Universität Bayreuth
Fakultät für Mathematik und Physik
Fachgruppe Mathematik

Heuristiken und genetische Algorithmen für modifizierte Steinerbaumprobleme

Diplomarbeit

Clemens Frey
Stuckbergstraße 3
95448 Bayreuth

1. Juli 1997

Betreuung: Prof. Dr. Günter Leugering
Fakultät für Mathematik und Physik
Fachgruppe Mathematik
Universität Bayreuth

Für Ulli

This document was prepared with LaTeX.

Inhaltsverzeichnis

Prolog

Im Mittelpunkt dieser Arbeit stehen Modifikationen des klassischen Steiner-Problems auf ungerichteten Graphen mit Kantenbewertung.

Nach Klärung graphentheoretischer Grundlagen werden im ersten Kapitel verschiedene Formulierungen sowohl des klassischen Steiner-Problems als auch des durch zusätzlich vorhandene Knotenbewertungen modifizierten Steiner-Problems definiert. Im letzten Abschnitt des ersten Kapitels wird das programmtechnische Fundament für alle später folgenden Algorithmen, nämlich die Datenstruktur für die Speicherung von Graphen, vorgestellt.

Das zweite Kapitel überträgt zwei Algorithmen, die zur Gruppe der sogenannten "Path Distance"-Heuristiken für das klassische Steiner-Problem gehören, auf die in der Einführung vorgestellten modifizierten Steiner-Probleme.

Im dritten Kapitel erläutern wir die bei der Implementierung der Heuristiken in der Programmiersprache C++ verwendeten Klassen-Deklarationen. Dies ermöglicht es dem Leser, die hier eingesetzten Algorithmen aus eigenen Programmen heraus aufzurufen und damit wiederzuverwenden.

Genetische Algorithmen als Mittel zur Verbesserung von Näherungslösungen wurden schon mehrmals auch für graphentheoretische Probleme eingesetzt. Im vierten Kapitel wird ein derartiger Algorithmus präsentiert, der sich durch die Verwendung eines dynamischen, sich an den von Heuristiken gelieferten Ergebnissen orientierenden Crossover-Operators vom bisher verwendeten klassichen Ansatz unterscheidet. Die Gültigkeit einer dem fundamentalen Satz von Holland ähnlichen Aussage über Schemata wird gezeigt. Vor der Darstellung der bei der Realisierung des genetischen Algorithmus verwendeten Klassen wird eine Methode dargestellt, mit deren Hilfe sich die Lage des Optimums eines Steiner-Problems nach unten abschätzen läßt.

Die erste Hälfte des letzten Kapitels enthält Ergebnisse und Laufzeiten, die die verschiedenen Heuristiken und der genetische Algorithmus bei Anwendung auf Standard-Testprobleme liefern. In der zweiten Hälfte sind Resultate wiedergegeben, die die Algorithmen liefern, wenn sie auf Graphen angewendet werden, die zwei- bzw. dreidimensionale Gitter modellieren.

Datenträger, die die Quellcodes der hier verwendeten Programme enthalten, und die dazugehörige technische Dokumentation sind vom Autor zu beziehen. Dasselbe gilt für Programmodule, die die Algorithmen für MATLAB*-Nutzer verfügbar machen, und für Packages, die die Handhabung von Graphen in* MATHEMATICA *erleichtern.*

Kapitel 1

Einführung

1.1 Graphentheoretische Notationen I - ungerichtete Graphen

In diesem Abschnitt soll die graphentheoretische Grundlage für die Behandlung ungerichteter Graphen im weiteren Verlauf der Arbeit gelegt werden. Zunächst erweitern wir eine allgemein gebräuchliche Definition ungerichteter Graphen als Paar $G = (V, E)$ einer endlichen Knotenmenge V und einer Kantenmenge $E \subseteq \{\{v, w\} \mid v, w \in V\}$, indem wir Bewertungen auf den Knoten und Kanten hinzufügen:

Definition 1.1.1
Seien V eine endliche Menge, $E \subseteq \{\{v, w\} \mid v, w \in V\}$, $d : E' \to \mathbb{R}_{\geqslant 0}$ eine Distanzfunktion für die Kanten mit $E \subseteq E'$ und $g : V' \to \mathbb{R}_{\geqslant 0}$ eine Gewichtung auf den Knoten mit $V \subseteq V'$. Dann heißt $G = (V, E, d, g)$ ungerichteter Graph mit (Kanten- und Knoten-) Bewertungen. $d(G)$ bzw. $g(G)$ bezeichne im weiteren das gesamte Kanten- bzw. Knotengewicht eines Graphen G, d.h.

$$d(G) := \sum_{e \in E} d_e \ ,$$
$$g(G) := \sum_{v \in V} g_v \ .$$

Bemerkung 1.1.2

i. Da die Kanten eines Graphen als zweielementige Mengen definiert wurden, gibt es keine Kanten von einem Knoten $v \in V$ nach v selbst.
ii. Da E ebenfalls als Menge definiert wurde, gibt jeweils höchstens eine Kante $e = \{v, w\} \in E$ zwischen zwei Knoten $v, w \in V$.
iii. Betrachten wir einen Graphen $G = (V, E, \mathbf{1}, \mathbf{1})$, so ist $d(G) = |E|$ die Anzahl der Kanten in G; das gleiche gilt für die Anzahl der Knoten, d.h. $g(G) = |V|$.
iv. Ein Graph $G = (V, E, d, g)$ heißt vollständig, falls für $G' = (V, E, \mathbf{1}, g)$ mit $n := |V|$ gilt:

$$d(G') = \frac{1}{2} n(n-1) \ ,$$

d.h. falls $E = \{ \{v,w\} \mid v,w \in V \}$.

Definition 1.1.3
Ein Pfad P auf einem Graphen $G = (V,E,d,g)$ ist ein l-Tupel

$$\begin{aligned} P &= (v_1,\ldots,v_l), && (l \in \mathbb{N}) \\ \text{mit } v_i &\in V, && \forall i = 1,\ldots,l \\ \{v_i, v_{i+1}\} &\in E, && \forall i = 1,\ldots,l-1 \text{ (falls } l > 1) \,. \end{aligned}$$

Als Längen von P in G definieren wir

$$\begin{aligned} d(P) &:= \sum_{i=1}^{l-1} d_{\{v_i,v_{i+1}\}} \,, \\ g(P) &:= \sum_{i=2}^{l-1} g_{v_i} \,. \end{aligned}$$

Es bezeichne nun $\mathcal{P}_G(v,w)$ die Menge aller Pfade zwischen zwei Knoten $v,w \in V$:

$$\begin{aligned} \mathcal{P}_G(v,w) := \{ &(v_1,\ldots,v_l) \in V^l, l \in \mathbb{N} \mid \\ &v_i \in V, \quad \forall i = 1,\ldots,l \ \wedge \\ &\{v_i,v_{i+1}\} \in E, \quad \forall i = 1,\ldots,l-1 \ \wedge \\ &v_1 = v \ , v_l = w \} \,. \end{aligned}$$

Damit definiert man weiter für $V', V'' \subseteq V$ und $v \in V$

$$\begin{aligned} \mathcal{P}_G(V',V'') &:= \bigcup_{\substack{v \in V' \\ w \in V''}} \mathcal{P}_G(v,w) \\ \mathcal{P}_G(v,V') &:= \mathcal{P}_G(\{v\},V') \\ \mathcal{P}_G(V',v) &:= \mathcal{P}_G(V',\{v\}) \end{aligned}$$

Ein Pfad $P \in \mathcal{P}_G(v,w)$ mit

$$d(P) + g(P) \leqslant d(P') + g(P'), \quad \forall P' \in \mathcal{P}_G(v,w)$$

heißt ein kürzester Pfad zwischen v und w in G. Die Länge dieses kürzesten Pfades über Knoten und Kanten benutzen wir zur Definition des kleinsten Abstandes zweier Knoten:

$$d_{\min(G)}(v,w) := \min\{ d(P) + g(P) \mid P \in \mathcal{P}_G(v,w) \} \,.$$

Wie für die Pfadmengen weiter oben seien für $V', V'' \subseteq V$ und $v \in V$

$$d_{\min(G)}(V',V'') := \min\{ d(P) + g(P) \mid P \in \mathcal{P}_G(V',V'') \}$$

und analog wieder $d_{\min(G)}(v,V')$ bzw. $d_{\min(G)}(V',v)$ definiert.

Bemerkung 1.1.4
In obiger Definition des kleinsten Abstandes sei

$$\min\{ d(P) + g(P) \mid P \in \mathcal{P}_G(v,w) \} = \infty, \text{ falls } \mathcal{P}_G(v,w) = \emptyset \,.$$

Es folgen nun spezialisierende Begriffe für Pfade; dabei betrachte man einen Pfad P wie oben. P heißt geschlossen, falls $v_1 = v_l$. Er heißt einfach, falls $v_i \neq v_j$ für $i, j \in \{1, \ldots, l\}$, $i \neq j$. Ein Kreis ist ein geschlossener Pfad mit $\{v_i, v_{i+1}\} \neq \{v_j, v_{j+1}\}$ für $i, j \in \{1, \ldots, l-1\}$, $i \neq j$; in einem Kreis sind also alle Kanten verschieden. Für einen Zyklus gilt zusätzlich, daß $v_i \neq v_j$ für $i, j \in \{1, \ldots, l-1\}$, $i \neq j$, d.h. $(v_1, \ldots, v_{l-1})$ ist einfacher Pfad in G. Ein Graph heißt azyklisch, wenn er keinen Zyklus enthält.

Man erhält eine Relation auf V durch folgende

Definition 1.1.5

$$v \overset{*}{\underset{G}{\longleftrightarrow}} w \;:\Longleftrightarrow\; \mathcal{P}_G(v, w) \neq \emptyset \; .$$

Man nennt zwei Knoten mit dieser Eigenschaft (stark) zusammenhängend. Ein Graph heißt zusammenhängend, falls gilt:

$$v \overset{*}{\underset{G}{\longleftrightarrow}} w, \quad \forall v, w \in V \; .$$

Bemerkung 1.1.6
Offensichtlich ist dies eine Äquivalenzrelation auf V. Die Äquivalenzklassen dieser Relation $[v]_{\overset{*}{\underset{G}{\longleftrightarrow}}}$ von $v \in V$ nennt man die Zusammenhangskomponenten des Graphen G.

Die direkten Nachbarn eines Knotens faßt man zusammen mit Hilfe von

Definition 1.1.7
Sei $G = (V, E, d, g)$ ein ungerichteter Graph, und sei ein Knoten $v \in V$ gegeben. Dann heißt

$$\mathrm{adj}_G(v) := \{\, w \in V \mid \{v, w\} \in E \,\}$$

Adjazenzliste oder Nachbarschaftsliste des Knotens v bezüglich G.
Man nennt $\gamma_G(v) := |\mathrm{adj}_G(v)|$ den Knotengrad von v bezüglich G.

Eine besondere Klasse von zusammenhängenden Graphen sind Bäume.

Definition 1.1.8
Ein zusammenhängender Graph $G = (V, E, d, g)$ heißt Baum, falls
$G' := (V, E \backslash \{e\}, d, g)$ für jede beliebige Kante $e \in E$ nicht zusammenhängt.

Satz 1.1.9
Betrachte einen zusammenhängenden Graphen $G = (V, E, d, g)$. Dann gilt:

$$G \text{ ist Baum} \iff G \text{ ist azyklisch.} \iff |E| = |V| - 1$$

Beweis.
Siehe etwa in Balakrishnan [Bal95] die Sätze 1.5 bis 1.9. □

Definition 1.1.10
Sei $G = (V, E, d, g)$ ein ungerichteter Graph.
Ein Graph $G' = (V', E', d', g')$ heißt Teilgraph von G, symbolisch $G' \subseteq G$, falls gilt:

$$V' \subseteq V, \; E' \subseteq E$$
$$g|_{V'} = g'|_{V'}, \; d|_{E'} = d'|_{E'}$$

Gilt $V'' \subseteq V'$, und ist G' zusammenhängend, so heiße G' ein die Menge V'' aufspannender Teilgraph von G. Ist G' außerdem Baum, so ist G' ein die Knotenmenge V'' aufspannender Teilbaum von G.

1.2 Graphentheoretische Notationen II - gerichtete Graphen

Parallel zum vorangegangenen Abschnitt werden nun Begriffe rund um gerichtete Graphen mit Kantenbewertung eingeführt.

Definition 1.2.1
Seien N eine endliche Menge, $A \subseteq N \times N$ eine irreflexive Relation auf N und $d : A' \to \mathbb{R}_{\geqslant 0}$ mit $A \subseteq A'$ eine Funktion, die die Längen der Kanten angibt. Dann heißt $D = (N, A, d)$ gerichteter Graph oder Digraph mit (Kanten-) Bewertung d.
Wie im ungerichteten Fall bezeichne $d(D)$ die gesamte Länge des Digraphen, d. h.

$$d(D) := \sum_{a \in A} d_a \ .$$

Bemerkung 1.2.2

i. Wegen der Irreflexivität der Relation A gilt:

$$(v, v) \notin A, \quad \forall v \in N \ ,$$

d.h. es gibt keine Kante von einem Knoten V in sich selbst.

ii. Weil A als Menge definiert ist, gibt es höchstens eine Kante $(v, w) \in A$ von v nach w im Digraphen D.

iii. Ein Digraph $D = (N, A, d)$ heißt vollständig, falls für den Digraphen $D' := (N, A, \mathbf{1})$ mit $n := |N|$ gilt:

$$d(D') = n(n-1) \ ,$$

d.h. falls $A = N \times N \setminus \{ (v, v) \mid v \in N \}$.

Die Pfaddefinition für Digraphen erfolgt analog zu der für ungerichtete Graphen:

Definition 1.2.3
Ein gerichteter Pfad P auf einem Digraphen $D = (N, A, d)$ ist ein l-Tupel

$$\begin{aligned} P = & (v_1, \ldots, v_l), && (l \in \mathbb{N}) \\ \text{mit } & v_i \in N, && \forall i = 1, \ldots, l \\ & (v_i, v_{i+1}) \in A, && \forall i = 1, \ldots, l-1 \text{ (falls } l > 1) \ . \end{aligned}$$

Als Länge von P in D definieren wir

$$d(P) := \sum_{i=1}^{l-1} d_{(v_i, v_{i+1})} \ .$$

Es bezeichne nun $\mathcal{P}_D(v,w)$ die Menge aller gerichteten Pfade zwischen zwei Knoten $v,w \in N$, genauer:

$$\begin{aligned}\mathcal{P}_D(v,w) := \{\, (v_1,\dots,v_l) \in N^l, l \in \mathbb{N} \mid\ & \\ & v_i \in N, \quad \forall i = 1,\dots,l \ \wedge \\ & \{v_i, v_{i+1}\} \in A, \quad \forall i = 1,\dots,l-1 \ \wedge \\ & v_1 = v \ , v_l = w \,\} \ .\end{aligned}$$

Dann heißt ein gerichteter Pfad $P \in \mathcal{P}_D(v,w)$ mit

$$d(P) \leqslant d(P'), \quad \forall P' \in \mathcal{P}_D(v,w)$$

ein kürzester gerichteter Pfad von v nach w in D. Die Länge eines solchen Pfades benutzen wir in der Definition des kleinsten Abstandes zweier Knoten:

$$d_{\min(D)}(v,w) := \min\{\, d(P) \mid P \in \mathcal{P}_D(v,w) \,\}, \quad \forall v,w \in N.$$

Dies kann man verallgemeinern auf den kleinsten Abstand eines Knotens von einer Knotenmenge $N' \subseteq N$:

$$d_{\min(D)}(v,N') := \min\{\, d(P) \mid P \in \mathcal{P}_D(v,w), w \in N' \,\}, \quad \forall v \in N.$$

Bemerkung 1.2.4
In obiger Definition des kleinsten Abstandes sei

$$\min\{\, d(P) \mid P \in \mathcal{P}_D(v,w) \,\} = \infty, \text{ falls } \mathcal{P}_D(v,w) = \emptyset \ .$$

Auch die folgenden Bezeichnungen sind Übertragungen vom ungerichteten Fall. Sei P wie in der vorhergehenden Definition. Der Pfad heißt geschlossen, falls $v_1 = v_l$. Er heißt einfach, falls $v_i \neq v_j$ für $i,j \in \{1,\dots,l\}$, $i \neq j$. Ein gerichteter Kreis ist ein geschlossener Pfad mit $(v_i,v_{i+1}) \neq (v_j,v_{j+1})$ für $i,j \in \{1,\dots,l\}$, $i \neq j$; in einem Kreis sind also alle Kanten verschieden. Für einen gerichteten Zyklus gilt zusätzlich, daß $v_i \neq v_j$ für $i,j \in \{1,\dots,l-1\}$, $i \neq j$, d.h. $(v_1,\dots,v_{l-1})$ ist einfacher gerichteter Pfad in D. Ein Digraph heiße azyklisch, wenn er keinen gerichteten Zyklus enthält.

Der Zusammenhangsbegriff im Falle von Digraphen muß im Gegensatz zu Abschnitt 1.1 ein wenig differenzierter abgehandelt werden.

Definition 1.2.5
Sei $D = (N,A,d)$ ein bewerteter Digraph, $v,w \in N$ seien Knoten darin.

i. Es heißt w erreichbar von v, falls

$$v \underset{D}{\overset{*}{\rightarrow}} w \ :\Longleftrightarrow\ \mathcal{P}_D(v,w) \neq \emptyset \ .$$

ii. Es heißen v und w stark zusammenhängend, falls

$$v \underset{D}{\overset{*}{\longleftrightarrow}} w \ :\Longleftrightarrow\ v \underset{D}{\overset{*}{\rightarrow}} w \wedge w \underset{D}{\overset{*}{\rightarrow}} v \ .$$

Bemerkung 1.2.6
Die zweite Relation ist offensichtlich reflexiv, transitiv und symmetrisch und damit eine Äquivalenzrelation auf den Knoten N. Die Äquivalenzklassen $[v]_{\underset{D}{\overset{*}{\leftrightarrow}}}$ bzgl. $v \in N$ nennt man (starke) Zusammenhangskomponenten von D. Ein Digraph mit einer einzigen Zusammenhangskomponente heißt (stark) zusammenhängend.

Bevor man zur Definition von Bäumen im gerichteten Fall gelangt, wird noch die nun folgende Definition gebraucht.

Definition 1.2.7
Sei $D = (N, A, d)$ ein Digraph mit Bewertung. Die Nachbarschaftslisten (engl.: adjacency list) eines Knoten $v \in N$ bezüglich D sind die Mengen:

$$\begin{aligned}\mathrm{adj}_D^+(v) &:= \{\, w \in N \mid (v, w) \in A \,\}\\ \mathrm{adj}_D^-(v) &:= \{\, w \in N \mid (w, v) \in A \,\}\ .\end{aligned}$$

Man nennt $\gamma_D^+(v) := |\mathrm{adj}_D^+(v)|$ den Ausgangs- und $\gamma_D^-(v) := |\mathrm{adj}_D^-(v)|$ den Eingangsgrad von $v \in N$.

Definition 1.2.8
Ein azyklischer Digraph $T = (N, A, d)$ heißt gerichteter Wurzelbaum, falls folgende Bedingungen für die Eingangsgrade der Knoten aus N zutreffen:

$$\begin{gathered}\mathrm{adj}_T^-(v) \leqslant 1, \quad \forall\, v \in N,\\ |\{\, v \in N \mid \mathrm{adj}_T^-(v) = 0 \,\}| = 1.\end{gathered}$$

$r \in N$ mit $\mathrm{adj}_T^-(r) = 0$ ist die eindeutig bestimmte Wurzel von T, die Knoten $l \in N$ mit $\mathrm{adj}_T^+(l) = 0$ sind die Blätter von T. Alle anderen Knoten heißen innere Knoten von T.

Definition 1.2.9
Sei $D = (N, A, d)$ ein bewerteter gerichteter Graph. Ein Digraph $D' = (N', A', d')$ heißt Teilgraph von D, in Zeichen $D' \subseteq D$, falls gilt:

$$N' \subseteq N,\ A' \subseteq A \text{ und } d|_{A'} = d'|_{A'}\ .$$

Sind außerdem $N'' \subseteq N'$ und D' zusammenhängend, so sprechen wir davon, daß D' die Knotenmenge N'' aufspannt. Ist D' weiterhin ein Wurzelbaum, so heißt D' ein die Knotenmenge N'' aufspannender Wurzelbaum in D.

1.3 Modifiziertes Steiner-Problem

Wir wollen nun die Aufgabe darstellen, die dieser Arbeit zugrunde liegt. Gegeben sei ein zusammenhängender ungerichteter Graph $G = (V, E, d, g)$ mit einer Distanzfunktion d auf einer Menge E', für die $E \subseteq E'$ gilt (gemäß Abschnitt 1.1). Die Knotenbewertung g spielt zunächst keine Rolle. Diese Situation könnte z.B. einem physikalischen Hintergrund entspringen; man stelle sich vor, der Graph G modelliere eine (ebene oder räumliche) Gitterstruktur.

Weiterhin sei eine Teilmenge $Z \subseteq V$ der Knoten gegeben; die Knotenmenge V ist also mit $S := V \backslash Z$ aufgeteilt in die Menge Z der speziellen Knoten und die Menge S der Steiner-Knoten. Mit diesen Bezeichnungen können wir nun das sogenannte Steiner-(Baum-) Problem auf ungerichteten Graphen formulieren:

$$\begin{aligned}&\min_T \quad d(T) && (\mathrm{USTP}_{G,Z})\\ &T \text{ ist Teilbaum von } G, \text{ der } Z \text{ aufspannt.}\end{aligned}$$

Dabei steht $\mathrm{USTP}_{G,Z}$ für das „Undirected Steiner Tree Problem“ auf einem Graphen G bezüglich einer Menge spezieller Knoten Z. Man nennt einen Baum, der dieses Problem löst, einen Steiner-Baum. Die Suche nach einem kürzesten Pfad zwischen zwei Knoten ist ein Spezialfall von $\mathrm{USTP}_{G,Z}$ (für $|Z| = 2$), die Suche nach einem minimalen aufspannenden Baum ein anderer (für $Z = V$).

