optimierung nach dem Vorbild der Evolution.* Wiesbaden: Vieweg, 1997.

[SHF94] Eberhard **Schöneburg**, Frank **Heinzmann** und Sven **Feddersen**. *Genetische Algorithmen und Evolutionsstrategien.* Addison-Wesley, 1994

[Sc95] Uwe **Schöning**. *Theoretische Informatik - kurzgefaßt.* Spektrum: Akad.Verl., 1995

[SH96] Volker **Sperschneider** und Barbara **Hammer**. *Theoretische Informatik: Eine problemorientierte Einführung.* Berlin: Springer-Verlag, 1996.

[St99] Matthias **Stümpfle**. *Planung und Optimierung von prioritätsbasierten Steuergerätenetzen für Fahrzeuge.* Dissertation an der Fakultät Elektrotechnik und Informationstechnik der Universität Stuttgart, 1999.
www.ind.uni-stuttgart.de/Content/Publications/Archive/St_Diss_30357.pdf (Stand 01/2003)

[We72] Hans Hermann **Weber**. *Einführung in Operation Research.* Frankfurt am Main: Akademische Verlagsgesellschaft, 1972.

[Zh01] Hantao **Zhang**. *Single Source Shortest Paths.* Vorlesungsskript an der University of Iowa, 2001. www.cs.uiowa.edu/∼hzhang/c44/lec26.PDF (Stand 01/2003)

Erklärung

Ich erkläre hiermit, dass die vorliegende Arbeit von mir selbst und ohne fremde Hilfe verfasst wurde. Alle benutzten Quellen sind im Literaturverzeichnis angegeben. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Brandenburg, 4. Februar 2003

Tino Schonert