Oftmals formuliert man obiges Problem etwas allgemeiner, indem man nämlich in einer größeren Menge nach einer Lösung sucht:

$$\min_H \quad d(H) \qquad (\mathrm{USP}_{G,Z})$$

H ist zusammenhängender Teilgraph von G, der Z aufspannt.

Klar ist, daß beide Minimierungsprobleme deshalb Lösungen besitzen, weil der Ausgangsgraph G als zusammenhängend vorausgesetzt wurde.

Bevor wir die Äquivalenz der beiden Aufgabenstellungen zeigen, wollen wir diese zunächst erweitern. Oft sucht man nämlich nicht nur nach einem Teilgraph bzw. -baum, der gewisse Knoten des gegebenen Graphen sicher enthält, sondern vielmehr nach einem, welcher zusätzlich weitere Knoten $z' \in Z' \subseteq S$ „möglichst nah“ passiert. Wir wollen die Knoten dieser Menge Z' Sensor- oder Steuerknoten nennen; diese Bezeichnung hat mit dem physikalischen Hintergrund der Aufgabe zu tun. Der Graph G modelliert dabei ein reales Gitter, auf dem sich Schwingungsenergie bewege. Die Knoten der Menge Z sind „Erdungspunkte“, an denen Energie in das Gitter eintritt bzw. daraus abfließt. Der Steiner-Baum ist der Weg, den die Energie durch das Gitter nimmt. Will man nun die auftretenden Schwingungen an bestimmten Knoten der Menge Z' messen bzw. steuern (d.h. dämpfen oder verstärken), so wird man, um beste Messungen zu erhalten bzw. die größte Wirkung auszuüben, diesen Steiner-Baum eben auch „möglichst nah“ an den festen Knoten $z' \in Z'$ vorbeiführen wollen. Die Benennung der Knoten $z' \in Z'$ als Sensor- oder Steuerknoten ergibt sich also aus ihrer Interpretation in der gerade geschilderten Anwendung.

Es bleibt noch eine Bemerkung zu den Knoten $z \in Z$: In den (sehr zahlreichen) Veröffentlichungen, die das Steiner-Problem betreffen, wurden zuweilen die Elemente von Z (statt derer aus $S = V \setminus Z$) als Steiner-Knoten bezeichnet, etwa bei Kou et al. in [KMB81]. Hier jedoch wollen wir, wie oben schon geschehen, die Knoten aus S als Steiner-Knoten bezeichnen, was sowohl mit den ersten Arbeiten zum Thema (etwa [GiP68]) als auch mit den meisten weiteren wichtigen Artikeln ([Ane80, Bea84, WWW86, RaC86]) bis hin zur neuesten Literatur (etwa [Bal95]) in Übereinstimmung steht.

Wir werden nun jene Forderung, die sich mit den Knoten $z' \in Z'$ verbindet, in die oben aufgestellten Probleme $\mathrm{USTP}_{G,Z}$ bzw. $\mathrm{USP}_{G,Z}$ einflechten. Genauso, wie beim ursprünglichen Steiner-Problem kürzere Kanten (d.h. Kanten e mit kleineren Kosten d_e) eher in die Lösung eingebaut werden sollten, muß man nun Steiner-Knoten um so eher in der Lösung berücksichtigen, je „näher“ sie den Knoten $z' \in Z'$ liegen; „näher“ deshalb, weil ja noch keine formale Spezifizierung eines Entfernungsbegriffs, kein Maß für die Nähe eines Knotens $v \in V$ zu Z' gefunden wurde.

Wir müssen jetzt also – parallel zur Bewertung der Kanten – eine Funktion auf V einführen, die jedem Knoten Kosten zuweist, welche von seiner

Lage relativ zu den $z' \in Z'$ abhängen werden. Diesen Zweck kann die Knotenbewertung g des gegebenen Graphen $G = (V, E, d, g)$ erfüllen. Man beachte, daß in den folgenden veränderten Formulierungen von USTP$_{G,Z}$ und USP$_{G,Z}$ die Menge Z' nicht explizit auftaucht; man wird später Z' vielmehr über die Definition von g indirekt einfließen lassen.

Für einen Graphen G wie oben und eine Menge $Z \subseteq V$ wollen wir nach Lösungen suchen für

$$\min_{T} \quad d(T) + g(T) \tag{USTP'$_{G,Z}$}$$

T ist Teilbaum von G, der Z aufspannt,

bzw. für

$$\min_{H} \quad d(H) + g(H) \tag{USP'$_{G,Z}$}$$

H ist zusammenhängender Teilgraph von G, der Z aufspannt.

Satz 1.3.1
Die beiden Probleme USTP'$_{G,Z}$ und USP'$_{G,Z}$ sind äquivalent in dem Sinne, daß die optimalen Werte der Zielfunktion übereinstimmen.

Beweis.

i. Jede Lösung von USTP'$_{G,Z}$ ist als zusammenhängender Teilgraph von G, der Z aufspannt, natürlich zulässig für USP'$_{G,Z}$.

$$\Rightarrow \min(\text{USTP'}_{G,Z}) \geqslant \min(\text{USP'}_{G,Z})$$

ii. Sei $G_0 = (V_0, E_0, d, g)$ eine Lösung von USP'$_{G,Z}$ mit $|V_0| = n$ und $|E_0| = m \geqslant n - 1$. Es ist zu zeigen, daß ein Teilbaum T von G existiert, der Z aufspannt und für den gilt:

$$d(T) + g(T) \leqslant d(G_0) + g(G_0)\ .$$

Falls $m = n - 1$, ist nach Satz 1.1.9 G_0 schon ein die Menge Z aufspannender Baum.
Falls $m - (n - 1) = k > 0$ gilt, muß G_0 einen Zyklus P_1 enthalten. Wir wählen eine Kante $e_1 \in E_0$ dieses Zyklus aus und erhalten durch $G_1 := (V_0, E_1, d, g)$ einen zusammenhängenden Teilgraphen von G, der die Knoten in Z aufspannt, wobei $E_1 := E_0 \setminus \{e_1\}$ sei. Weiter gilt, da d nicht-negativ ist, insbesondere $d(e_1) \geqslant 0$ und damit

$$d(G_1) + g(G_1) \leqslant d(G_0) + g(G_0)\ ;$$

außerdem ist $|E_1| = m - 1$.
Wir führen diesen Schritt induktiv k-mal fort und erhalten jeweils Teilgraphen G_i von G, die Z aufspannen mit

$$d(G_i) + g(G_i) \leqslant d(G_{i-1}) + g(G_{i-1}), \quad \forall 1 \leqslant i \leqslant k\ ,$$
$$|E_i| = m - i, \quad \forall 1 \leqslant i \leqslant k\ .$$

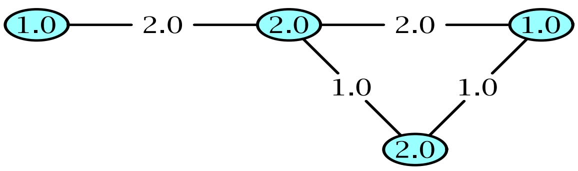

Abbildung 1.1: Beispielgraph G zur Transformation gemäß (1.1)

$G_k = (V_0, E_k, d, g)$ schließlich ist dann nach Satz 1.1.9 Teilbaum von G, welcher Z aufspannt, also Lösung von USTP'$_{G,Z}$ mit

$$d(G_k) + g(G_k) \leqslant d(G_0) + g(G_0) \;.$$

$$\Rightarrow \min(\text{USTP'}_{G,Z}) \leqslant \min(\text{USP'}_{G,Z})$$

Nach i. und ii. gilt deshalb die Behauptung.

□

Corollar 1.3.2
Nach Satz 1.3.1 genügt es, eine Lösung für USTP'$_{G,Z}$ zu finden. Diese löst dann auch USP'$_{G,Z}$.

Corollar 1.3.3
Falls man einen Graphen $G = (V, E, d, g)$ hat mit $d(e) > 0$ für alle $e \in E$, so ersieht man aus obigem Beweis, daß die Lösung von USP'$_{G,Z}$ sogar ein Baum sein muß.

Wir kennen bis jetzt zwei Formulierungen des modifizierten Steiner-Problems auf ungerichteten Graphen; man erhält eine weitere, wenn man ungerichtete Graphen auf Digraphen transformiert. Hierzu wird einem zusammenhängenden Graphen $G = (V, E, d, g)$ durch die Definitionen

$$\begin{gathered} N := V\;;\quad A := \bigcup_{\{v,w\}\in E} \{\,(v,w),\,(w,v)\,\} \\ d' : A \to \mathbb{R}_{\geqslant 0};\; d'_{(v,w)} := d_{\{v,w\}} + g_w,\;\; \forall (v,w) \in A \end{gathered} \tag{1.1}$$

ein Digraph $D_G = (N, A, d')$ zugeordnet. Dieser gerichtete Graph hängt natürlich ebenfalls zusammen. Liegt zum Beispiel der in Abbildung 1.1 gezeigte Graph G vor, so entspricht diesem gemäß (1.1) der in der Abbildung darunter dargestellte Digraph D_G.

Ist wieder eine Menge $Z \subset N = V$ mit $r \in Z$ gegeben, so kann man mit Hilfe von $Z' := Z \backslash \{r\}$ ein Steiner-Problem DSTP$_{D_G,Z}$ – das steht für

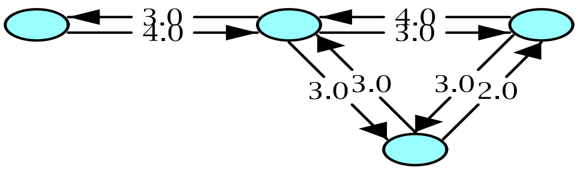

Abbildung 1.2: Gemäß (1.1) von G abgeleiteter Digraph D_G

„Directed Steiner Tree Problem“ – auf dem Digraphen aufstellen:

$$\min_{T'} \quad d'(T') \qquad (\mathrm{DSTP}_{D_G,Z})$$

T' ist ein Z' aufspannender Wurzelbaum in D_G mit Wurzel $r \in Z$.

Der folgende Satz beruht im wesentlichen auf der Eigenschaft eines Wurzelbaumes $T' = (N', A', d')$ mit Wurzel $r \in N'$, daß

$$\begin{aligned} & \mathrm{adj}^-_{T'}(v) = 1\,, \quad \forall v \in N' \setminus \{r\} \\ \text{bzw.}\ & \mathrm{adj}^-_{T'}(r) = 0\,. \end{aligned}$$

Satz 1.3.4
Sei $Z \subseteq V$ die Menge spezieller Knoten bezüglich eines Graphen $G = (V, E, d, g)$ mit $g(z) = 0,\ \forall z \in Z$. Dann gilt die Gleichheit

$$\min(\mathit{USTP'}_{G,Z}) = \min(\mathit{DSTP}_{D_G,Z})$$

mit dem gemäß (1.1) von G abgeleiteten Digraphen D_G.

Beweis.

i. Betrachte eine zulässige Lösung von $\mathrm{DSTP}_{D_G,Z}$, nämlich den Wurzelbaum $T' = (N', A', d')$. Man konstruiere einen Baum $\hat{T} = (\hat{V}, \hat{E}, d, g)$ in G durch

$$\hat{V} := N'\ ; \quad \hat{E} := \{\, \{v, w\} \mid (v, w) \in A' \,\}\ .$$

$\hat{T}$ ist Teilbaum von G; er spannt Z auf. Deshalb ist $\hat{T}$ zulässig für $\mathrm{USTP'}_{G,Z}$; weiter gilt

$$\begin{aligned} d'(T') &= \sum_{a \in A'} d'_a = \sum_{(v,w) \in A'} d_{\{v,w\}} + g_w = \\ &= \sum_{e \in \hat{E}} d_e + \sum_{v \in \hat{V} \setminus \{r\}} g_v = \\ &= \sum_{e \in \hat{E}} d_e + \sum_{v \in \hat{V}} g_v - \underbrace{g_r}_{=0,\ \text{da}\ r \in Z} = \\ &= d(\hat{T}) + g(\hat{T})\ . \end{aligned}$$

Zu jeder Lösung von $\text{DSTP}_{D_G,Z}$ gehört demnach eine Lösung des Problems $\text{USTP'}_{G,Z}$ mit gleichem Zielfunktionswert. Das bedeutet, daß

$$\min(\text{USTP'}_{G,Z}) \leqslant \min(\text{DSTP}_{D_G,Z}) \ .$$

ii. Betrachte jetzt eine zulässige Lösung $\hat{T} = (\hat{V}, \hat{E}, d, g)$ von $\text{USTP'}_{G,Z}$. Mit beliebigem $r \in Z$ konstruiert man daraus induktiv mit Hilfe von Niveaumengen (Suche in die Breite auf $\hat{T}$) eine zulässige Lösung T' für $\text{DSTP}_{D_G,Z}$; dabei sei $n := |\hat{V}|$.

$$A_0 := \varnothing \text{ und } N_0 := \{r\}$$

und für alle i mit $0 \leqslant i < n-1$:

$$A_{i+1} := \left\{ (v,w) \mid v \in N_i, w \in \hat{V} \setminus \bigcup_{j=0}^{i} N_j, \{v,w\} \in \hat{E} \right\} ,$$
$$N_{i+1} := \left\{ w \in \hat{V} \mid \exists v \in N_i : (v,w) \in A_{i+1} \right\}$$

Es sei $D_G = (N, A, d')$ der Digraph, wie er durch Gleichung (1.1) definiert wird. Weiter seien für $0 \leqslant i \leqslant n-1$

$$N^{(i)} := \bigcup_{j=0}^{i} N_j \ ,$$
$$A^{(i)} := \bigcup_{j=0}^{i} A_j \ .$$

Wie zeigen per Induktion nach i, daß $T^{(i)} := \left(N^{(i)}, A^{(i)}, d'\right)$ für i wie oben Wurzelbaum mit Wurzel r ist.
Zunächst ist $T^{(0)} = \left(\{r\}, \varnothing, d'\right)$ azyklisch, die Wurzelbaumeigenschaften (1.2.8) gelten. Sei dies nun für $T^{(i)}$ mit $0 \leqslant i < n-1$ der Fall. Nach Auswahl von A_{i+1} ist $T^{(i+1)}$ azyklisch, und es bleibt $\text{adj}^-_{T^{(i+1)}}(r) = 0$. Nach Wahl von N_{i+1} gilt für alle $v \in N^{(i+1)} \setminus \{r\}$, daß $\text{adj}^-_{T^{(i+1)}}(v) = 1$. Daher gilt die Behauptung auch für $T^{(i+1)}$.
Weil $\hat{T}$ zusammenhängend ist, gilt $N^{(n-1)} = \hat{V}$. Deshalb ist $T' := T^{(n-1)}$ zulässig für $\text{DSTP}_{D_G,Z}$. Wie im vorigen Beweisteil ist

$$d(\hat{T}) + g(\hat{T}) = d'(T')$$

und damit

$$\min(\text{USTP'}_{G,Z}) \geqslant \min(\text{DSTP}_{D_G,Z}) \ .$$

□

Bemerkung 1.3.5

i. In Satz 1.3.1 auf Seite 8 und in obigem Satz 1.3.4 wurde die Notation min(.) für den optimalen Zielfunktionswert des Problems (.) benutzt.

ii. Die Voraussetzung in Satz 1.3.4, daß die Knotenbewertung auf den speziellen Knoten verschwinden soll, stellt keine echte Einschränkung dar, da die Knoten $z \in Z$ ja in jedem Falle in einer zulässigen Lösung von USTP'$_{G,Z}$ bzw. DSTP$_{D_G,Z}$ enthalten sein müssen.
Ist diese Voraussetzung nicht gegeben, betrachte man statt G den Graphen $G' = (V, E, d, g')$, wobei g' durch

$$g'(v) := \begin{cases} g(v)\,, & v \in V \setminus Z \\ 0\,, & v \in Z \end{cases}$$

festgelegt wird. Dann gilt für die optimalen Zielfunktionswerte

$$\min(\mathrm{USTP'}_{G',Z}) = \min(\mathrm{USTP'}_{G,Z}) - \sum_{z \in Z} g(z)$$

und entsprechend

$$\min(\mathrm{DSTP}_{D_{G'},Z}) = \min(\mathrm{DSTP}_{D_G,Z}) - \sum_{z \in Z'} g(z)\ .$$

iii. Die Interpretation in (1.1) von ungerichteten Graphen als Spezialfall gerichteter Graphen, deren Kantenmenge eine symmetrische Relation darstellt, ist nicht nur in obigem Zusammenhang nützlich. Sie wird noch häufiger auftauchen.

Wir haben nun die Fragestellung dieser Arbeit, das durch Knotenbewertungen erweiterte Steiner-Problem, kennengelernt. Der Großteil der vorhandenen Literatur beschäftigt sich mit USTP$_{G,Z}$, einige Arbeiten wie z.B. [Won82] auch mit DSTP$_{D,Z}$. Der Leser sei auf die Bibliographie der vorliegenden Arbeit verwiesen. Einen breiten Überblick über die vorhandenen Ansätze sowohl für exakte Lösungsalgorithmen als auch für Heuristiken liefert Winter in [Win87]. Dieser Autor hat wichtige Heuristiken auch klassifiziert (zusammen mit MacGregor-Smith in [WiM92]).

Bevor wir daran gehen, Lösungen für USTP'$_{G,Z}$ zu suchen, definieren wir eine Klasse zur Repräsentierung sowohl gerichteter als auch ungerichteter Graphen innerhalb von C++ - Programmen. Diese Klasse stellt die grundlegende Datenstruktur für alle weiteren Algorithmen, welche in dieser Arbeit vorgestellt werden, bereit.

1.4 Die Klasse `TGr`

1.4.1 Anforderungen an die Klasse `TGr`

Zunächst ist festzulegen, welchen Anforderungen das Interface, d.h. der von außen zugängliche Teil der zu deklarierenden Klasse `TGr`, genügen soll. Hier sind sie wichtigsten:

Repräsentierung (un-) gerichteter Graphen

Die entstehende Datenstruktur soll sich dafür eignen, sowohl gerichtete als auch ungerichtete Graphen mit Knoten- und Kantenbewertungen aufzunehmen. Wird die Klasse dafür eingesetzt, einen ungerichteten Graphen der

Gestalt $G = (V, E, d, g)$ zu repräsentieren, muß der benutzende Programmteil selbst dafür Sorge tragen, die Symmetrie in der Kantenmenge E und in der Bewertungsfunktion d zu erhalten.

Kapselung der Datenstruktur

Nach außen darf die interne Datenstruktur (z.B. Zeiger-verkettete Listen zur Darstellung von Adjazenzlisten) nicht sichtbar sein. Dadurch stellt man sicher, daß bei einer etwaigen Änderung der Implementierung dieser internen Struktur andere, die Klasse `TGr` benutzende Algorithmen nicht angepaßt werden müssen. Außerdem verringert man hierdurch das Risiko, daß die interne Struktur durch falsche Benutzung korrumpiert wird.

Zugriff auf Knoten und Kanten

Man nimmt ohne Einschränkung an, daß für den dargestellten ungerichteten Graphen G bzw. gerichteten Graphen D mit $n \in \mathbb{N}$ gelte:

$$\begin{aligned} G &= (\,\{1, \ldots, n\}, E, d, g\,) \text{ bzw.} \\ D &= (\,\{1, \ldots, n\}, A, d\,)\ . \end{aligned}$$

Der Zugriff auf die Knoten des repräsentierten Graphen soll dann durch Angabe eines $v \in \{1, \ldots, n\}$ erfolgen; die Kanten werden entweder innerhalb einer Iteration über alle Kanten, die in einen Knoten münden, angesprochen oder aber analog dem Zugriff auf die Knoten durch Angabe eines Paars (v, w) mit $v, w \in \{1, \ldots, n\}$. Wie eine derartige Iteration aussieht, wenn die eigentliche Datenstruktur (gemäß der vorhergehenden Bemerkung) verborgen ist, wird weiter unten besprochen.

Dynamische Speicherverwaltung

Ein Graph soll eine stets konstante Anzahl von Knoten besitzen; er kann jedoch vollständig gelöscht werden, um dann mit einer neuen Anzahl n von Knoten wiederaufgebaut zu werden. Dadurch wird auch eine weitgehende Kontrolle über den tatsächlichen Speicherbedarf eines Graphen möglich. Operationen wie Einfügen und Löschen einer Kante oder das Ändern eines Kantengewichts sollen durch Methoden der Klasse `TGr` verfügbar gemacht werden.

Ein- und Ausgabeoperationen

Natürlich müssen zwei Methoden definiert werden, nämlich eine zur Eingabe und eine zur Ausgabe von Instanzen der Klasse `TGr`; diese Aufgabe übernehmen die Extractor- bzw. Insertor-Operatoren $\gg$ bzw. $\ll$. Für alle Klassen, die noch deklariert werden, gehören die Ein-/Ausgabeoperatoren zur Grundausstattung; wir werden – soweit sie keine besonderen Merkmale aufweisen – später nicht mehr gesondert auf sie eingehen.

1.4.2 Deklaration der Klasse `TGr`

Es werden nun die nichttrivialen Elementfunktionen der Klasse `TGr` näher erläutert; die vollständige Deklaration dieser Klasse findet man in Abbildung 1.3. Die darin verwendeten Klassen `TGrItem` und `TGrIterator` werden später noch genauer dargestellt, wenn es um die Implementierung von `TGr` gehen wird. Einzig die Datenelemente `v` bzw. `e` sind jetzt schon von Interesse: Sie enthalten die Anzahl an Knoten resp. die Anzahl an Kanten des aktuell in einer Instanz von `TGr` gespeicherten gerichteten Graphen.

Alle Elementfunktionen sind darauf ausgelegt, daß ein Digraph der Gestalt $D = (N, A, d)$ mit

$$d : A \to \mathbb{R}_{\geqslant 0} \text{ und } g : N \to \mathbb{R}_{\geqslant 0} ,$$

wobei ohne Einschränkung $N = \{1, \ldots, n\}$ mit $n \in \mathbb{N}$ gelte, gespeichert wird. Die zusätzlich verwaltete Funktion g wird erst bei der Transformation eines ungerichteten Graphen $G = (V, E, d', g')$ in einen gerichteten Graphen wichtig (o.E. $V = \{1, \ldots, n\}$ mit $n \in \mathbb{N}$). Diese Transformation verläuft ähnlich wie in (1.1) auf Seite 9, nur daß die Knotengewichte des ungerichteten Graphen diesmal nicht mit in die Kantenbewertungen des entstehenden gerichteten Graphen einfließen:

$$\begin{gathered} N := V \; ; \quad g := g'|_N \\ A := \bigcup_{\{v,w\} \in E} \{ (v, w), (w, v) \} \\ d : A \to \mathbb{R}_{\geqslant 0};\ d_{(v,w)} := d'_{\{v,w\}},\ \forall (v, w) \in A \end{gathered} \tag{1.2}$$

Wir werden im Rest dieses Abschnittes nur noch von gerichteten Graphen sprechen, die mit Hilfe von `TGr` gespeichert werden, immer eingedenk der Möglichkeit, auch ungerichtete Graphen gemäß (1.2) als gerichtete Graphen zu interpretieren und ebenfalls in `TGr` abzulegen.

Geht man umgekehrt davon aus, daß in einer Instanz von `TGr` ein gerichteter Graph $D = (N, A, d)$ mit einer zusätzlichen Gewichtsfunktion $g : N \to \mathbb{R}_{\geqslant 0}$ abgelegt ist, dann kann man, falls zusätzlich

$$\{ (v, w), (w, v) \} \subset A \Rightarrow d_{(v,w)} = d_{(w,v)}$$

gilt, den Digraphen mittels der gleich folgenden Definitionen auch als ungerichteten Graphen $G = (V, E, d', g')$ interpretieren:

$$\begin{gathered} V := N \; ; \quad g' := g \\ E := \{ \{v, w\} \mid (v, w) \in A \} \\ d' : E \to \mathbb{R}_{\geqslant 0};\ d'_{\{v,w\}} := d_{(v,w)},\ \forall \{v, w\} \in E \end{gathered} \tag{1.3}$$

In der folgenden Erläuterung der einzelnen Elementfunktionen steht das Makro `FPTYPE` für einen Fließkomma-Typ, also etwa an Stelle von `float` oder `double`:

`TGr()` und `void rset(void)`
: Löscht den Graphen vollständig, setzt `v` und `e` auf 0. In diesem Zustand können keine Kanten eingefügt werden.

```
class TGr
    {
    private:
        int v,e;
        TGrItem *na;
        static TGrItem z;

        int _inse(int, int, FPTYPE);
    public:
        TGr();
        TGr(int n);
        TGr(const TGr &gr);
        ~TGr();

        void rset(void);
        void rset(int n);
        void rset(const TGr &gr);
        TGr &operator=(const TGr &gr);

        int  getv(void) const;
        int  gete(void) const;

        int  inse(int v, int w, FPTYPE d);
        int  ince(int v, int w, FPTYPE d);
        int  dele(int v, int w);
        int  exst(int v, int w) const;
        int  getw(int v, int w, FPTYPE &d) const;

        TGrItem  getNode(int v) const;
        TGrItem &getNode(int v);
        TGrItem  getEdge(int v, int w) const;
        TGrItem &getEdge(int v, int w);

        friend ostream &operator<<(ostream &, const TGr &);
        friend istream &operator>>(istream &, TGr &);

        friend class TGrIterator;
    };
```

Abbildung 1.3: Deklaration der Klasse TGr

`TGr(int n)` und `void rset(int n)`
Bereitet einen Graphen mit `n` Knoten vor.
Danach können Kanten eingefügt werden.

`TGr(const TGr &gr)` und `void rset(const TGr &gr)`
Kopiert die Instanz `gr`.
Die Elementfunktion wird vom Zuweisungsoperator benützt.

`int inse(int v, int w, FPTYPE d)`
Fügt die Kante `(v,w)` mit Gewicht `d` ein. Falls die Kante schon existiert, wird ihr Gewicht auf `d` gesetzt. Wie bei allen folgenden Funktionen derselben Bauart muß $v, w \in \{1, \ldots, n\}$ gelten;
Im Fehlerfall wird -1 zurückgeliefert, sonst 0.

`int ince(int v, int w, FPTYPE d)`
Erhöht das Gewicht der Kante `(v,w)` um `d` Falls die Kante noch nicht existiert, wird sie mit diesem Gewicht in den Graphen eingefügt.

`int dele(int v, int w)`
Löscht die Kante `(v,w)` aus dem Graphen.

`int exst(int v, int w) const`
Prüft, ob die Kante `(v,w)` im Graphen existiert. Falls ja, wird 1 zurückgegeben, sonst 0 (bzw. -1 im Fehlerfall).

`int getw(int v, int w, FPTYPE &d) const`
Liefert das Gewicht der Kante `(v,w)` in `d`.

`TGrItem getNode(int v) const`
Liefert eine Instanz der Struktur `TGrItem`, die den Knoten mit Nummer `v` beschreibt. Im Fehlerfall wird der ungültige Knoten mit -1 in der Kennung zurückgeliefert.

`TGrItem &getNode(int v)`
Liefert eine Referenz der Instanz, die den Knoten `v` beschreibt.

`TGrItem getEdge(int v, int w) const`
Liefert eine Instanz der Struktur `TGrItem`, die die Kante `(v,w)` beschreibt.

`TGrItem &getEdge(int v, int w)`
Liefert eine Referenz der Instanz, die die Kante `(v,w)` beschreibt.

1.4.3 Implementierung von TGr

In Instanzen der oben deklarierten Klasse sollen in der Hauptsache dünnbesetzte Graphen abgelegt werden; deshalb wurde `TGr` als Adjazenzstruktur realisiert (und nicht etwa als Adjazenzmatrix).

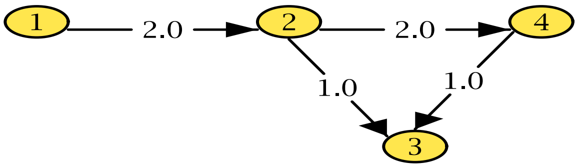

Abbildung 1.4: Digraph mit vier Knoten und Kanten;
jeder Knoten habe Gewicht 1.0.

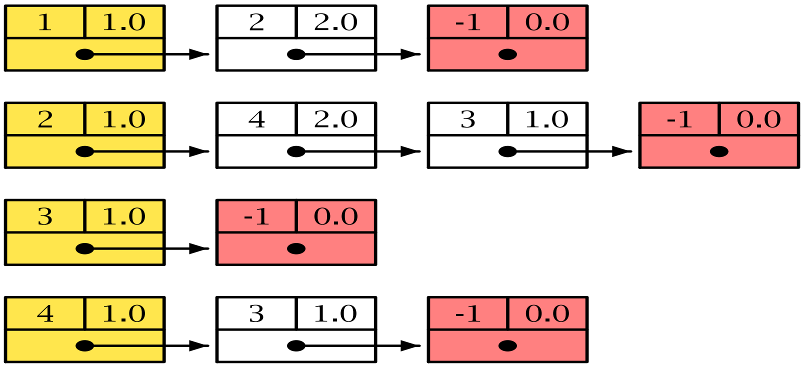

Abbildung 1.5: Adjazenzstruktur zum Digraph aus Abb. 1.4

Speicherbedarf

Speicherung in einer Adjazenzstruktur bedeutet zunächst, daß für die Knoteninformationen ein Array der Länge `TGr::v` verwandt wird. Für jeden Knoten v eines Graphen $D = (N, A, d)$ muß ein Zeiger auf den Anfang seiner Adjazenzliste, die die Information über alle Kanten der Form $(v, w) \in A$ enthält, vorhanden sein; da auch ungerichtete Graphen (nach Transformation gemäß (1.2)) mit Knotenbewertung g in `TGr` abgelegt werden sollen, muß außerdem für jeden Knoten ein `FPTYPE`-Datenfeld, das das Gewicht des Knotens aufnehmen kann, bereitgestellt werden. Schließlich sehen wir für jeden Knoten noch ein weiteres Datenfeld zur Aufnahme einer ganzen Zahl vor; diese Knotenkennung könnte z.B. einen zusätzlichen Knotennamen oder einen Verweis in ein externes Array aufnehmen.

Für jeden Knoten $w \in \mathrm{adj}_D^+(v)$ braucht man eine Speicherstelle für den Endknoten w selbst, eine für das Gewicht $d_{(v,w)}$ der Kante, die zu diesem Knoten führt, und einen Zeiger auf den nächsten Knoten in der (einfach verketteten) Adjazenzliste. Um das Ende einer jeden Adjazenzliste erkennen zu können, stellen wir dorthin ein Listenelement mit der ungültigen Kno-

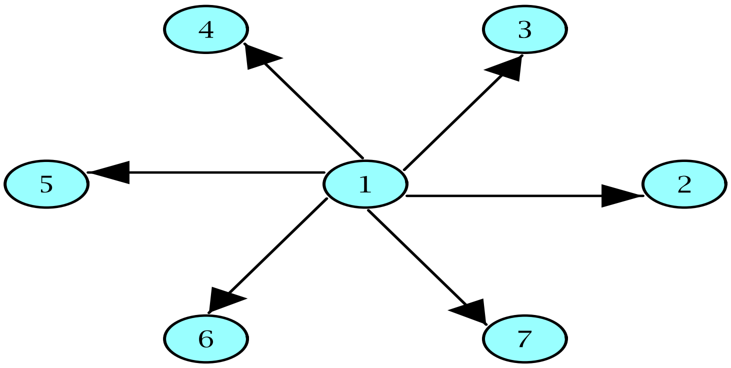

Abbildung 1.6: Beispiel für möglichen worst-case

tenkennung $w = -1$; diese Rolle einer Endmarkierung übernimmt `TGr::z`.

Insgesamt hat diese Art der Darstellung eines Graphen einen Bedarf von $O(|N| + |A|)$ an Speicherplatz für einen gerichteten Graphen $D = (N, A, d)$.

In Abbildung 1.5 findet man eine schematische Darstellung einer Adjazenzstruktur, die den Graphen der Abbildung 1.4 beschreibt. Alle dort dargestellten Datenelemente sind vom unten noch näher beschriebenen Typ `TGrItem`; die orange unterlegten gehören zum Knotenarray des Graphen, die rot unterlegten stellen die Endmarkierung `TGr::z` dar. Die vier anderen Instanzen von `TGrItem` sind in Adjazenzlisten eingebunden und enthalten die Kanteninformationen des dargestellten Graphen.

Komplexität der Elementfunktionen

Nach Betrachtung des von Instanzen von `TGr` benötigten Speichers sind noch die Komplexitäten der Elementfunktionen dieser Klasse zu nennen.

Die Initialisierungsfunktion `rset` hat in jeder der aufrufbaren Formen die worst-case Komplexität $O(|N| + |A|)$, da immer ein kompletter Graph gelöscht oder aufgebaut werden muß. Für den Assignment-Operator und den Copy-Konstruktor gilt dasselbe, denn diese rufen ja `rset` auf.

Die Funktionen `inse`, `ince`, `dele`, `exst`, `getw` und `getEdge` durchsuchen stets die Adjazenzliste eines Knotens des Graphen; sie haben deshalb die worst-case-Komplexität $O(|A|)$. Dieser worst-case tritt ein, wenn z.B. die Situation von Abbildung 1.6 vorliegt und nach dem letzten Kante in der Adjazenzliste des Knotens 1 gesucht wird. Eine Implementierung der Listen von Nachbarknoten als Heap statt als einfach verkettete Listen könnte eine bessere Komplexität garantieren, doch erscheint dies unnötig vor dem Hintergrund, daß hier Gittergraphen die größte Rolle spielen.

Als einzige der nicht-trivialen Elementfunktionen von `TGr` hat `getNode` die Komplexität $O(1)$, denn ihre Funktion besteht lediglich aus einem Zugriff auf einen Knoten; die Knoten aber sind in einem einfachen Feld abgelegt.

```
class TGrItem
    {
    enum { EXPSIZE = 1024};
    private:
        TGrItem *nxt;
        int nde;
        FPTYPE wgt;
        // (...)
    public:
        int      getn(void)  const;
        int     &getn(void);
        FPTYPE   getw(void)  const;
        FPTYPE  &getw(void);
        TGrItem &operator=(const TGrItem &item);

        void *operator new(size_t);
        void operator delete(void *, size_t);
        // (...)
    };
```

Abbildung 1.7: Teile der Klassendeklaration von `TGrItem`

1.4.4 `TGrItem` und `TGrIterator`

Im vorigen Unterabschnitt wurde bereits die von `TGr` betriebene Verwaltung der zu einem Knoten $v \in V$ benachbarten Knoten in einer einfach verketteten Adjazenzliste angesprochen. Instanzen der Klasse `TGrItem` repräsentieren die Knoten einer solchen Liste.

Die Klasse `TGrItem`

Dieser Datentyp wird sowohl zur Speicherung der Knoten im Knotenarray als auch als Typ für die Knoten der Adjazenzlisten verwendet. In Abbildung 1.7 sind die wichtigsten Teile der Klassendeklaration dargestellt (Auslassungen sind – wie später stets – durch (`...`) markiert).

Bemerkenswert sind hier lediglich die für `TGrItem` neu definierten Operatoren `new` und `delete`. Statt beim dynamischen Aufbau eines Graphen bzw. bei seiner Löschung jeden Knoten der Adjazenzliste einzeln zu allokieren bzw. freizugeben, ermöglichen diese speziell für diese Klasse definierten Operatoren, diese Aktionen blockweise, d.h. für `EXPSIZE` Instanzen von `TGrItem` gleichzeitig auszuführen. Dies verbessert sowohl Laufzeitverhalten als auch Speicherverbrauch.

Die Klasse `TGrIterator`

Es sei der Digraph $D = (N, A, d)$ in einer Instanz von `TGr` abgelegt. Dann ist es bis jetzt nicht möglich, über die Kanten in der Adjazenzliste $\mathrm{adj}_D^+(v)$ eines Knotens $v \in N$ zu iterieren, denn die interne Listenstruktur von `TGr` ist nach außen unsichtbar. Für eine derartige Operation, welche häufig

```
class TGrIterator
    {
    private:
        // (...)
    public:
        TGrIterator(const TGr &gr);
        void setg(const TGr &gr);

        TGrItem &fste(int i);
        TGrItem &fste(const TGr &gr, int v);
        TGrItem &nxte(void);
    };
```

Abbildung 1.8: Klassendeklaration von `TGrIterator`

benötigt wird, muß die Klasse `TGrIterator` benutzt werden (siehe Abbildung 1.8). Eine Instanz dieser Klasse bezieht sich auf einen Graphen `TGr`. Deshalb muß sie mit Angabe einer Referenz auf einen Graphen konstruiert werden; der Standard-Konstruktor ist explizit verboten. Der Bezugsgraph kann aber jederzeit mit Hilfe der Funktion `setg` verändert werden.

Um eine Iteration über die Adjazenzliste eines Knotens $v \in N$ einzuleiten, wird `fste` für diesen Knoten aufgerufen. Als Rückgabe erhält man eine Referenz des ersten Knotens der Liste; über die Referenz können die Datenelemente des Knotens auch verändert werden. Jeder folgende Aufruf von `nxte` liefert eine Referenz auf den nächsten Knoten der Adjazenzliste von v. Ist das Ende der Liste erreicht, wird `TGr::z` geliefert.

Das eben Ausgeführte wird vom nun folgenden Codefragment illustriert; es dient dazu, alle Kanten eines Graphen `gr` in den Datenstrom `cout` auszugeben:

```
    TGrIterator it(gr);
    TGrItem edge;

    for (int v=1; v<=gr->getv(); v++)
        for (
            edge=it.fste(v);
            edge.getn()!=-1;
            edge=it.nxte()
            )
            cout << edge;
```

Zunächst wird ein Iterator mit Bezug auf `gr` konstruiert; `edge` wird die Kanteninformationen aufnehmen. Danach wird, Knoten für Knoten, jeweils die Adjazenzliste bis zum Ende, welches an einer Knotenkennung mit Wert -1 erkennbar ist, durchlaufen. Die in `edge` gespeicherten Kanteninformationen werden in den Strom gegeben.

```
4  4
1  1.0     2 2.0  -1 0.0
2  1.0     4 2.0   3 1.0  -1 0.0
3  1.0    -1 0.0
4  1.0     3 1.0  -1 0.0
```

Abbildung 1.9: Datei zum Graphen aus Abb. 1.4

1.4.5 Ein- und Ausgabeformat

Um das von den Ein- und Ausgabeoperatoren der Klasse `TGr` verwendete Dateiformat darzustellen, nehmen wir an, in einer Instanz dieser Klasse sei ein Digraph $D = (N, A, d)$ gespeichert mit $N = \{1, \dots, n\}$, $n \in \mathbb{N}$ und einer zusätzlichen Funktion $g : N \to \mathbb{R}_{\geqslant 0}$. Weiter gelte:

$$\operatorname{adj}_D^+(i) = \{v_{i,1}, \dots, v_{i,n_i}\}, \quad \forall i \in \{1, \dots, n\}$$

mit $n_i \in \mathbb{N} \cup \{0\}$.

Eine durch `TGr::operator`$\ll$ erzeugte Ausgabe hat dann das folgende Format:

$$\begin{array}{llllllll}
|V| & |E| & & & & & & \\
1 & g_1 & v_{1,1} & d_{\{1,v_{1,1}\}} & \dots & v_{1,n_1} & d_{\{1,v_{1,n_1}\}} & -1 \quad 0.0 \\
 & \vdots & & & & & & \\
i & g_i & v_{i,1} & d_{\{i,v_{i,1}\}} & \dots & v_{i,n_i} & d_{\{i,v_{i,n_i}\}} & -1 \quad 0.0 \\
 & \vdots & & & & & & \\
n & g_n & v_{n,1} & d_{\{n,v_{n,1}\}} & \dots & v_{n,n_n} & d_{\{n,v_{n,n_n}\}} & -1 \quad 0.0
\end{array}$$

Die Funktion `TGr::operator`$\gg$ akzeptiert umgekehrt natürlich auch genau Eingaben in dieser Form.

Als Beispiel betrachte man die in Abbildung 1.9 dargestellte, zum Graphen aus Abbildung 1.4 (siehe Seite 17) gehörende Ausgabe.

Kapitel 2

Heuristiken

In diesem Kapitel werden mehrere Ausprägungen zweier Heuristiken zur näherungsweisen Lösung von USTP'$_{G,Z}$ für einen ungerichteten Graphen $G = (V, E, d, g)$ und eine Menge $Z \subseteq V$ spezieller Knoten vorgestellt. Die beiden Heuristiken gründen in der Übertragung von Ideen, die bei der Suche nach einem minimalen aufspannenden Baum auf einem Graphen innerhalb der klassischen Algorithmen von Prim (siehe im Original [Pri57]) bzw. von Kruskal (siehe im Original [Kru56]) Anwendung finden. Da diese Algorithmen später noch gebraucht werden, hat der Autor sie in Kapitel 3 ähnlich den Anregungen aus dem Buch von Sedgewick [Sed92] implementiert. Die Suche nach einem minimalen aufspannenden Baum T, d.h. nach einem Baum, der zu minimalen Kosten $d(T) + g(T)$ alle Knoten des Graphen G aufspannt, kann als Spezialfall von USTP'$_{G,Z}$ aufgefaßt werden, indem man $Z = V$ setzt. Dieses Problem wird im weiteren MST$_G$ genannt, nach dem englischen „minimum spanning tree". Es ist mit Hilfe obengenannter Algorithmen effektiv in polynomialer Zeit zu lösen, genauso wie die Suche SP$_{G,v,w}$ („shortest paths") nach einem kürzesten Pfad in G zwischen zwei Knoten $v, w \in V$. Die Lösung von SP$_{G,v,w}$, das gerade den zu MST$_G$ „entgegengesetzten" Spezialfall von USTP'$_{G,Z}$ für $|Z| = 2$ darstellt, erfolgt durch den Algorithmus von Dijkstra oder den von Floyd-Warshall. Auch diese beiden Algorithmen werden im Rahmen von Kapitel 3 implementiert.

Der entscheidende Unterschied, der zwischen den beiden gerade angesprochenen Spezialfällen ($|Z| = |V|, |Z| = 2$) und dem Problem USTP'$_{G,Z}$ für $2 < |Z| < |V|$ besteht, ist die Eigenschaft des letzteren, NP-vollständig zu sein; es wird also wohl keinen Algorithmus geben, der es in polynomialer Zeit zu lösen imstande sein könnte. Karp hat gezeigt ([Kar72]), daß das zu USTP$_{G,Z}$ gehörende Entscheidungsproblem („Gibt es einen Teilbaum in G, der Z überdeckt und dessen Gewicht eine gegebene Schranke nicht übersteigt ?") im allgemeinen NP-schwer ist (siehe [GaJ79] auf den Seiten 208f.). Das Problem bleibt sogar NP-vollständig, falls alle Kanten des Graphen das gleiche Gewicht aufweisen, falls G bipartit ist und es in Z bzw. $V \backslash Z$ keine Paare adjazenter Knoten gibt (Berlekamp, 1976) oder falls G ein planarer Graph ist (siehe [GaJ77]).

Nach ([Won82]) ist auch DSTP$_{D_G,Z}$ NP-vollständig; USTP'$_{G,Z}$ ist ebenfalls NP-vollständig, weil es USTP$_{G,Z}$ als Spezialfall, nämlich für $g \equiv 0$, umfaßt.

Die NP-Vollständigkeit von USTP'$_{G,Z}$ lenkt den Blick auf Heuristiken und in neuerer Zeit auch auf genetische Algorithmen, mit deren Hilfe man in der Praxis unter vertretbarem Zeit- und Speicheraufwand wenigstens zulässige Näherungslösungen für die oben aufgezählten Steiner-Probleme erhalten kann.

2.1 Path Distance-Heuristiken

Im Aufsatz [WiM92] klassifizieren Winter und MacGregor-Smith die zur Lösung des Steiner-Problems vorhandenen Heuristiken. Die „Shortest Path-Heuristik" (SPH), der wir uns im nächsten Abschnitt zuwenden wollen, gehört demnach genauso wie die „Distance Network-Heuristik" (DNH, siehe Abschnitt 2.3) zur Klasse der „Path Distance-Heuristiken".

2.1.1 Algorithmus

PD-Heuristiken zur Lösung von USP'$_{G,Z}$ auf dem zusammenhängenden ungerichteten Graphen $G = (V, E, d, g)$ mit der Menge $Z \subseteq V$, für die $|Z| > 1$ gelten soll, haben den folgenden allgemeinen Aufbau:

Algorithmus 1 (Path Distance - Heuristik)

Schritt 1: Initialisierung

Setze $i = 0$ und betrachte mit $V_0 := Z$ und $E_0 := \emptyset$ den Graphen

$$T_0 := (V_0, E_0, d, g);$$

Schritt 2: Auswahl eines Pfades

$P_{i+1} = (v_{i+1,1}, \dots, v_{i+1,n_{i+1}})$ mit $n_{i+1} > 1$ sei Pfad in G, der zwei Zusammenhangskomponenten $[v_{i+1}]_{\overset{*}{\underset{T_i}{\leftrightarrow}}}$ und $[w_{i+1}]_{\overset{*}{\underset{T_i}{\leftrightarrow}}}$ mit

$$[v_{i+1}]_{\overset{*}{\underset{T_i}{\leftrightarrow}}} \cap [w_{i+1}]_{\overset{*}{\underset{T_i}{\leftrightarrow}}} = \emptyset \tag{2.1}$$

verbindet, d.h. es soll gelten:

$$v_{i+1,1} \in [v_{i+1}]_{\overset{*}{\underset{T_i}{\leftrightarrow}}} \text{ und } v_{i+1,n_{i+1}} \in [w_{i+1}]_{\overset{*}{\underset{T_i}{\leftrightarrow}}}, \tag{2.2}$$

Man setze dann

$$T_{i+1} := (V_{i+1}, E_{i+1}, d, g) \text{ mit}$$

$$V_{i+1} := V_i \cup \{\, v_{i+1,k} \mid k \in \{2, \dots, n_{i+1} - 1\} \,\},$$

$$E_{i+1} := E_i \cup \bigcup_{k=1}^{n_{i+1}-1} \{v_{i+1,k}, v_{i+1,k+1}\},$$

und schließlich noch $i := i + 1$.

Schritt 3: Terminierung

Man breche ab, falls

$$|\,\{\, [z]_{\overset{*}{\underset{T_i}{\leftrightarrow}}} \mid z \in Z \,\}\,| = 1$$

gilt. Ansonsten wiederhole man Schritt 2.

2.1.2 Eigenschaften von PD-Heuristiken

Satz 2.1.1
PDH ist für $|Z| > 1$ wohldefiniert und terminiert.

Beweis.
Wir betrachten die Graphen G und T_0 wie oben angegeben. Weil G zusammenhängt und $V_0 = Z$ mit $|Z| > 1$ gilt, existiert ein Pfad P_1, der die Bedingungen (2.1) und (2.2) erfüllt. Ist

$$\bar{z}_1 := \left|\left\{ [z]_{\underset{T_1}{\overset{*}{\leftrightarrow}}} \;\middle|\; z \in Z \right\}\right| = 1 ,$$

so terminiert der Algorithmus beim ersten Durchlauf von Schritt 3. Andernfalls ($\bar{z}_1 > 1$) gilt $\bar{z}_1 < |Z|$, und Schritt 2 wird für $i + 1 = 2$ wiederholt. Sei in Schritt 3 nun

$$\bar{z}_i := \left|\left\{ [z]_{\underset{T_i}{\overset{*}{\leftrightarrow}}} \;\middle|\; z \in Z \right\}\right| > 1$$

für ein $i \in \mathbb{N}$. Weil G zusammenhängt, existiert ein Pfad P_{i+1} mit (2.1) und (2.2). Für

$$\bar{z}_{i+1} := \left|\left\{ [z]_{\underset{T_{i+1}}{\overset{*}{\leftrightarrow}}} \;\middle|\; z \in Z \right\}\right| > 1$$

gilt die Ungleichung

$$\bar{z}_{i+1} < \bar{z}_i \; . \tag{2.3}$$

Gilt $\bar{z}_{i+1} = 1$, so terminiert Algorithmus 1. Andernfalls wird Schritt 2 für $i + 1$ wiederholt. Insgesamt ist eine PD-Heuristik wegen (2.3) und $\bar{z}_1 < |Z|$ nach höchstens $|Z| - 1$ Iterationen von Schritt 2 beendet. □

Wegen der Abbruchbedingung in Schritt 3 und der Initialisierung $V_0 = Z$ ist klar, daß eine PDH einen zusammenhängenden Graphen liefert, welcher die Knoten aus Z aufspannt.

Satz 2.1.2
PDH liefert eine zulässige Lösung für USP'$_{G,Z}$.

Wir weisen darauf hin, daß PD-Heuristiken keineswegs Bäume liefern müssen. Erweitert man PDH jedoch durch Hinzufügen zweier weiterer Bedingungen an die in Schritt 2 gewählten Pfade, so stellt man fest, daß der auf diese Weise modifizierte Algorithmus sogar Näherungen für Lösungen von USTP'$_{G,Z}$ liefert.

Man betrachte nun also Algorithmus 1 mit dem Zusatz in Schritt 2, daß P_{i+1} jeweils ein

$$\begin{array}{l}\text{einfacher Pfad sei mit}\\ v_{i+1,k} \notin \bigcup\limits_{z \in Z} [z]_{\underset{T_i}{\overset{*}{\leftrightarrow}}}, \quad \forall k \in \{2, \ldots, n_{i+1} - 1\}.\end{array} \tag{2.4}$$

Dann gilt wie oben der folgende

Satz 2.1.3
PDH mit (2.4) ist für $|Z| > 1$ wohldefiniert und terminiert.

Beim Beweis dieses Satzes beachte man, daß die Zusammenhangs-Eigenschaft von G sogar die Existenz von einfachen Pfaden P_{i+1} sichert, welche zusätzlich (2.4) erfüllen. Die weitere Argumentation verläuft exakt so wie im Beweis von Satz 2.1.1. Schließlich gilt auch hier, daß Schritt 2 genau $|Z| - 1$ -mal iteriert wird.

Man hat nun erfreulicherweise den

Satz 2.1.4
PDH mit (2.4) liefert eine zulässige Lösung für USTP'$_{G,Z}$.

Beweis.
Wir zeigen per Induktion nach i, daß jede Zusammenhangskomponente $[z]_{\overset{*}{\underset{T_i}{\leftrightarrow}}}$ für $z \in Z$ stets einen Baum auf dem Graphen T_i induziert, d.h. daß nach 1.1.9 gilt:

$$\left|\left\{ \{v,w\} \in E_i \mid v,w \in [z]_{\overset{*}{\underset{T_i}{\leftrightarrow}}} \right\}\right| = \left|[z]_{\overset{*}{\underset{T_i}{\leftrightarrow}}}\right| - 1, \forall z \in Z . \tag{2.5}$$

Für $i = 0$ ist nichts zu zeigen, da dann jede Zusammenhangskomponente $[z]_{\overset{*}{\underset{T_0}{\leftrightarrow}}}$ aus genau einem Knoten $z \in Z$ besteht.

Für $0 \leqslant i < |Z| - 1$ nehmen wir an, daß die Zusammenhangskomponenten $[z]_{\overset{*}{\underset{T_i}{\leftrightarrow}}}$ für alle $z \in Z$ auf T_i gemäß (2.5) Bäume induzieren. Wir brauchen nur die durch Schritt 2 betroffenen Zusammenhangskomponenten $[v_{i+1}]_{\overset{*}{\underset{T_i}{\leftrightarrow}}}$ und $[w_{i+1}]_{\overset{*}{\underset{T_i}{\leftrightarrow}}}$ zu betrachten. Alle anderen bleiben wegen der Zusatzbedingung (2.4) unverändert und damit nach Induktionsannahme Bäume. Der Pfad, durch den die beiden angesprochenen Zusammenhangskomponenten verschmolzen werden, hat $n_{i+1} - 1$ Kanten außerhalb der durch die Zusammenhangskomponenten induzierten Graphen, denn für $n_{i+1} - 2$ Knoten des Pfades gilt nach Gleichung (2.4), daß sie in keiner der beiden Zusammenhangskomponenten liegen.
Deshalb gilt für die entstehende Komponente:

$$\begin{aligned}
\left|\left\{ \{x,y\} \in E_{i+1} \mid x,y \in [v_{i+1}]_{\overset{*}{\underset{T_{i+1}}{\leftrightarrow}}} \right\}\right| = & \\
= & \left|\left\{ \{x,y\} \in E_i \mid x,y \in [v_{i+1}]_{\overset{*}{\underset{T_i}{\leftrightarrow}}} \right\}\right| + \\
+ & \left|\left\{ \{x,y\} \in E_i \mid x,y \in [w_{i+1}]_{\overset{*}{\underset{T_i}{\leftrightarrow}}} \right\}\right| + n_{i+1} - 1 = \\
\underset{\text{Ind.-Ann.}}{=} & \left|[v_{i+1}]_{\overset{*}{\underset{T_i}{\leftrightarrow}}}\right| - 1 + \left|[w_{i+1}]_{\overset{*}{\underset{T_i}{\leftrightarrow}}}\right| - 1 + n_{i+1} - 1 = \\
= & \left|[v_{i+1}]_{\overset{*}{\underset{T_i}{\leftrightarrow}}}\right| + \left|[w_{i+1}]_{\overset{*}{\underset{T_i}{\leftrightarrow}}}\right| + n_{i+1} - 2 - 1 = \\
\underset{(2.4)}{=} & \left|[v_{i+1}]_{\overset{*}{\underset{T_{i+1}}{\leftrightarrow}}}\right| - 1
\end{aligned}$$

Damit ist die Behauptung (2.5) auch für $i + 1$ gezeigt. □

2.1.3 Zwei weitere Schritte für PDH

Im letzten Unterabschnitt wurde Bedingung (2.4) in den Algorithmus 1 eingefügt, um zulässige Lösungen für USTP'$_{G,Z}$ statt lediglich für USP'$_{G,Z}$ zu erhalten. Den Schlüssel zu einer alternativen Vorgehensweise liefern Rayward-Smith und Clare ([RaC86]) oder auch Winter und MacGregor-Smith ([WiM92]), wenn sie Verfahren zur Lösung von USTP'$_{G,Z}$ als Algorithmen verstehen, welche die Steiner-Knoten $S' \subseteq S$ liefern, die in einer Lösung von USTP'$_{G,Z}$ enthalten sind. In Kenntnis dieser Knoten S' erhält man die Kanten der Lösung von USTP'$_{G,Z}$, indem man einen kleinsten aufspannenden Baum auf dem von $S' \cup Z$ in G induzierten Graphen berechnet.

Definition 2.1.5
Sei $G = (V, E, d, g)$ ein ungerichteter Graph. Dann heißt $G'_{G,V'}$ der durch $V' \subseteq V$ induzierte Teilgraph auf G, falls gilt:

$$\begin{aligned} G'_{G,V'} &:= (V', E', d, g) \text{ mit} \\ E' &:= \{\, \{v, w\} \in E \mid v, w \in V' \,\} \,. \end{aligned}$$

PD-Heuristiken sind in der Lage, eine Knotenmenge $S'' \subseteq S$ zu liefern, sodaß in $G'_{G,S''\cup Z}$ ein die Knotenmenge Z aufspannender Baum existiert, denn das Ergebnis von PDH ist ja gerade ein zusammenhängender Graph, der Z aufspannt. Man erhält nun eine Näherungslösung für USTP'$_{G,Z}$ durch Anwendung eines MST-Algorithmus (also eines der klassischen Verfahren von Prim oder Kruskal) auf den induzierten Teilgraphen $G'_{G,S''\cup Z}$. Ein weiterer Schritt, nämlich das Abschneiden unnötiger Blätter von diesem kleinsten aufspannenden Baum, vermag die Näherungslösung möglicherweise noch zu verbessern.

Statt also bei der Pfadauswahl die beiden im letzten Unterabschnitt aufgeführten zusätzlichen Bedingungen einzuhalten, kann man die von einer PD-Heuristik gelieferte approximative Lösung von USP'$_{G,Z}$ durch Anwendung der beiden folgenden Schritte in eine zulässige Lösung von USTP'$_{G,Z}$ transformieren; wir nennen den dann entstehenden (gesamten) Algorithmus PDH':

Algorithmus 2 (PDH': Zwei weitere Schritte für PDH)

Schritt 4: Lösung eines MST-Problems

Es sei

$$T_{|Z|-1} := (V_{|Z|-1}, E_{|Z|-1}, d, g)$$

der durch Algorithmus 1 gelieferte zusammenhängende Graph, der die Knoten in $V_{|Z|-1} = Z \cup S''$ aufspannt.
$T_{|Z|} := (V_{|Z|}, E_{|Z|}, d, g)$ sei Lösung des Minimum Spanning Tree - Problems $\mathrm{MST}_{G'_{G,V_{|Z|-1}}}$. Weiter sei $i := |Z|$.

Schritt 5: Abschneiden unnötiger Blätter

Falls folgende Gleichung gilt, stoppe man den Algorithmus:

$$\{\, v \in V_i \backslash Z \mid \gamma_{T_i}(v) = 1 \,\} = \emptyset$$

Andernfalls wähle ein

$$v_i \in \{\, v \in V_i \backslash Z \mid \gamma_{T_i}(v) = 1 \,\}$$

und definiere

$$\begin{aligned} T_{i+1} &:= (V_{i+1}, E_{i+1}, d, g) \text{ mit} \\ V_{i+1} &:= V_i \setminus \{v_i\} \ , \\ E_{i+1} &:= E_i \setminus \{\, \{v_i, w\} \mid w \in \mathrm{adj}_{T_i}(v_i) \,\} \ . \end{aligned}$$

Setze $i := i + 1$ und wiederhole Schritt 5.

$T_{|Z|-1}$ ist gemäß Satz 2.1.2 zulässig für USP'$_{G,Z}$. Schritt 4 liefert einen aufspannenden Baum $T_{|Z|}$ in G für die Knoten in $Z \cup S''$, der deshalb zulässig ist für USTP'$_{G,Z}$.

Da Schritt 5 nur Knoten $v \in S$ vom Grad 1 betrifft, liefert dieser Schritt in jeder Iteration wieder eine zulässige Lösung für USTP'$_{G,Z}$. Sei $i \geqslant |Z|$ und gelte

$$\{\, v \in V_i \setminus Z \mid \gamma_{T_i}(v) = 1 \,\} \neq \emptyset \ .$$

Dann ist $|V_{i+1}| = |V_i| - 1$, und die Iteration von Schritt 5 muß nach spätestens $V_{|Z|} - |Z|$ Schritten enden, da jeder für USTP'$_{G,Z}$ zulässige Baum mindestens $|Z|$ Knoten besitzt.

Insgesamt gelten also auch für PDH' die in Unterabschnitt 2.1.2 für PDH mit (2.4) aufgestellten Sätze:

Satz 2.1.6
PDH' ist für $|Z| > 1$ *wohldefiniert und terminiert.*

Satz 2.1.7
PDH' liefert eine zulässige Lösung für USTP'$_{G,Z}$.

Die Komplexität von Schritt 4 für dünnbesetzte Graphen beträgt je nachdem, ob der Algorithmus von Prim oder der von Kruskal (siehe dazu Kapitel 3) verwendet wird, $O\big((|E| + |V|)\log|V|\big)$ bzw. $O\big(|E|\log|E|\big)$. Die Komplexität $O(V)$ des fünften Schrittes wird von der des Schrittes davor dominiert.

Bemerkung 2.1.8
Auch wenn schon eine zulässige Lösung für USTP'$_{G,Z}$ vorliegt und vom Gesichtspunkt der Zulässigkeit eigentlich die Durchführung der obigen Schritte nicht notwendig erscheint, kann diese wegen des Abschneidens unnötiger Blätter in Schritt 5 (engl. „pruning“ bzw. „trimming“, siehe [RaC86] bzw. [WiM92]) durchaus noch eine Verbesserung der Lösung bringen. Man beachte in diesem Zusammenhang die in Kapitel 5 vorgerechneten Beispiele.

2.2 Shortest Path - Heuristik

Die SP-Heuristik wurde (freilich noch nicht unter dieser Bezeichnung) ursprünglich von Takahashi und Matsuyama in [TaM80] veröffentlicht. Sie hat die allgemeine Form von Algorithmus 1.

2.2.1 Algorithmus SPH-z

Wir spezifizieren nun Schritt 2 für die Shortest Path Heuristic. Dabei nehmen wir an, daß ein zusammenhängender, ungerichteter Graph

$$G = (V, E, d, g) \text{ mit } d(e) > 0, \quad \forall e \in E$$

und ein spezieller Knoten $z' \in Z \subseteq V$ (mit $|Z| > 1$) vorgegeben sind.

Algorithmus 3 (Shortest Path - Heuristik: SPH-z)

Schritt 1: Initialisierung

Setze $i = 0$ und betrachte mit $V_0 := Z$ und $E_0 := \emptyset$ den Graphen

$$T_0 := (V_0, E_0, d, g) \;.$$

Schritt 2: Auswahl eines Pfades

$P_{i+1} = (v_{i+1,1}, \ldots, v_{i+1,n_{i+1}})$ mit $n_{i+1} > 1$ sei Pfad in G, der einen Knoten $v_{i+1,1}$ mit der Zusammenhangskomponente $[z']_{\overset{*}{\underset{T_i}{\leftrightarrow}}}$ verbindet. Es gelte:

$$v_{i+1,1} \in Z \backslash [z']_{\overset{*}{\underset{T_i}{\leftrightarrow}}} \text{ und } v_{i+1,n_{i+1}} \in [z']_{\overset{*}{\underset{T_i}{\leftrightarrow}}} \tag{2.6}$$

$$d(P_{i+1}) + g(P_{i+1}) = \min\{\, d_{\min(G)}(z, [z']_{\overset{*}{\underset{T_i}{\leftrightarrow}}}) \mid z \in Z \backslash [z']_{\overset{*}{\underset{T_i}{\leftrightarrow}}} \} \tag{2.7}$$

Man setze

$$T_{i+1} := (V_{i+1}, E_{i+1}, d, g) \text{ mit}$$

$$V_{i+1} := V_i \cup \{\, v_{i+1,k} \mid k \in \{2, \ldots, n_{i+1} - 1\} \,\} \;,$$

$$E_{i+1} := E_i \cup \bigcup_{k=1}^{n_{i+1}-1} \{v_{i+1,k}, v_{i+1,k+1}\} \;,$$

und schließlich noch $i := i + 1$.

Schritt 3: Terminierung

Man breche ab, falls

$$\left| \{\, [z]_{\overset{*}{\underset{T_i}{\leftrightarrow}}} \mid z \in Z \,\} \right| = 1$$

gilt. Ansonsten wiederhole man Schritt 2.

Bemerkung 2.2.1

Aus den Forderungen in (2.6) an die Endpunkte des Pfades P_i folgt mit Bedingung (2.7) insbesondere, daß in Schritt 2 gilt:

$$d(P_{i+1}) + g(P_{i+1}) = d_{\min(G)}(v_{i+1,1}, v_{i+1,n_{i+1}}) \;. \tag{2.8}$$

2.2.2 Wohldefiniertheit

Seien $i \in \mathbb{N} \cup \{0\}$ und

$$Z \backslash [z']_{\overset{*}{\underset{T_i}{\leftrightarrow}}} \neq \emptyset.$$

Da G zusammenhängt, existiert ein Pfad P_{i+1} wie in Algorithmus 3 gefordert. Dann ist $v_{i+1,1}$ in Schritt 2 wegen Gleichung (2.7) derjenige spezielle Knoten, welcher der Zusammenhangskomponente des (festen) Knotens z' am nächsten liegt; deshalb heißt die Heuristik „Shortest Path Heuristic", kurz „SPH-z". In der Notation von Algorithmus 1 schreibt man:

$$v_{i+1} = v_{i+1,1} \text{ und } w_{i+1} = z'. \tag{2.9}$$

Die Bedingungen (2.1) und (2.2) ergeben sich damit sofort aus der Wahl von $v_{i+1,1}$ in der SP-Heuristik gemäß (2.6). Wegen der Minimalitätsbedingung (2.8) und der Positivität von d muß P_{i+1} ein einfacher Pfad sein. Für die speziellen Knoten $z \in Z \backslash [z']_{\underset{T_i}{\overset{*}{\leftrightarrow}}}$ gilt wegen der Wahl von $V_0 := Z$, daß

$$[z]_{\underset{T_i}{\overset{*}{\leftrightarrow}}} = \{z\}.$$

Die nicht-Endknoten im gewählten Pfad P_{i+1} erfüllen die Bedingung (2.4). Wir nehmen im Gegenteil an, (2.4) würde nicht gelten, d.h. es existiert ein $k_0 \in \{2, \ldots, n_{i+1} - 1\}$, sodaß $v_{i+1,k_0} \in [z]_{\underset{T_i}{\overset{*}{\leftrightarrow}}}$ für ein $z \in Z$. Man unterscheidet zwei Fälle; ist $v_{i+1,k_0} \in [z']_{\underset{T_i}{\overset{*}{\leftrightarrow}}}$, so würde für

$$P' := (v_{i+1,1}, \ldots, v_{i+1,k_0})$$

neben (2.6) wegen der Positivität von d im Widerspruch zu (2.7) auch

$$d(P') + g(P') < d(P_{i+1}) + g(P_{i+1})$$

gelten. Ist andererseits $v_{i+1,k_0} = z$ mit einem $z \in Z \backslash [z']_{\underset{T_i}{\overset{*}{\leftrightarrow}}}$, so ergäbe sich obiger Widerspruch für den Pfad

$$P' := (v_{i+1,k_0}, \ldots, v_{i+1,n_{i+1}}) \ .$$

Ingesamt hat man nun gezeigt, daß die Pfadauswahl in SPH-z den Bedingungen einer Path Distance-Heuristik (mit Zusatzbedingungen) genügt. Deshalb lassen sich wie in Abschnitt 2.1.2 die folgenden beiden Sätze auch für SPH-z aufschreiben.

Satz 2.2.2
SPH-z ist wohldefiniert und terminiert, falls $|Z| > 1$.

Satz 2.2.3
SPH-z liefert eine zulässige Lösung für USTP'$_{G,Z}$.

2.2.3 Komplexität

In SPH-z werden für jeden speziellen Knoten Informationen über die kürzesten Pfade zu allen anderen Knoten im Graphen benötigt. Wir nehmen an, daß man diese kürzesten Pfade mit Hilfe des Dijkstra-Algorithmus für dünnbesetzte Graphen berechnet, wie er in Kapitel 3 mit einer Komplexität von $O\big((|V|+|E|)\log|V|\big)$ implementiert wird. Die Pfadberechnung, die man

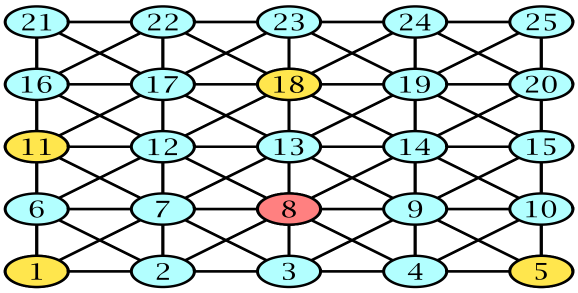

Abbildung 2.1: Gitter zur Demostration von SPH-z

allerdings - insbesondere, wenn man das Steiner-Problem mehrmals auf nicht allzu großen Graphen zu lösen hat - offline realisieren kann, kommt also auf eine Komplexität von $O(|Z|(|V| + |E|) \log |V|)$.

Alternativ dazu kann man auch–soweit es die Anzahl $|V|$ von Knoten in G erlaubt–den Algorithmus von Floyd-Warshall verwenden, um die Pfadinformationen in einer Zeit von $O(|V|^3)$ zu erhalten.

In den später implementierten Versionen von SPH-z werden die Pfade vor der Anwendung von SPH-z berechnet und gespeichert, um während Schritt 2 des Algorithmus nur noch ausgelesen zu werden; deshalb kann man die Komplexität von SPH-z getrennt betrachten. Damit sich am Ende des Algorithmus alle $z \in Z$ in einer einzigen Zusammenhangskomponente befinden, werden schlimmstenfalls $O(|V|)$ Kanten eingefügt. Für alle eingefügten Kanten und die verbliebenen isolierten speziellen Knoten muß ggf. die Distanz zur Zusammenhangskomponente von z' korrigiert werden; das geschieht für eine eingefügte Kante durch Operationen auf einer Priority Queue (siehe Kapitel 3), die die speziellen Knoten verwaltet, in $O(|Z| \log |Z|)$, und man hat schließlich $O(|V||Z| \log |Z|)$ als worst case-Komplexität von SPH-z ohne die Pfadberechnungen.

Insgesamt dominiert wegen $\log |Z| \leqslant \log |V|$ die $|Z|$-malige Anwendung des Dijkstra-Algorithmus, was zu einer Gesamt-Komplexität der SP-Heuristik von $O(|Z|(|V| + |E|) \log |V|)$ führt. Im Fall, daß die Pfadinformationen mit Hilfe des Algorithmus von Floyd-Warshall erlangt werden, ist der beherrschende Term $O(|V|^3)$.

2.2.4 Beispiel

Wir demonstrieren die Funktionweise von SPH-z an dem in Abbildung 2.1 dargestellten ungerichteten Graphen $G = (V, E, d, g)$ mit $g \equiv 0$. Die Gewichte der waagerechten und senkrechten Kanten von G betragen jeweils 1.0, die der diagonalen 1.4. Es liegt also eine (annähernd) euklidische Bewertung vor. Die speziellen Knoten des Problems sind in der Abbildung orange bzw. rot markiert; es ist $Z = \{1, 5, 8, 11, 18\}$. Wir starten SPH-z mit dem „Wurzelknoten“ $z' = 18$. Die rote Farbe kennzeichnet Knoten $v \in Z$

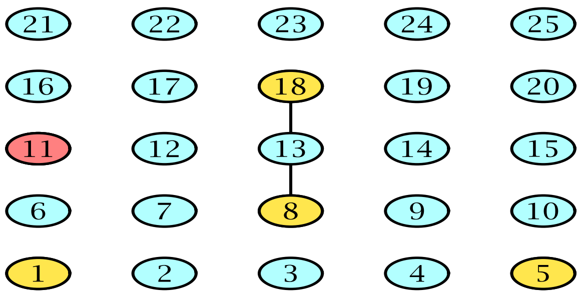

Abbildung 2.2: Nach der ersten Iteration von SPH-z: (V, E_1, d, g)

mit der Eigenschaft

$$d_{\min(G)}(v, [z']_{\overset{*}{\underset{T_i}{\leftrightarrow}}}) = \min\{\, d_{\min(G)}(z, [z']_{\overset{*}{\underset{T_i}{\leftrightarrow}}}) \mid z \in Z \setminus [z']_{\overset{*}{\underset{T_i}{\leftrightarrow}}} \,\}.$$

Im dargestellten Beispiel erfüllt immer genau ein Knoten diese Eigenschaft. Ein solcher Knoten wird in der nächsten Iteration wegen Gleichung (2.7) an die bestehende Teillösung angehängt. Es sei darauf hingewiesen, daß die Abbildungen stets die in G eingebetteten Graphen (V, E_i, d, g) darstellen.
Für $i = 0$ gilt:

$$d_{\min(G)}(1, [18]_{\overset{*}{\underset{T_0}{\leftrightarrow}}}) = 3.8 \qquad d_{\min(G)}(5, [18]_{\overset{*}{\underset{T_0}{\leftrightarrow}}}) = 3.8$$
$$d_{\min(G)}(8, [18]_{\overset{*}{\underset{T_0}{\leftrightarrow}}}) = 2.0 \qquad d_{\min(G)}(11, [18]_{\overset{*}{\underset{T_0}{\leftrightarrow}}}) = 2.4$$

Um T_1 zu erhalten, werden deshalb die Kanten $\{8, 13\}$ und $\{13, 18\}$ an T_0 angehängt.
Damit gilt für die verbliebenen isolierten speziellen Knoten in T_1:

$$d_{\min(G)}(1, [18]_{\overset{*}{\underset{T_1}{\leftrightarrow}}}) = 2.4 \qquad d_{\min(G)}(5, [18]_{\overset{*}{\underset{T_1}{\leftrightarrow}}}) = 2.4$$
$$d_{\min(G)}(11, [18]_{\overset{*}{\underset{T_1}{\leftrightarrow}}}) = 2.0$$

Man beachte, daß für jeden eingefügten Knoten v und jeden der verbliebenen isolierten speziellen Knoten z getestet werden muß, ob sich der Abstand $d_{\min(G)}(z, [18]_{\overset{*}{\underset{T_0}{\leftrightarrow}}})$ durch Einfügen von v ändert; gegebenenfalls muß $d_{\min(G)}(z, [18]_{\overset{*}{\underset{T_1}{\leftrightarrow}}})$ entsprechend aktualisiert werden.

Jetzt kommt also Knoten 11 an die Reihe, und man erhält T_2 durch Einfügen der Kanten $\{11, 12\}$ und $\{12, 13\}$ in T_1. Nach der zweiten Iteration von SPH-z gilt:

$$d_{\min(G)}(1, [18]_{\overset{*}{\underset{T_2}{\leftrightarrow}}}) = 2.0 \qquad d_{\min(G)}(5, [18]_{\overset{*}{\underset{T_2}{\leftrightarrow}}}) = 2.4$$

Es geht damit weiter, daß $\{1, 6\}$ und $\{6, 11\}$ hinzugefügt werden.

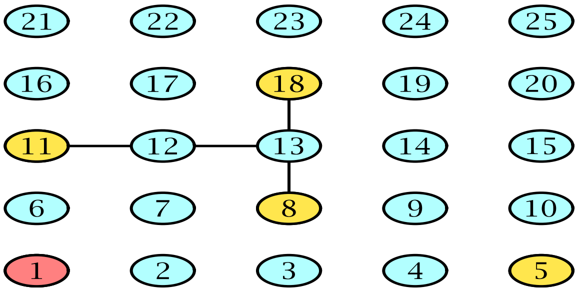

Abbildung 2.3: Nach der zweiten Iteration von SPH-z: (V, E_2, d, g)

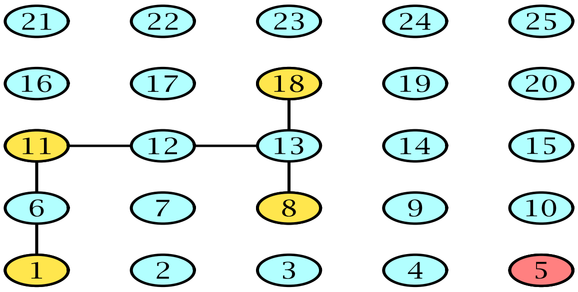

Abbildung 2.4: Nach der dritten Iteration von SPH-z: (V, E_3, d, g)

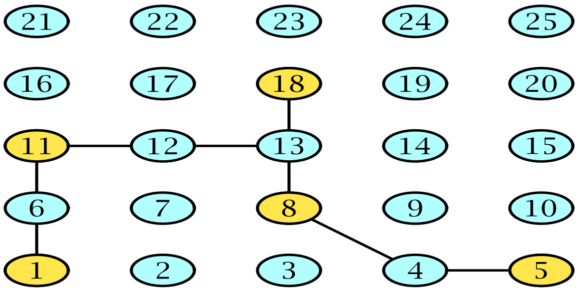

Abbildung 2.5: Endergebnis von SPH-z: (V, E_4, d, g)

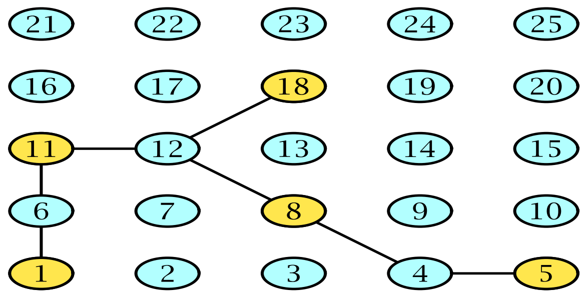

Abbildung 2.6: Lösung mit SPH-V (erzielt im Schritt mit Startknoten 12)

Schließlich bleibt nur noch Knoten 5 übrig. Man verbindet ihn über die Kanten $\{5,4\}$ und $\{4,8\}$ mit T_3. Das Endergebnis T_4 ist in Abbildung 2.5 zu sehen. Der entstandene Baum hat ein Gewicht von

$$d(T_4) + g(T_4) = d(T_4) = 8.4 \,.$$

2.2.5 Iterierte Formen der SP-Heuristik

Die SP-Heuristik in der oben dargestellten Form ist von der Wahl eines „Wurzelknotens“ z' (im Beispiel war das Knoten 18) abhängig. Weil bei beliebigen Graphen die beste Wahl eines solchen Startknotens von vornherein nicht zu sehen ist, versucht man, diese Abhängigkeit zu beheben, indem man SPH-z einmal für jeden Knoten $z \in Z$ ausführt und dann das beste während dieser Iteration erhaltene Ergebnis als Näherungslösung nimmt. Man nennt die entstehende Heuristik SPH-Z. Sie hat, vorher berechnete Pfade für die Knoten in Z vorausgesetzt (siehe die Bemerkungen in der Komplexitätsbetrachtung von SPH-z), eine Komplexität von $O(|V||Z|^2 \log |Z|)$ für den schlimmsten Fall. Hier erweist sich die Zwischenspeicherung der einmal errechneten Pfadinformationen als wirkungsvoll, denn man kann diese einmal berechnen und sie dann in $|Z|$ Wiederholungen von SPH-z für die verschiedenen Startknoten immer wieder benutzen.

Statt wie in SPH-Z nur Knoten aus Z als Startknoten in Betracht zu ziehen, kann man SPH-z auch für jedes $v \in V$ als Startknoten ausführen. Im Falle eines $v \in V \backslash Z$ erhält man dann zwar sogar eine zulässige Lösung für USTP'$_{G,Z\cup\{v\}}$, doch ist ein Baum, welcher $Z \cup \{v\}$ aufspannt, natürlich insbesondere zulässig für USTP'$_{G,Z}$. Diese Version von SPH-z hat ihre Motivation darin, daß es in G Knoten geben kann, welche nicht auf einem der innerhalb von SPH-z betrachteten kürzesten Pfade liegen, die aber trotzdem bei der Konstruktion eines Steiner-Baums nützlich sein können. SPH-V hat eine worst case-Komplexität von $O(|V|^2|Z| \log |Z|)$ (ohne Pfadberechnungen).

Wendet man SPH-V etwa an, um das weiter vorn vorgestellte Beispiel zu lösen, dann liefert diese Methode bei $Z = \{1,5,8,11,18\}$ im Iterationsschritt mit dem Startknoten 12 als Ergebnis den in Abbildung 2.6 dargestellten Baum, welcher für USTP'$_{G,Z\cup\{12\}}$ und somit auch für USTP'$_{G,Z}$

zulässig ist aber nur ein Gewicht von 8.2 (statt 8.4 oben) besitzt.

Bezeichnet man mit res_h das Gewicht des von Algorithmus h gelieferten, für USTP'$_{G,Z}$ zulässigen Baums, so gilt

$$\text{res}_\text{SPH-Z} \leqslant \text{res}_\text{SPH-z} \quad \text{und}$$
$$\text{res}_\text{SPH-V} \leqslant \text{res}_\text{SPH-z},$$

da die beiden Algorithmen SPH-Z und SPH-V ja gerade die wiederholte Anwendung von SPH für jeden Startknoten $z \in Z$ bzw. $v \in V$ darstellen.

Man kann im Bemühen, SPH-z zu verbessern, auch einen anderen Ansatz verfolgen, der wohl von Winter und MacGregor-Smith selbst stammt (nach einer Bemerkung in ihrem Artikel [WiM92]). In SPH-z wird immer ein der Zusammenhangskomponente des Startknotens nächstliegender spezieller Knoten mit dem kürzestmöglichen Pfad an diese angeschlossen. Es kann aber von Vorteil sein, im ersten Schritt des Algorithmus nicht den dem Startknoten nächstliegenden speziellen Knoten, sondern einen vorgegebenen anderen Knoten $z'' \in Z$ dafür zu verwenden. Den entstehenden abgewandelten Algorithmus wollen wir SPH-zz nennen.

Es seien wie für SPH-z (siehe (2.2.1)) ein zusammenhängender, ungerichteter Graph

$$G = (V, E, d, g) \text{ mit } d(e) > 0, \quad \forall e \in E, \tag{2.10}$$

diesmal aber zwei spezielle Knoten $z', z'' \in Z \subseteq V$ mit $z' \neq z''$ ($\Rightarrow |Z| > 1$) vorgegeben.

Algorithmus 4 (Abgewandelte Shortest Path Heuristic SPH-zz)

Schritt 1: Initialisierung, Einfügen des ersten Pfades

$P_1 = (v_{1,1}, \ldots, v_{1,n_1})$ sei Pfad in G, für den gelte:

$$v_{1,1} = z' \text{ und } v_{1,n_1} = z'', \tag{2.11}$$
$$d(P_1) + g(P_1) = d_{\min(G)}(z', z'') \,. \tag{2.12}$$

Man setze $i := 1$ und

$$T_1 := (V_1, E_1, d, g) \text{ mit}$$

$$V_1 := Z \cup \{ v_{1,k} \mid k \in \{2, \ldots, n_1 - 1\} \}$$
$$E_1 := \bigcup_{k=1}^{n_1 - 1} \{v_{1,k}, v_{1,k+1}\} \,.$$

Schritt 2: Terminierung

Man breche ab, falls

$$|\{ [z]_{\overset{*}{\underset{T_i}{\longleftrightarrow}}} \mid z \in Z \}| = 1$$

gilt. Ansonsten gehe man zu Schritt 3.

Schritt 3: Auswahl eines Pfades

$P_{i+1} = (v_{i+1,1}, \ldots, v_{i+1,n_{i+1}})$ sei Pfad in G, der einen Knoten $v_{i+1,1}$ mit der Zusammenhangskomponente $[z']_{\overset{*}{\underset{T_i}{\leftrightarrow}}}$ verbindet. Es gelte:

$$v_{i+1,1} \in Z \backslash [z']_{\overset{*}{\underset{T_i}{\leftrightarrow}}} \text{ und } v_{i+1,n_{i+1}} \in [z']_{\overset{*}{\underset{T_i}{\leftrightarrow}}}, \tag{2.13}$$

$$d(P_{i+1}) + g(P_{i+1}) = \min\{\, d_{\min(G)}(z, [z']_{\overset{*}{\underset{T_i}{\leftrightarrow}}}) \mid z \in Z \backslash [z']_{\overset{*}{\underset{T_i}{\leftrightarrow}}} \} \tag{2.14}$$

Man setze

$$T_{i+1} := (V_{i+1}, E_{i+1}, d, g) \text{ mit}$$

$$V_{i+1} := V_i \cup \{\, v_{i+1,k} \mid k \in \{2, \ldots, n_{i+1} - 1\} \}$$

$$E_{i+1} := E_i \cup \bigcup_{k=1}^{n_{i+1}-1} \{v_{i+1,k}, v_{i+1,k+1}\}$$

und schließlich noch $i = i + 1$.
Man wiederhole Schritt 2.

SPH-zz ist zwar eine PDH, doch erfüllt diese Heuristik nicht die Zusatzbedingungen für die Pfadauswahl aus Unterabschnitt 2.1.2, denn P_1 kann die Bedingung (2.4) verletzen. Dennoch können wir einige uns schon von PDH und SPH-z bekannte Argumente anwenden und erhalten zunächst:

Satz 2.2.4
SPH-zz ist wohldefiniert und terminiert, falls $|Z| > 1$.

Beweis.
$\bar{z}_i$ sei definiert wie in Unterabschnitt 2.1.2. Ist $\bar{z}_1 = 1$, so enthält P_1 schon alle speziellen Knoten, und SPH-zz terminiert beim ersten Durchlauf von Schritt 2. In jedem Fall ist wegen der Gleichungen in (2.11) und $z' \neq z''$ die Ungleichung $\bar{z}_1 < |Z|$ erfüllt. Ist in Schritt 2 $\bar{z}_i > 1$ für ein $i \in \mathbb{N}$, so wird Schritt 3 mit einem Pfad P_{i+1} ausgeführt, der wegen der Zusammenhangs-Eigenschaft des Graphen G existiert.

Man beobachte, daß Schritt 3 in SPH-zz dem Schritt 2 in SPH-z gleicht. Wegen Bedingung (2.13) gilt $\bar{z}_{i+1} \leqslant \bar{z}_i - 1$. Nach (2.14) und derselben Argumentation wie in Unterabschnitt 2.2.2 erhält man, daß (2.4) Gültigkeit für P_{i+1} besitzt. Deshalb gilt sogar $\bar{z}_{i+1} \geqslant \bar{z}_i - 1$ und daher $\bar{z}_{i+1} = \bar{z}_i - 1$.
SPH-zz terminiert also nach $|Z| - \bar{z}_1 - 1$ Iterationen von Schritt 3. □

Satz 2.2.5
SPH-zz liefert eine zulässige Lösung für USTP'$_{G,Z}$.

Beweis.
Wir zeigen per Induktion nach i, daß jede Zusammenhangskomponente $[z]_{\overset{*}{\underset{T_i}{\leftrightarrow}}}$ für $z \in Z$ auf T_i einen Baum induziert. Dies ist zunächst sowohl für $[z']_{\overset{*}{\underset{T_1}{\leftrightarrow}}}$ wegen (2.12) der Fall, als auch für alle anderen Zusammenhangskomponenten wegen

$$z \in Z \backslash [z']_{\overset{*}{\underset{T_1}{\leftrightarrow}}} \Rightarrow [z]_{\overset{*}{\underset{T_1}{\leftrightarrow}}} = \{z\}$$

nach Wahl von V_1.
Sei nun $\bar{z}_i > 1$ für ein $i \in \mathbb{N}$; alle $[z]_{\underset{T_i}{\overset{*}{\longleftrightarrow}}}$ $(z \in Z)$ sollen Bäume auf T_i induzieren. Wie im Beweis zum vorigen Satz angeführt gilt wegen (2.14) auch die Bedingung (2.4) für P_{i+1}. Es werden also auch hier nur die Zusammenhangskomponenten $[v_{i+1,1}]_{\underset{T_i}{\overset{*}{\longleftrightarrow}}}$ und $[z']_{\underset{T_i}{\overset{*}{\longleftrightarrow}}}$ verändert. Alle anderen bleiben unverändert und nach Induktionsannahme Bäume. Der Pfad, durch den die beiden angesprochenen Zusammenhangskomponenten verschmolzen werden, hat $n_{i+1} - 1$ Kanten außerhalb der von den Zusammenhangskomponenten induzierten Graphen, weil für $n_{i+1} - 2$ Knoten des Pfades nach Gleichung (2.4) gilt, daß sie in keiner der beiden Zusammenhangskomponenten liegen.
Damit gilt für die entstehende Zusammenhangskomponente:

$$
\begin{aligned}
&\Big|\big\{ \{x,y\} \in E_{i+1} \;\big|\; x,y \in [z']_{\underset{T_{i+1}}{\overset{*}{\longleftrightarrow}}} \big\}\Big| = \\
&\qquad = \;\Big|\big\{ \{x,y\} \in E_i \;\big|\; x,y \in [z']_{\underset{T_i}{\overset{*}{\longleftrightarrow}}} \big\}\Big| + \\
&\qquad\quad + \;\Big|\big\{ \{x,y\} \in E_i \;\big|\; x,y \in [v_{i+1,1}]_{\underset{T_i}{\overset{*}{\longleftrightarrow}}} \big\}\Big| + n_{i+1} - 1 = \\
&\qquad \underset{\text{Ind.-Ann.}}{=} \Big|[z']_{\underset{T_i}{\overset{*}{\longleftrightarrow}}}\Big| - 1 + \Big|[v_{i+1,1}]_{\underset{T_i}{\overset{*}{\longleftrightarrow}}}\Big| - 1 + n_{i+1} - 1 = \\
&\qquad = \;\Big|[z']_{\underset{T_i}{\overset{*}{\longleftrightarrow}}}\Big| + \Big|[v_{i+1,1}]_{\underset{T_i}{\overset{*}{\longleftrightarrow}}}\Big| + n_{i+1} - 2 - 1 = \\
&\qquad \underset{(2.14)}{=} \Big|[z']_{\underset{T_{i+1}}{\overset{*}{\longleftrightarrow}}}\Big| - 1
\end{aligned}
$$

Schließlich ist dann auch $T_{|Z|-\bar{z}_1-1}$ ein Baum, für den wegen Satz 2.2.4 dann $\bar{z}_{|Z|-\bar{z}_1-1} = 1$ gilt. □

Um nun wieder wie bei SPH-z die Abhängigkeit des Algorithmus SPH-zz von der Wahl der beiden Anfangsknoten $z', z'' \in Z$ zu eliminieren, iteriert man SPH-zz entweder bei festem z' über alle $z'' \in Z\backslash\{z'\}$ – man erhält das Verfahren SPH-zZ mit einer worst-case-Komplexität von $O(|V||Z|^2 \log |Z|)$ – oder man iteriert sogar über alle $|Z|(|Z|-1)/2$ möglichen Kombinationen von $z', z'' \in Z$ mit $z' \neq z''$ und erhält dann das Verfahren SPH-ZZ mit einer worst-case-Komplexität von $O(|V||Z|^3 \log |Z|)$. Die Komplexitäten sind wieder ohne Berücksichtigung der einmal zu Anfang durchzuführenden Pfadberechnungen angegeben.

Da innerhalb beider Iterationsverfahren insbesondere auch der von SPH-z (bzw. SPH-Z) verwendete Pfad P_1 getestet wird, hat man die Abschätzungen

$$
\begin{aligned}
&\mathrm{res}_{\text{SPH-zZ}} \leqslant \mathrm{res}_{\text{SPH-z}} \text{ (bei gleichem Startknoten } z'\text{) und} \\
&\mathrm{res}_{\text{SPH-ZZ}} \leqslant \mathrm{res}_{\text{SPH-zZ}} \;.
\end{aligned}
$$

2.2.6 Erweiterung um die PDH-Schritte 4 und 5

Dieser Abschnitt beschäftigte sich bisher in erster Linie mit den Heuristiken SPH-z und SPH-zz, die formal den ersten drei Schritten von PDH entspre-

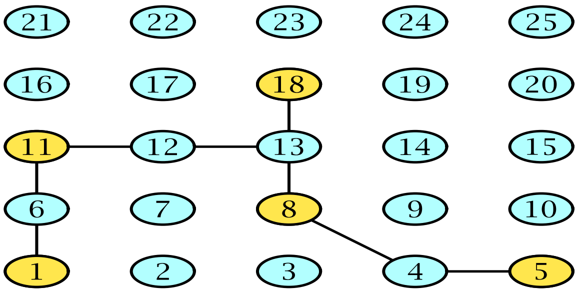

Abbildung 2.7: Lösung mit SPH-zZ, Startknoten 18

	SPH-	SPH'(Kruskal)	SPH'(Prim)
z	8.4	8.4	8.4
Z	8.4	8.4	8.4
V	8.2	8.2	8.2
zZ	8.4	8.2	8.2
ZZ	8.2	8.2	8.2

Tabelle 2.1: Alle Ergebnisse für das Beispiel aus 2.2.4

chen. Man kann diese beiden aber genauso wie PDH um die Schritte 4 und 5 (siehe Algorithmus 2) erweitern. Wir bezeichnen die dadurch entstehende Verfahren mit SPH'-z und SPH'-zz, die davon abgeleiteten iterierten Formen mit SPH'-Z, SPH'-V etc.

Die aufgrund der Ausführung der Schritte 4 und 5 möglichen Verbesserungen werden besonders bei größeren Problemen zu Tage treten, wie sie im letzten Kapitel dieser Arbeit bearbeitet werden. Aber auch schon im Beispiel aus Unterabschnitt 2.2.4 verbessert sich das Ergebnis von SPH-zZ, das in Abbildung 2.7 zu sehen ist (Gewicht: 8.4), auf den Baum mit Gewicht 8.2, den auch SPH-V liefert (siehe Abbildung 2.6). In Tabelle 2.1 sind noch einmal die Ergebnisse aller Variationen von SPH für das Beispiel aus Unterabschnitt 2.2.4 zusammengefaßt. Die von Startknoten abhängigen Algorithmen wurden jeweils mit $z' = 18$ getestet; für die Verfahren, die die Schritte 4 und 5 enthalten, wird unterschieden, ob der kleinste aufspannende Baum mit Kruskals oder mit Prims Algorithmus berechnet wurde.

2.3 Distance Network-Heuristik

Wir werden in diesem Abschnitt eine zweite Heuristik vom Typ PDH kennenlernen, welche von Winter und MacGregor-Smith den Namen „Distance Network-Heuristik" (DNH) bekommen hat; diese beiden Autoren schreiben das Verdienst, diese Heuristik zum ersten Mal veröffentlicht zu haben, zu gleichen Teilen Plesnik (1981), El-Arbi (1978) bzw. Kou, Markowsky und Berman (1981 in [KMB81]) zu. Es scheint jedoch so, daß die letztere Arbeit das größte Echo hervorgerufen hat ([RaC86], [WWW86], [Meh88] etc.).

Wir werden den Algorithmus hier zunächst genauso, wie es für SPH durchgeführt wurde, in der Form einer PDH formulieren, um dann zur Implementierung der DNH wie bei Wu et al.([WWW86]) zu gelangen. Dies läßt sich auch leicht mit dem etwas allgemeineren Ansatz in Übereinstimmung bringen, den Rayward-Smith und Clare in [RaC86] verwendet haben.

2.3.1 Algorithmus DNH

Winter und MacGregor-Smith stellen eine Version von DNH dar, welche wie SPH-z von einem Startknoten abhängt. Der im Original von Kou, Markowsky und Berman vorgestellte Algorithmus entbehrt dieser Einschränkung, liefert deshalb zunächst aber auch nur Näherungslösungen für USP'$_{G,Z}$; bezüglich dieses Unterschieds beachte man die Bemerkungen in [WiM92].

Man setzt einen ungerichteten zusammenhängenden Graphen mit

$$G = (V, E, d, g) \text{ mit } d(e) > 0, \quad \forall e \in E$$

und eine Teilmenge $Z \subseteq V$ mit $|Z| > 1$ voraus und formuliert DNH dem Aufbau einer PDH folgend in der ursprünglichen Gestalt gemäß [KMB81]:

Algorithmus 5 (Distance Network-Heuristik: DNH)

Schritt 1: Initialisierung

Setze $i = 0$ und betrachte mit $V_0 := Z$ und $E_0 := \emptyset$ den Graphen

$$T_0 := (V_0, E_0, d, g);$$

Schritt 2: Auswahl eines Pfades

$P_{i+1} = (v_{i+1,1}, \dots, v_{i+1,n_{i+1}})$ sei Pfad in G, für den gilt:

$$v_{i+1,1}, v_{i+1,n_{i+1}} \in Z \,, \tag{2.15}$$

$$[v_{i+1,1}]_{\overset{*}{\underset{T_i}{\leftrightarrow}}} \cap [v_{i+1,n_{i+1}}]_{\overset{*}{\underset{T_i}{\leftrightarrow}}} = \emptyset \,, \tag{2.16}$$

$$\begin{aligned} d(P_{i+1}) + g(P_{i+1}) = \min\{ & d_{\min(G)}(z_1, z_2) \mid \\ & z_1, z_2 \in Z, [z_1]_{\overset{*}{\underset{T_i}{\leftrightarrow}}} \cap [z_2]_{\overset{*}{\underset{T_i}{\leftrightarrow}}} = \emptyset \} \end{aligned} \tag{2.17}$$

Man setze dann

$$T_{i+1} := (V_{i+1}, E_{i+1}, d, g) \text{ mit}$$

$$V_{i+1} := V_i \cup \{ v_{i+1,j} \mid j \in \{2, \dots, n_{i+1} - 1\} \}$$

$$E_{i+1} := E_i \cup \bigcup_{k=1}^{n_{i+1}-1} \{v_{i+1,k}, v_{i+1,k+1}\}$$

und schließlich noch $i := i + 1$.

Schritt 3: Terminierung

Man breche ab, falls

$$\left|\,\{\,[z]_{\overset{*}{\underset{T_i}{\leftrightarrow}}} \mid z \in Z\,\}\,\right| = 1$$

gilt. Ansonsten wiederhole man Schritt 2.

Bemerkung 2.3.1

Wegen der Eigenschaften (2.15) und (2.16) der Endpunkte von P'_{i+1} folgt aus Bedingung (2.17), daß in Schritt 2 insbesondere gilt:

$$d(P_{i+1}) + g(P_{i+1}) = d_{\min(G)}(v_{i+1,1}, v_{i+1,n_{i+1}}) \ . \tag{2.18}$$

Da G wieder als zusammenhängend vorausgesetzt wurde, existiert ein Pfad P_{i+1} mit den drei obengenannten Eigenschaften, falls

$$\bar{z}_i := \left|\,\{\,[z]_{\overset{*}{\underset{T_i}{\leftrightarrow}}} \mid z \in Z\,\}\,\right| > 1 \ .$$

Dies ist wegen $|Z| > 1$ für $i = 0$ gegeben. Da aus den Bedingungen (2.15) und (2.16) die beiden Bedingungen für die Pfadauswahl in PDH (siehe Seite 23) folgen, gelten sofort auch für DNH die Sätze

Satz 2.3.2

DNH ist wohldefiniert und terminiert, falls $|Z| > 1$.

Satz 2.3.3

DNH liefert eine zulässige Lösung für USP'$_{G,Z}$.

Um DNH zur näherungsweisen Lösung von USTP'$_{G,Z}$ einsetzen zu können, kann man zum einen die beiden Schritte 4 und 5, die auf den Seiten 26ff. aufgeführt worden sind, an DNH anhängen; zum anderen folgt aus Gleichung (2.18) und der Voraussetzung $d_e > 0$ für alle $e \in E$ schon, daß alle in Schritt 2 gewählten Pfade P_{i+1} insbesondere einfach sind. Diese Beobachtung zusammen mit der unten angegebenen Bedingung (2.19) machen DNH zu einer Heuristik für USTP'$_{G,Z}$.

Man betrachte einen Pfad P_{i+1} wie er in Schritt 2 von Algorithmus 5 gewählt wird; mit der Zusatzbedingung

$$\begin{aligned} & v_{i+1,k_1}, v_{i+1,k_2} \in [z]_{\overset{*}{\underset{T_i}{\leftrightarrow}}} \text{ mit } 1 \leqslant k_1 < k_2 \leqslant n_{i+1}, z \in Z \\ \Rightarrow & \begin{cases} v_{i+1,k} \in [z]_{\overset{*}{\underset{T_i}{\leftrightarrow}}} \\ \{v_{i+1,k}, v_{i+1,k+1}\} \in E_i \end{cases} \quad \forall k_1 \leqslant k < k_2 \end{aligned} \tag{2.19}$$

gilt dann Satz 2.3.5. Zuvor muß man sich aber noch vergewissern, daß die Wohldefiniertheit von Algorithmus 5 bei Hinzunahme von Eigenschaft (2.19) erhalten bleibt.

Lemma 2.3.4

Für $i \in \mathbb{N} \cup \{0\}$ *mit* $\bar{z}_i > 1$ *existiert ein Pfad* P_{i+1}*, der die Eigenschaften aus Algorithmus 5 besitzt und gleichzeitig (2.19) erfüllt.*

Beweis.
Trivialerweise erfüllt jeder den Bedingungen aus DNH gemäß gewählte Pfad P_1 die Bedingung (2.19).

Sei nun $i \in \mathbb{N}$ mit $\bar{z}_i > 1$; T_i sei gemäß DNH aufgebaut worden, P_{i+1} sei gewählt wie in Schritt 2 und verletze die Bedingung (2.19), d.h. der Pfad bilde mit T_i zusammen eine Zyklus. Wir zeigen, daß ein Pfad P'_{i+1} existiert mit

$$d(P'_{i+1}) + g(P'_{i+1}) = d(P_{i+1}) + g(P_{i+1}) \ , \tag{2.20}$$

der neben den Eigenschaften, die Algorithmus 5 verlangt, zusätzlich (2.19) aufweist.

Wir nehmen an, P_{i+1} verletze (2.19) zwischen den Knoten v_{i+1,k_1} und v_{i+1,k_2}, d.h. ohne Einschränkung existiere ein Zyklus

$$(w_1, \ldots, w_m, \ldots, w_n)$$

mit den einfachen Pfaden $(2 < m < n)$:

$$(w_1, \ldots, w_{m-1}) \text{ und } (w_m, \ldots, w_n) \ ,$$

für die folgendes gilt:

$$\{\, \{w_k, w_{k+1}\} \mid k = 1, \ldots m-2 \,\} \subseteq E_i$$

$$\{w_m, \ldots, w_{n-1}\} \cap [w_1]_{\overset{*}{\underset{T_i}{\leftrightarrow}}} = \emptyset$$

$$(w_{m-1}, \ldots, w_n) = (v_{i+1,k_1}, \ldots, v_{i+1,k_2}) \ .$$

Man unterscheidet jetzt zwei Fälle. Ist

$$\{w_1, \ldots, w_{m-1}\} \cap Z = \emptyset \ ,$$

so wurde $(w_1, \ldots, w_{m-1})$ während einer früheren Iteration von Schritt 2 als Teil eines kürzesten Pfades eingefügt. Wegen (2.18) sind also sowohl $(w_{m-1}, w_{m-2}, \ldots, w_1)$ als auch das Tupel $(w_{m-1}, \ldots, w_n)$ kürzeste Pfade zwischen den Knoten v_{i+1,k_1} und v_{i+1,k_2}. Wir definieren

$$P' := (v_{i+1,1}, \ldots, v_{i+1,k_1-1}, \underbrace{w_{m-1}}_{=v_{i+1,k_1}}, w_{m-2}, \ldots, \underbrace{w_1}_{=v_{i+1,k_2}}, v_{i+1,k_2+1}, \ldots, v_{i+1,n_{i+1}})$$

und erhalten damit einen Pfad, welcher nun (2.19) zwischen v_{i+1,k_1} und v_{i+1,k_2} erfüllt, und für den gilt:

$$d(P') + g(P') = d(P_{i+1}) + g(P_{i+1}) \ .$$

Sukzessive Durchführung der obigen Schritte für alle Zyklen von P_{i+1} liefert schließlich den in der Behauptung gesuchten Pfad P'_{i+1}.

Wir zeigen nun, daß der Fall

$$\{w_1, \ldots, w_{m-1}\} \cap Z \neq \emptyset \ ,$$

überhaupt nicht eintreten kann. Es seien $i_0, i_1 \leqslant i$ die Iterationen, innerhalb derer die Knoten

$$v_{i+1,k_1} = v_{i_0,k_3} \text{ und } v_{i+1,k_2} = v_{i_1,k_4}$$

eingefügt worden sind. Man unterteilt die Pfade folgendermaßen:

$$P_{i_0} = (\underbrace{v_{i_0,1},\ldots,\overbrace{v_{i_0,k_3}}^{}}_{=:X_1}\!\!\overbrace{\ ,\ldots,v_{i_0,n_{i_0}}}^{=:X_2})$$

$$P_{i_1} = (\underbrace{v_{i_1,1},\ldots,\overbrace{v_{i_1,k_4}}^{}}_{=:X_3}\!\!\overbrace{\ ,\ldots,v_{i_1,n_{i_1}}}^{=:X_4})$$

$$P_{i+1} = (\underbrace{v_{i+1,1},\ldots,v_{i+1,k_1}}_{=:A},\ldots,\underbrace{v_{i+1,k_2},\ldots,v_{i+1,n_{i+1}}}_{=:B})$$

where $C := (v_{i+1,k_1},\ldots,v_{i+1,k_2})$ is marked by an overbrace.

Nach (2.17) gelten die sechs Ungleichungen:

$$d(B)+g(B)+d(C)+g(C)+g_{v_{i+1,k_2}} \leqslant \begin{cases} d(X_1)+g(X_1) \\ d(X_2)+g(X_2) \end{cases}$$

$$d(A)+g(A)+d(C)+g(C)+g_{v_{i+1,k_1}} \leqslant \begin{cases} d(X_3)+g(X_3) \\ d(X_4)+g(X_4) \end{cases}$$

$$d(A)+g(A)+d(B)+g(B)+d(C)+g(C)+g_{v_{i+1,k_1}}+g_{v_{i+1,k_2}} \geqslant$$
$$\geqslant \begin{cases} d(X_1)+g(X_1)+d(X_2)+g(X_2)+g_{v_{i+1,k_1}} \\ d(X_3)+g(X_3)+d(X_4)+g(X_4)+g_{v_{i+1,k_2}} \end{cases}$$

Man addiert die Ungleichungen zwei und drei zu den letzten beiden und erhält

$$d(A)+g(A)+d(B)+g(B) \geqslant d(X_1)+g(X_1)+d(X_4)+g(X_4)$$

im Widerspruch zur Summe der anderen beiden:

$$d(A)+g(A)+d(B)+g(B)+2d(C)+2g(C)+g_{v_{i+1,k_1}}+g_{v_{i+1,k_2}} \leqslant$$
$$\leqslant d(X_1)+g(X_1)+d(X_4)+g(X_4)\ .$$

□

Satz 2.3.5
DNH mit (2.19) liefert eine zulässige Lösung für USTP'$_{G,Z}$.

Beweis.
Wir zeigen per Induktion nach i, daß jede Zusammenhangskomponente von T_i auf T_i einen Baum induziert. Für $i = 0$ ist nichts zu zeigen, da in diesem Fall jede Komponente aus genau einem Knoten $z \in Z$ besteht.
Es gelte nun für ein $i \in \mathbb{N} \cup \{0\}$

$$\tilde{z}_i := \left|\left\{ [z]_{\underset{T_i}{\overset{*}{\leftrightarrow}}} \;\middle|\; z \in Z \right\}\right| > 1\ ;$$

weil G zusammenhängend ist, kann man einen Pfad P_{i+1} auswählen, der sowohl die Eigenschaften aus Algorithmus 5 als auch (2.19) erfüllt. Wir nehmen an, die Behauptung gelte für T_i; man betrachte nun den von der Zusammenhangskomponente von $z \in Z$ bezüglich T_{i+1} in T_{i+1} induzierten Teilgraphen

$$G_{z,T_{i+1}} := G'_{T_{i+1},[z]_{\overset{*}{\underset{T_{i+1}}{\longleftrightarrow}}}} .$$

Nach Induktionsannahme gilt zunächst

$$[z]_{\overset{*}{\underset{T_i+1}{\longleftrightarrow}}} \cap \{v_{i+1,1}, \dots, v_{i+1,n_{i+1}}\} = \emptyset \Rightarrow G_{z,T_{i+1}} = G_{z,T_i} \text{ ist Baum.}$$

Andernfalls nehmen wir an, $G_{z,T_{i+1}}$ sei kein Baum, d.h. dieser von $[z]_{\overset{*}{\underset{T_{i+1}}{\longleftrightarrow}}}$ induzierte Teilgraph enthält einen Zyklus $(w_1, \dots, w_n)$. Weil P_{i+1} gemäß Gleichung (2.18) einfacher Pfad ist und weil G_{z,T_i} nach Induktionsannahme Baum ist, können wir ohne Einschränkung annehmen, daß zwei einfache Pfade

$$(w_1, \dots, w_{m-1}) \text{ und } (w_m, \dots, w_{n-1})$$

existieren mit $2 < m < n$ und den Eigenschaften

$$\{\,\{w_k, w_{k+1}\} \mid k = 1, \dots m-2\,\} \subseteq E_i$$

$$\{w_m, \dots, w_{n-1}\} \subseteq [z]_{\overset{*}{\underset{T_i}{\longleftrightarrow}}}$$

$$\exists k \in \mathbb{N} : (w_{m-1}, \dots, w_n) = (v_{i+1,k}, \dots, v_{i+1,k+n-m+1})$$

d.h. der eine Teil des Zyklus verläuft innerhalb des von der Zusammenhangskomponente von z bzgl. T_i induzierten Teilgraphen, der andere außerhalb davon auf dem eingefügten Pfad P_{i+1}.
Wegen

$$w_{m-1} \in [z]_{\overset{*}{\underset{T_i}{\longleftrightarrow}}} \text{ und } w_1 = w_n \in [z]_{\overset{*}{\underset{T_i}{\longleftrightarrow}}}$$

steht $w_m \notin [z]_{\overset{*}{\underset{T_i}{\longleftrightarrow}}}$ aber im Widerspruch zur Eigenschaft (2.19) des Pfades P_{i+1}. Deshalb gilt die Behauptung für $i + 1$. □

Bemerkung 2.3.6
Bedingung (2.19) bewirkt, daß ein kürzester Pfad, der während einer Iteration von Schritt 2 eingefügt wird, dem schon bestehenden Graphen so weit wie möglich folgt. Dies ist auch – wie Winter in [WiM92] bemerkt – der Unterschied zwischen der Original-Distance Network-Heuristik und dem (von einem Startknoten abhängigen) Verfahren, das Winter und MacGregor-Smith darstellen.

Beispiel 2.3.7
Wir demonstrieren die Wirkung von (2.19) am in Abbildung 2.8 dargestellten Graphen $G = (V, E, d, g)$ mit Kantenbewertungen wie in der Abbildung, $g \equiv 0$ und $Z = \{1, 3, 5\}$.

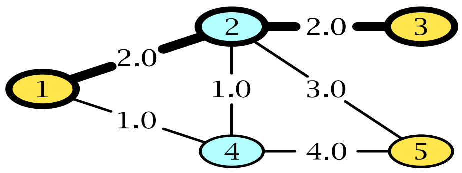

Abbildung 2.8: Gitter zur Demostration von DNH mit (2.19)

Nach der ersten Iteration von DNH liegen die Mengen $V_1 = \{1,2,3,5\}$ und $E_1 = \{\{1,2\},\{2,3\}\}$ vor. Um die Näherungslösung zu vervollständigen, stehen in der zweiten Iteration die beiden Pfade $P' = (1,4,2,5)$ und $P'' = (1,2,5)$ mit einem Gewicht von $d(P') = d(P'') = 5.0$ zur Wahl. Einhaltung der Bedingung (2.19) hat zur Folge, daß P'' eingefügt werden muß und der entstehende Graph T_2 ein Baum ist. Würde man P' einfügen, so enthielte T_2 den Zyklus $(1,2,4,1)$.

2.3.2 Algorithmus mit impliziter Pfad-Berechnung: DNIH

Im Gegensatz zur Shortest Path-Heuristik, die direkt wie angegeben implementiert wird, werden wir DNH in einer effizienteren Form realisieren, welche keine vorherige Berechnung kürzester Pfade braucht. In [WiM92] wird in diesem Zusammenhang auf die Arbeiten von Wu, Widmayer und Wong [WWW86], von Widmayer [Wid86] bzw. von Mehlhorn [Meh88] hingewiesen. Wu et al. haben ihre Heuristik, die hier DNIH genannt werden soll (DN-Heuristik mit impliziter Pfadberechnung), nur andeutungsweise beschrieben, weshalb wir sie hier ausführlich darstellen werden. Dabei verwendet man in jedem Schritt ($j \in \mathbb{N} \cup \{0\}$) folgende Funktionen auf den Knoten von G:

$$\begin{aligned} s_j &: V \to Z \cup \{\diamond\}\ , \\ l_j &: V \to \mathbb{R}_{\geqslant 0} \cup \{\infty\}\ , \\ p_j &: V \to V \cup \{\diamond\}\ . \end{aligned}$$

Man setzt $\diamond \notin V$ voraus und interpretiert $\diamond$ als „nicht definiert“. Weiter werden noch Mengen

$$Q_j \subseteq \{\, (d, v, z, \bar{z}, \bar{v}) \mid d \in \mathbb{R}_{>0}, v \in V, z \in Z, \bar{z}, \bar{v} \in V \cup \{\diamond\} \,\}$$

vorkommen. Es gelten die Voraussetzungen von DNH; außerdem sei die Menge spezieller Knoten $Z = \{z^1, \ldots, z^{|Z|}\}$ abgezählt.

Algorithmus 6 (DNH mit impliziter Pfadberechnung: DNIH)

Schritt 1: Initialisierung

Setze $i = 0$ und betrachte mit $V_0 := Z$ und $E_0 := \emptyset$ den Graphen

$$T_0 := (V_0, E_0, d, g)\ .$$

Setze $j = 0$ und betrachte

$$\begin{aligned} s_0(z) := p_0(z) &:= z, \qquad l_0(z) := 0.0, \quad \forall z \in Z \\ s_0(v) := p_0(v) &:= \diamond, \qquad l_0(v) := \infty, \quad \forall v \in V \setminus Z\ . \end{aligned}$$

Außerdem sei am Anfang

$$\begin{aligned} Q_0 := &\{\, (d_{\{v,z\}}, v, z, z, \diamond) \mid v \in V\setminus Z, z \in Z, \{v,z\} \in E \,\} \cup \\ &\{\, (d_{\{z^k,z^l\}}, z^k, z^l, z^l, z^k) \mid \{z^k, z^l\} \in E, 1 \leqslant k < l \leqslant |Z| \,\}\ . \end{aligned}$$

Schritt 2: Pfadkonstruktion, Erweiterung der Näherung

Wähle ein $q_j := (d_j, v_j, z_j, \bar{z}_j, \bar{v}_j) \in Q_j$ mit der Eigenschaft, daß

$$d_j = \min\{\, d \mid (d, v, z, \bar{z}, \bar{v}) \in Q_j \,\}\ .$$

Fall 1: $s_j(v_j) = \diamond$:

In diesem Fall erhält man neue Funktionen durch

$$s_{j+1}(v) := \begin{cases} z_j & \text{falls } v = v_j, \\ s_j(v) & \text{sonst} \end{cases}$$

$$l_{j+1}(v) := \begin{cases} d_j & \text{falls } v = v_j, \\ l_j(v) & \text{sonst} \end{cases}$$

$$p_{j+1}(v) := \begin{cases} \bar{z}_j & \text{falls } v = v_j, \\ p_j(v) & \text{sonst} \end{cases}$$

Man setzt

$$\begin{aligned} Q_{j+1} := Q_j \setminus \{q_j\} \cup \{\, (d_j + d_{\{v_j,v\}} + g_{v_j}, v, z_j, v_j, \diamond) \mid \\ \{v_j, v\} \in E, s_j(v) = \diamond \,\} \end{aligned}$$

und schließlich $j := j + 1$. Man wiederhole Schritt 2.

Fall 2: $s_j(v_j) \neq \diamond$ und $z_j \in [s_j(v_j)]_{\overset{*}{\underset{T_i}{\leftrightarrow}}}$

In diesem Fall passiert praktisch nichts:

$$s_{j+1} := s_j, \qquad l_{j+1} := l_j, \qquad p_{j+1} := p_j$$
$$Q_{j+1} := Q_j \setminus \{q_j\}$$

Vor der Wiederholung von Schritt 2 setzt man $j := j + 1$.

Fall 3: $s_j(v_j) \neq \diamond$ und $z_j \notin [s_j(v_j)]_{\overset{*}{\underset{T_i}{\leftrightarrow}}}$

$$s_{j+1} := s_j, \qquad l_{j+1} := l_j, \qquad p_{j+1} := p_j$$

Falls $v_j \in Z$ gilt, fügt man z_j und v_j mit Hilfe der Pfadinformationen in $\bar{z}_j$, $\bar{v}_j$ und p_j zusammen (siehe Algorithmus 7); man erhält

T_{i+1} und setzt $i := i+1$; wie in Fall 2 definiert man außerdem $Q_{j+1} := Q_j \setminus \{q_j\}$.
Falls dagegen $v_j \notin Z$ gilt, setze man

$$Q_{j+1} := Q_j \setminus \{q_j\} \cup \{\, (d_j + l_j(v_j) + g(v_j), s_j(v_j), z_j, \bar{z}_j, v_j) \,\} .$$

Schließlich noch $j := j+1$, dann Sprung zu Schritt 3.

Schritt 3: Terminierung
Man breche ab, falls

$$|\, \{\, [z]_{\underset{T_i}{\overset{*}{\leftrightarrow}}} \mid z \in Z \,\} \,| = 1$$

gilt. Ansonsten wiederhole man Schritt 2.

Es ist noch zu spezifizieren, wie die Rekonstruktionen der Pfade in Schritt 2, Fall 3 für $v_j \in Z$ erfolgt. Es seien jetzt $i, j \in \mathbb{N} \cup \{0\}$. Weiter sei q_j wie in Algorithmus 6 gewählt. Die Pfade werden konstruiert durch

Algorithmus 7 (Pfadkonstruktion in DNIH)

Schritt 1: Initialisierung der Rekonstruktion
Es seien $k := 0$ und ein Baum $T_{i,0} := (V_{i,0}, E_{i,0}, d, g)$ gegeben durch

$$V_{i,0} := V_i \cup \{\bar{z}_j, \bar{v}_j\},\; E_{i,0} := E_i \cup \{\, \{z_j, v_j\} \,\} .$$

Schritt 2: Zurückverfolgen des Pfades ab $\bar{z}_j$

i. Man setzt $w_{i+1,0} := \bar{z}_j$.
ii. Falls $w_{i+1,k} = p_j(w_{i+1,k})$, geht man zu Schritt 3.
Ansonsten setze man $T_{i+1,k+1} := (V_{i+1,k+1}, E_{i+1,k+1}, d, g)$ mit

$$V_{i+1,k+1} := V_{i+1,k} \cup \{p_j(w_{i+1,k})\},$$
$$E_{i+1,k+1} := E_{i+1,k} \cup \{\, \{w_{i+1,k}, p_j(w_{i+1,k})\} \,\} .$$

Es seien $w_{i+1,k+1} := p_j(w_{i+1,k})$ und $k := k+1$.
Man wiederhole diesen Teilschritt ii.

Schritt 3: Zurückverfolgen des Pfades ab $\bar{v}_j$

i. Man setzt $w_{i+1,k} := \bar{v}_j$.
ii. Man iteriert Teilschritt ii. von Schritt 2 und geht, falls $w_{i+1,k} = p_j(w_{i+1,k})$ gilt, dann zu Schritt 4.

Schritt 4: Ergebnis
Man setzt $T_{i+1} := T_{i+1,k}$ und $i := i+1$.

Natürlich kann man, die Wohldefiniertheit der DNI-Heuristik voraussetzend, auch hier versuchen, die erhaltene Näherungslösung mit Hilfe der Schritte 4 und 5 aus der PD-Heuristik (siehe Unterabschnitt 2.1.3) zu verbessern.

2.3.3 Eigenschaften von DNIH

Zunächst einige allgemeine Eigenschaften von Algorithmus 6:

Lemma 2.3.8
Fall 1 von Schritt 2 wird höchstens $|V\setminus Z|$-mal durchlaufen. Es gilt:

$$s_{j_0}(v) \neq \diamond \Rightarrow \begin{cases} s_j(v) = s_{j_0}(v) \\ l_j(v) = l_{j_0}(v) \\ p_j(v) = p_{j_0}(v) \end{cases} \qquad \forall\, j \geqslant j_0\ .$$

Beweis.
Nach der Initialisierung, die in Schritt 1 erfolgt, ist

$$\big|\{\, v \in V \mid s_0(v) = \diamond \,\}\big| = \big|V\setminus Z\big|\ .$$

In jeder Iteration $j \geqslant 0$ von Schritt 2 gilt:

$$s_j(v) \neq \diamond \Rightarrow s_{j+1}(v) = s_j(v), \quad \forall\, v \in V$$

und, falls Fall 1 durchlaufen wird:

$$\begin{aligned} \big|\{\, v \in V \mid s_{j+1}(v) = \diamond \,\}\big| &= \big|\{\, v \in V \mid s_j(v) = \diamond \,\}\setminus\{v_j\}\big| = \\ &= \big|\{\, v \in V \mid s_j(v) = \diamond \,\}\big| - 1\ . \end{aligned}$$

Daraus folgt die Behauptung. □

Bemerkung 2.3.9
Folgende drei Bedingungen sind gleichbedeutend:

$$s_j(v) \neq \diamond \Leftrightarrow p_j(v) \neq \diamond \Leftrightarrow l_j(v) \neq \infty \quad \forall\, v \in V, j \in \mathbb{N} \cup \{0\}\ .$$

Lemma 2.3.10
Betrachte die Folge $(q_j)_{j\in\mathbb{N}\cup\{0\}}$ der durch Schritt 2 ausgewählten Tupel.
Es gilt:

$$0 < d_j \leqslant d_{j+1}, \quad \forall\, j \in \mathbb{N} \cup \{0\}\ .$$

Beweis.
Wegen der Minimalitätseigenschaft von d_j gilt:

$$d_j \leqslant d, \quad \forall\, (d, v, z, \bar{z}, \bar{v}) \in Q_j\ .$$

Nach Konstruktion von Q_{j+1} gilt auch

$$d_j < d, \quad \forall\, (d, v, z, \bar{z}, \bar{v}) \in Q_{j+1}\setminus Q_j\ .$$

Außerdem ist

$$d > 0, \quad \forall\, (d, v, z, \bar{z}, \bar{v}) \in Q_0$$

wegen $d_e > 0$ für alle Kanten $e \in E$. □

Im folgenden bezeichne 2^M die Potenzmenge einer Menge M; wir definieren Funktionen l, s und $\mathcal{P}$ auf dem zusammenhängenden Graphen G:

$$\begin{aligned} l &: V \to \mathbb{R}_{\geqslant 0}; && v \mapsto d_{\min(G)}(Z, v)\ ; \\ s &: V \to 2^Z; && v \mapsto \{\, z \in Z \mid d_{\min(G)}(z, v) = l(v) \,\}\ ; \\ \mathcal{P} &: V \to 2^{\mathcal{P}_G(s(v),v)}; && v \mapsto \{\, P \in \mathcal{P}_G(s(v), v) \mid d(P) + g(P) = l(v) \,\}\ ; \end{aligned}$$

außerdem sei $p : V \to 2^V$ gegeben durch

$$p(v) := \begin{cases} \{\, w \in V \mid \exists (v_1, \ldots, v_n) \in \mathcal{P}(v),\ w = v_{n-1} \,\}, & \forall v \in V \backslash Z \\ \{v\}, & \forall v \in Z \end{cases}$$

$l(v)$ stellt also den Abstand des Knotens v von nächstliegenden speziellen Knoten dar; genau diese sind in $s(v)$ enthalten. Die Menge $p(v)$ bezeichnet alle „Vorgänger" von v auf den von $z \in s(v)$ nach v möglichen Pfaden (aus $\mathcal{P}(v)$). Für $z \in Z$ gilt trivialerweise $l(z) = 0$, $s(z) = z$; weiter wegen $d > 0$ außerdem $\mathcal{P}(z) = \{(z)\}$ und $p(z) = \{z\}$. Wegen der Positivität von d und der Minimalitätsforderung in der Definition von $\mathcal{P}$ muß jeder Pfad $P \in \mathcal{P}(v)$ für $v \in V$ einfacher Pfad sein. Im allgemeinen hat man auch sofort

Lemma 2.3.11
Seien $v \in V$ und $(v_i)_{i \in \mathbb{N}}$ eine Folge mit $v_1 = v$ und $v_{i+1} \in p(v_i)$ für alle $i \in \mathbb{N}$. Ein Index $n_0 \in \mathbb{N}$ sei so gewählt, daß gilt:

$$v_{n_0-1} \neq v_{n_0} = v_{n_0+1} = \ldots\ .$$

Für Pfade $P_i := (v_{n_0}, v_{n_0-1}, \ldots, v_{n_0-i})$ mit $0 \leqslant i < n_0$ gilt dann:

$$\begin{aligned} P_i &\in \mathcal{P}(v_{n_0-i}) \\ d(P_i) + g(P_i) &= l(v_{n_0-i}) \\ v_{n_0} &\in s(v_{n_0-i})\ . \end{aligned}$$

Die nun folgenden Sätze führen zur Wohldefiniertheit von Algorithmus 6.

Satz 2.3.12
In jeder Iteration $j \geqslant 0$ von Schritt 2 der DNIH gilt:

$$s_j(v) \neq \diamond \Rightarrow l_j(v) = l(v)\ .$$

Beweis.
Wegen der Initialisierung in Schritt 1 gilt die Behauptung trivialerweise für $j = 0$.
Sie gelte nun für ein $j \geqslant 0$. Trifft in diesem Schritt Fall 2 oder Fall 3 zu, so gilt die Behauptung wegen der in diesen Fällen erfolgenden Definitionen von s_{j+1} und l_{j+1} auch für $j + 1$.
Es bleibt im Fall 1 wegen $l_{j+1}(v_j) := d_j$ zu überprüfen, ob $d_j = l(v_j)$ gilt. Wir nehmen an, es gelte $d_j \neq l(v_j)$ und damit schon $d_j > l(v_j)$. Man betrachte einen Pfad $P := (w_0, \ldots, w_n) \in \mathcal{P}(v_j)$ mit $n \in \mathbb{N}$.
Wäre $s_j(w_{n-1}) \neq \diamond$ so würde nach Induktionsannahme

$$l_j(w_{n-1}) = l(w_{n-1})$$

gelten; außerdem wäre

$$(\underbrace{l(w_{n-1}) + g(w_{n-1}) + d_{\{w_{n-1},w_n\}}}_{=l(w_n)=l(v_j)<d_j}, v_j, z_j, \bar{z}_j, \diamond) \in Q_j\ .$$

Dies ist ein Widerspruch zur Minimalität von d_j in Q_j, und es muß demnach $s_j(w_{n-1}) = \diamond$ gelten.

Wir wiederholen diesen Schluß per Induktion über die Anzahl der Knoten in P für $1 < i < n$. Wir setzen $s_j(w_{n-i+1}) = \diamond$ voraus; wäre $s_j(w_{n-i}) \neq \diamond$ so würde nach Induktionsannahme gelten, daß

$$l_j(w_{n-i}) = l(w_{n-i})$$
$$(\underbrace{l(w_{n-i}) + g(w_{n-i}) + d_{\{w_{n-i},w_{n-i+1}\}}}_{=l(w_{n-i+1})<l(w_{n-i+2})<d_j}, w_{n-i+1}, z_j, \bar{z}_j, \diamond) \in Q_j \ .$$

Dies ist ein Widerspruch zur Minimalität von d_j in Q_j, und man hat wieder $s_j(w_{n-i}) = \diamond$.
Schließlich gilt $s_j(w_1) = \diamond$ und $s_j(w_0) \neq \diamond$ wegen $w_0 \in Z$. Dann muß aber

$$(d_{\{w_0,w_1\}}, w_1, w_0, w_0, \diamond) \in Q_j$$

gelten mit $d_{\{w_0,w_1\}} < l(v_j) < d_j$ im Widerspruch zur Minimalität von d_j. Deshalb kann insgesamt nicht $l(v_j) < d_j$ gelten, und man hat $l_{j+1}(v_j) = l(v_j)$ für $j + 1$ gezeigt. □

Genau wie im vorangehenden Beweis für l_j zeigt man folgendes

Corollar 2.3.13
Für beliebige $j \geqslant 0$ und $v \in V$ gilt:

$$s_j(v) \neq \diamond \Rightarrow \begin{cases} p_j(v) \in p(v) & \textit{und} \\ s_j(v) \in s(v) \ . \end{cases}$$

Mit Hilfe von Satz 2.3.12 kann man nun auch beweisen:

Satz 2.3.14
Seien $j \in \mathbb{N} \cup \{0\}$ und $q_j \in Q_j$ gewählt wie in Algorithmus 6, wobei das Tupel die Gestalt $q_j = (d_j, v_j, z_j, \bar{z}_j, \bar{v}_j)$ besitze.
Dann gilt für $v \in V$:

$$l(v) < d_j \qquad \Rightarrow \qquad \exists\, j_0 \leqslant j : s_{j_0}(v) \neq \diamond \ .$$

Beweis.
Die Behauptung gilt ohne weiteres für $j = 0$.
Sie gelte nun für ein $j \geqslant 0$. Wir nehmen an, es gäbe ein $v \in V$ mit $l(v) < d_j$ und $s_{j+1}(v) = \diamond$. Damit gilt wegen Lemma 2.3.8 insbesondere, daß

$$s_{j'}(v) = \diamond, \quad \forall 0 \leqslant j' \leqslant j \ .$$

Betrachte nun einen Pfad $P \in \mathcal{P}(v)$, der die Gestalt $P = (w_0, \ldots, w_n)$ besitze. Es gilt nach Satz 2.3.12:

$$s_j(w_{n-1}) \neq \diamond \Rightarrow l_j(w_{n-1}) = l(w_{n-1})$$

und damit wegen Schritt 2, Fall 1

$$(\underbrace{l(w_{n-1}) + g(w_{n-1}) + d_{\{w_{n-1},w_n\}}}_{=l(w_n)=l(v)<d_j}, v, z, \bar{z}, \diamond) \in Q_j \ .$$

Dies ist ein Widerspruch zur Minimalität von d_j in Q_j, und es muß demnach $s_j(w_{n-1}) = \diamond$ gelten.

Man fährt nun induktiv wie im letzten Beweis fort und erhält dadurch wieder die im Widerspruch zu Initialisierung von $s_0(w_0) = w_0 \neq \diamond$ stehende Gleichung $s_j(w_0) = s_0(w_0) = \diamond$.
Insgesamt gilt die Behauptung also auch für $j+1$. □

Wir sind jetzt imstande zu zeigen, daß die Pfade, welche in Schritt 2, Fall 3 von der DNI-Heuristik für $q_j = (d_j, v_j, z_j, \bar{z}_j, \bar{v}_j)$ für $v_j \in Z$ in den Baum T_i eingefügt werden, tatsächlich den Bedingungen (2.15) bis (2.17) entsprechen. Ein derartiger Pfad hat für $i \in \mathbb{N} \cup \{0\}$ mit $n' \in \mathbb{N}$ und $n'' \in \mathbb{N} \cup \{0\}$ die Gestalt

$$P'_{i+1} = (w_{i+1,n'-1}, \ldots, w_{i+1,0}, w_{i+1,n'}, \ldots, w_{i+1,n'+n''})$$

mit den Eigenschaften

$$\begin{aligned} w_{i+1,0} &= \bar{z}_j & \qquad w_{i+1,n'} &= \bar{v}_j \\ p_j(w_{i+1,n'-1}) &= w_{i+1,n'-1} & \qquad p_j(w_{i+1,n'+n''}) &= w_{i+1,n'+n''} \end{aligned}$$

und

$$p_j(w_{i+1,n}) = w_{i+1,n+1} \quad \forall n \in \{0, \ldots, n'-2, n', \ldots, n'+n''-1\}\ .$$

Während des Ablaufs von Algorithmus 7 werden lediglich die in p_j enthaltenen Pfadinformationen für $\bar{z}_j$ und $\bar{v}_j$ zurückverfolgt. Aufgrund des obigen Corollars und des Satzes 2.3.14 sind alle während der Schleifen in Schritt 2/3, ii. von Algorithmus 7 verwendeten Werte $p_j(w_{i+1,n}) \neq \diamond$ für $i \geqslant 0$ und $0 \leqslant n \leqslant n' + n''$. Dieses Unterprogramm ist also wohldefiniert und liefert wegen Corollar 2.3.13 und der Monotonie der d_j nach Lemma 2.3.10 einen kürzesten Pfad zwischen $w_{i+1,n'-1}$ und $w_{i+1,n'+n''}$. Es terminiert und liefert den Pfad P'_{i+1} mit Eigenschaft (2.15), da nach Initialisierung in Algorithmus 6 gilt:

$$p_j(v) = v \Leftrightarrow v \in Z\ .$$

DNIH setzt für Schritt 2, Fall 3 voraus, daß $w_{i+1,n'-1} \notin [s_j(w_{i+1,n'+n''})]_{\overset{*}{\underset{T_i}{\longleftrightarrow}}}$ gilt; deshalb trifft auch (2.16) für P'_{i+1} zu. Die Pfade werden mit Hilfe der p_j rekonstruiert, welche , wenn einmal definiert, nach Lemma 2.3.8 konstant bleiben. Deshalb gilt auch die Zusatzbedingung (2.19) ohne weiteres.

Es bleibt zu zeigen, daß Bedingung (2.17) für den Pfad P'_{i+1} gilt. Wir nehmen im Gegenteil an:

$$\begin{gathered} \exists 1 \leqslant k' < l' \leqslant |Z|,\ [z^{k'}]_{\overset{*}{\underset{T_i}{\longleftrightarrow}}} \cap [z^{l'}]_{\overset{*}{\underset{T_i}{\longleftrightarrow}}} = \emptyset : \\ d(P'_{i+1}) + g(P'_{i+1}) > d_{\min(G)}(z^{k'}, z^{l'})\ . \end{gathered}$$

Man fordert weiter, daß sogar gelte:

$$\begin{aligned} d_{\min(G)}(z^k, z^l) \geqslant d_{\min(G)}(z^{k'}, z^{l'}) \quad & \forall 1 \leqslant k < l \leqslant |Z| \\ & [z^k]_{\overset{*}{\underset{T_i}{\longleftrightarrow}}} \cap [z^l]_{\overset{*}{\underset{T_i}{\longleftrightarrow}}} = \emptyset \end{aligned}$$

Falls $z^{k'} \in \mathrm{adj}_G(z^{l'})$, so gilt wegen Schritt 1

$$(d_{\min(G)}(z^{k'}, z^{l'}), z^{k'}, z^{l'}, z^{l'}, z^{k'}) \in Q_j$$

im Widerspruch zur Minimalität von d_j. Andernfalls gibt es einen Pfad

$$\begin{aligned} P &= (w_1, \ldots, w_{m'}) \in \mathcal{P}_G(z^{k'}, z^{l'}) \\ d(P) + g(P) &= d_{\min(G)}(z^{k'}, z^{l'}) \ . \end{aligned}$$

Man wähle $1 \leqslant m'' < m'$ so, daß gilt:

$$w_1 \in s(w_{m''}) \text{ und } w_{m'} \in s(w_{m''+1}) \ .$$

Aufgrund der Widerspruchsannahme und wegen Satz 2.3.14 gilt $s(w_m) \neq \diamond$ für alle $1 \leqslant m \leqslant m'$, und wir können wegen der oben getroffenen Zusatzvoraussetzung ohne Einschränkung annehmen, daß

$$s_j(w_{m''}) = w_1 \text{ und } s_j(w_{m''+1}) = w_{m'} \ .$$

Wegen Fall 1 des Schrittes 2 muß nun ein $j_0 < j$ existieren, sodaß wenigstens eine der beiden folgenden Inklusionen gilt:

$$\begin{aligned} (d_{\{w_{m''}, w_{m''+1}\}}, w_{m''}, w_{m'}, w_{m''+1}, \diamond) &\in Q_{j_0} \text{ oder} \\ (d_{\{w_{m''}, w_{m''+1}\}}, w_{m''+1}, w_1, w_{m''}, \diamond) &\in Q_{j_0} \ . \end{aligned}$$

Dann aber wurde Fall 3 für $w_{m''} \notin Z$ oder für $w_{m''+1} \notin Z$ durchlaufen, und es muß eines der beiden folgenden Tupel in der aktuellen Kandidatenmenge enthalten sein:

$$\begin{aligned} (d_{\min(G)}(z^{k'}, z^{l'}), z^{k'}, z^{l'}, w_{m''+1}, w_{m''}) &\in Q_j \text{ oder} \\ (d_{\min(G)}(z^{k'}, z^{l'}), z^{l'}, z^{k'}, w_{m''}, w_{m''+1}) &\in Q_j \ . \end{aligned}$$

In jedem Fall ergibt sich ein Widerspruch zur Minimalität von d_j; insgesamt gilt also auch (2.17) für jeden Pfad P'_{i+1} ($i \in \mathbb{N} \cup \{0\}$).

Alles in allem haben wir gesehen, daß die DNI-Heuristik eine Näherungslösung für USP'$_{G,Z}$ aus Pfaden aufbaut, die auch DNH (mit Zusatzbedingung) verwenden darf. Deswegen lassen sich die Sätze 2.3.2 und 2.3.5 auf die Distance Network-Heuristik mit impliziter Pfadberechnung übertragen.

Satz 2.3.15
DNIH ist wohldefiniert und terminiert.

Satz 2.3.16
DNIH liefert eine zulässige Lösung für USTP'$_{G,Z}$.

2.3.4 Komplexität von DNIH

Man macht zunächst folgende Beobachtung:

Lemma 2.3.17
Für die Anzahl der Elemente in Q_j gilt:

$$|Q_j| \leqslant |E|, \quad \forall j \in \mathbb{N} \cup \{0\} \ .$$

Beweis.
Das Einfügen von Elementen in Q_j findet in Schritt 1 und Schritt 2 (Fall 1 oder Fall 3) statt. Wird Fall 3 durchlaufen, so ist $|Q_{j+1}| = |Q_j|$. Während Schritt 2, Fall 1 wird ein Tupel der Gestalt $(d, v, z, \bar{z}, \diamond)$ nur in die Menge Q_{j+1} eingefügt, wenn $s_j(v) = \diamond$ gilt; außerdem wird dabei auch $s_{j+1}(\bar{z}) := z \neq \diamond$ gesetzt. In der Initialisierungsphase wurde ebenfalls für jedes hinzukommende Tupel obiger Gestalt $s_0(\bar{z}) := z \neq \diamond$ gesetzt. Außerdem wurde die Menge Z spezieller Knoten geordnet, um doppeltes Einfügen von Tupeln für Kanten zwischen speziellen Knoten zu verhindern.

Dies alles bedeutet, daß einmal beim Einfügen eines Tupels in Q_{j+1} benutzte Kanten von G hierfür kein zweites Mal benutzt werden können. □

Wir werden die Mengen Q_j durch eine Priority-Queue (siehe 3) darstellen. Das Einfügen bzw. Löschen eines Tupels aus einer derartigen Datenstruktur, die nach Lemma 2.3.17 eine Länge von maximal $|E|$ hat, läuft in der Zeit $O(\log |E|)$ ab; wegen der Ungleichung $|E| \leqslant |V|^2$ benutzen Wu et al. stattdessen $O(\log |V|)$.

Wir betrachten nun DNIH Schritt für Schritt; dabei beachte man, daß die s_j, l_j und p_j später durch je ein Array realisiert werden, das an den entsprechenden Stellen nur angepaßt wird.

Während der Initialisierung werden diese Arrays in $O(|V|)$, die Priority-Queue in $O(|E| \log |V|)$ vorbesetzt. Die Fälle 1 und 2 von Schritt 2 führen nach obigem Lemma höchstens $O(|E|)$ Einfüge- und Löschoperationen auf der Queue aus. Da nun – ebenfalls wegen Lemma 2.3.17 – Schritt 2, Fall 3 mit $v_j \notin Z$ höchstens $|E|$-mal durchlaufen wird, hat man als Komplexität von Schritt 2 den Term $O(|E| \log |V|)$. Man weiß nämlich, daß Fall 3 mit $v_j \in Z$ insgesamt höchstens $O(|Z|)$-mal ein Tupel in der Queue löscht und insgesamt höchstens $|V| - 1$ Kanten in die Näherungslösung einfügt.

Der Union-Find-Algorithmus (siehe Abschnitt 3.1.1), der eingesetzt wird, um über die aktuell vorliegenden Zusammenhangskomponenten Buch zu führen, hat für jede Operation eine Komplexität, welche auf keinen Fall schlechter ist als $O(\log |V|)$. Auch für Schritt 3 erhält man daher eine worst-case-Komplexität von $O(|E| \log |V|)$.

Die gesamte Laufzeit der DNI-Heuristik kann man nach obigen Bemerkungen jetzt angeben zu $O(|E| \log |V|)$.

2.3.5 Beispiel

Wie betrachten den ungerichteten Graphen aus Abbildung 2.9 . Die Kantenbewertungen sind dort abzulesen, die Knoten des Graphen seien alle mit 1.0 bewertet. Wir suchen einen Steiner-Baum für $Z = \{1, 4, 5\}$.

Die Zustände, die die Tupel s_j, l_j und p_j im Verlauf des Algorithmus einnehmen, sind in Tabelle 2.2 dargestellt; analog dazu die Elemente von

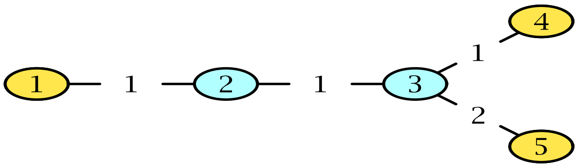

Abbildung 2.9: Beispiel für DNIH

	1	2	3	4	5
s_j	1	1	4	4	5
l_j	0.0	1.0	1.0	0.0	0.0
p_j	1	1	4	4	5
	$j = 0$	$j = 1$	$j = 2$	$j = 0$	$j = 0$

Tabelle 2.2: Werte von s_j, l_j und p_j

der Mengen Q_j:

$$\begin{array}{lcr}
\xrightarrow{j=0} & (1.0, 2, 1, 1, \diamond) & \xrightarrow{j=1\ (1)} \\
\xrightarrow{j=0} & (*)\ (1.0, 3, 4, 4, \diamond) & \xrightarrow{j=2\ (1)} \\
\xrightarrow{j=0} & (*)\ (2.0, 3, 5, 5, \diamond) & \xrightarrow{j=3\ (3.2)} \\
\xrightarrow{j=1} & (*)\ (3.0, 3, 1, 2, \diamond) & \xrightarrow{j=4\ (3.2)} \\
\xrightarrow{j=3} & (4.0, 4, 5, 5, 3) & \xrightarrow{j=5\ (3.1)} \\
\xrightarrow{j=4} & (5.0, 4, 1, 2, 3) & \xrightarrow{j=6\ (3.1)}
\end{array}$$

In der Initialisierungsphase (Schritt 1) von Algorithmus 6 sind nur die Tupel in Q_0 enthalten, welche mit $j = 0$ gekennzeichnet sind. Ebenso sind die Funktionen s_0 und p_0 nur für die Knoten aus Z definiert, und nur für diese Knoten ist l_0 am Anfang $< \infty$.

Der Pfeil rechts vom Tupel $(1.0, 2, 1, 1, \diamond)$ besagt, daß dieses Tupel in der ersten Iteration aus Q_0 abgezogen wird und in Schritt 2 des Algorithmus der Fall 1 eintritt. Während der Abarbeitung dieses Falls gehen $s_1(2)$ und $p_1(2)$ in einen definierten Zustand über; $l_1(2) := 1.0$ und ersetzt den Wert $l_0(2) = \infty$. Außerdem wird in der ersten Iteration ein weiteres Tupel in Q_1 eingefügt, welches Pfadinformation über einen Pfad der Länge 3.0 vom Knoten 3 zum speziellen Knoten 1 via Knoten 2 enthält.

Ähnliches geschieht in der zweiten Iteration mit Knoten 3, nur daß diesmal, da jetzt alle Knoten besucht wurden (d.h. $s_2(v) \neq \diamond$ für alle $v \in V$),

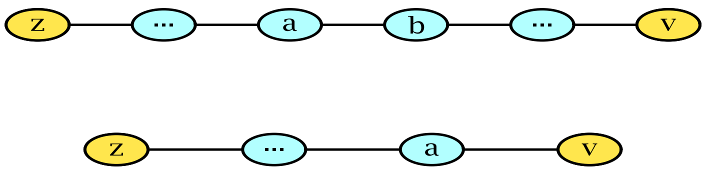

Abbildung 2.10: Zwei Arten von Tupeln in DNIH

Fall 1 keine Tupel mehr in Q_2 einfügt. Keine der drei Funktionen aus Tabelle 2.2 wird im weiteren Verlauf noch geändert werden, da Fall 1 aus Schritt 2 nicht mehr eintreten wird. Man beachte, daß in diesem Iterationsschritt die drei mit $(*)$ gekennzeichneten Tupel darum konkurrieren, Knoten Nummer 3 mit einem der speziellen Knoten zu verbinden. Der Pfad von 3 nach 4 stellt sich schließlich als der kürzeste heraus; die Funktionen in Tabelle 2.2 für $v = 3$ werden entsprechend gesetzt.

Der restliche Ablauf ist klar: die verbleibenden Tupel werden in den Iterationen $j = 3$ und $j = 4$ zu den Tupeln $(4.0, 4, 5, 5, 3)$ und $(5.0, 4, 1, 2, 3)$ umgewandelt, welche Kandidaten für kürzeste Pfade zwischen speziellen Knoten darstellen.

Die Lösung schließlich hat, weil der Ausgangsgraph so einfach beschaffen ist, ebenfalls die in Abbildung 2.9 dargestellte Form; sie setzt sich zusammen aus dem kürzesten Pfad zwischen den Knoten 4 und 5 via Knoten 3 und dem kürzesten Pfad zwischen 4 und 1 vorbei an den Knoten 2 und 3.

Wir verdeutlichen die Interpretation der beiden Arten von Tupeln, die während des Algorithmus 6 in Q_j für $j \in \mathbb{N} \cup \{0\}$ auftreten können, in der folgenden Bemerkung. Es sei $(d, v, z, a, b) \in Q_j$.

Bemerkung 2.3.18

$b \neq \diamond$ Abbildung 2.10 illustriert diesen Fall (gleichbedeutend mit $v \in Z$) mit der oberen Skizze eines Pfades in G. Die Pfade von a nach $z = s_j(a)$ bzw. von b nach $v = s_j(b)$ können mit Hilfe von Algorithmus 7 zurückverfolgt werden. Tupel dieser Art sind Kandidaten für Pfade zwischen speziellen Knoten in der Näherungslösung für USTP'$_{G,Z}$. Die Länge des Pfades ist d.

$b = \diamond$ Dieser Fall ist gleichbedeutend mit $v \in V \setminus Z$ und im unteren Teil von Abbildung 2.10 dargestellt. Die Knoten v und a des durch ein derartiges Tupel repräsentierten Pfades sind mit einer Kante verbunden, der Pfad von a nach $z = s_j(a)$ könnte mit Hilfe von p_j iterativ zurückverfolgt werden. Solche Tupel dienen als Kandidaten für einen kürzesten Pfad zwischen v und z; anders formuliert: z ist Kandidat für $s_{j_0}(v)$ (in einem späteren Iterationsschritt $j_0 \geqslant j$).

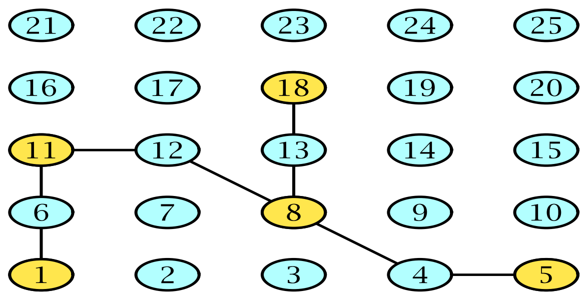

Abbildung 2.11: Lösung mit DNIH

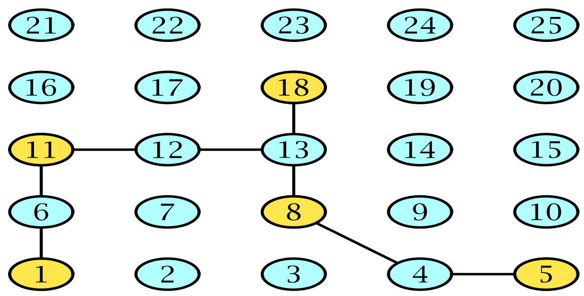

Abbildung 2.12: Lösung mit DNIH(P)

Wir betrachten jetzt nochmal das Beispiel aus dem Abschnitt 2.2 über die SP-Heuristik; die DNI-Heuristik liefert die in Abbildung 2.11 dargestellte Näherungslösung, die ein Gewicht von 8.8 aufweist. Natürlich kann man auch hier die beiden Schritte 4 und 5 als Nachbehandlung einsetzen; der dadurch erweiterte Algorithmus werde DNIH(P) bzw. DNIH(K) genannt, je nachdem, ob der Algorithmus von Prim oder der von Kruskal zum Einsatz kommt. Man erhält tatsächlich eine bessere Lösung, die nur ein Gewicht von 8.4 besitzt (siehe Abbildung 2.12).

Kapitel 3

Implementierung der Heuristiken

In diesem Kapitel werden zunächst mehrere Datenstrukturen und Algorithmen vorgestellt, die dazu benötigt werden, die Heuristiken aus Kapitel 2 zu implementieren. Dabei wird klassenorientiert vorgegangen, d.h. jeder Algorithmus wird innerhalb einer Klasse gekapselt. Dadurch kann man, wenn man nur die Deklaration der Klasse kennt, diese leicht wiederverwenden. Es wird versucht, den Leser nicht mit allzuviel Quelltext zu quälen, und deshalb werden wir uns bei allgemein bekannten Algorithmen darauf beschränken, die Klassendeklarationen zusammen mit den wichtigsten Details der Realisierung zu erläutern. Bemerkungen zur Komplexität dürfen freilich nicht fehlen, um die angegebenen Komplexitäten der Heuristiken für das Steiner-Problem begründen zu können.

3.1 Allgemeine Datenstrukturen

3.1.1 Union-Find-Algorithmus

In diesem Unterabschnitt soll ein sogenannter Union-Find-Algorithmus abgehandelt werden, der es uns ermöglicht, während des Aufbaus der Graphen T_i innerhalb von PD-Heuristiken über die Zusammenhangskomponenten Buch zu führen. Im Algorithmus für minimale aufspannende Bäume von Kruskal wird diese Datenstruktur ebenfalls Anwendung finden. Ihre besondere Eigenschaft ist, daß die Abfrage, ob sich zwei Knoten in derselben Zusammenhangskomponente von T_i befinden (`find`-Operator), und die Eingabe neuer (Pfad-)Informationen über den weiteren Aufbau der Näherungslösung (Übergang zu T_{i+1}, `union`-Operator) gemischt werden können.

Die Klasse, die hier die Union-Find-Struktur aufnehmen wird, bekommt den Namen `TEQ` (weil sie Information über die Äquivalenzklassen, engl. „equivalence classes" liefert); sie wurde nach Anregungen aus dem Buch von Sedgewick [Sed92] realisiert und hat in der Sprache C++ die in Abbildung 3.1 dargestellte Deklaration. Wir gehen im weiteren davon aus, daß

```
class TEQ
    {
    private:
        int size;
        int *dad;
        // (...)
    public:
        TEQ();
        TEQ(int asize);
        ~TEQ();

        void reset(void);
        void reset(int asize);
        void clear(void);
        int  clear(const int *preset);

        int unio(int v, int w);
        int find(int v, int w);

        friend ostream &operator<<(ostream &, const TEQ &);
        friend istream &operator>>(istream &, TEQ &);
    };
```

Abbildung 3.1: Deklaration der Klasse `TEQ`

Äquivalenzrelationen auf einer Menge $N = \{1, \ldots, n\}$ von `TEQ` verwaltet werden. Die Elementfunktionen haben im einzelnen folgende Aufgaben:

`TEQ()` und `void reset(void)`
: Stellen eine leere Struktur ohne Funktionalität bereit.

`TEQ(int asize)` und `void reset(int asize)`
: Bereiten eine Struktur der Größe $n =$ `asize` vor, in der jedes Element eine eigene Äquivalenzklasse repräsentiert.

`void clear(void)`
: Löscht die Informationen in der Struktur. Jedes Element bildet wieder seine eigene Äquivalenzklasse.

`int clear(const int *preset)`
: Löscht die Informationen in der Struktur. Jedes Element `i` des übergebenen Feldes `preset` mit Wert 1 und alle anderen Elemente zusammen bilden jeweils eine Äquivalenzklasse. Rückgabewert ist die Anzahl der Stellen in `preset` mit Wert 1.

`int unio(int v, int w)`
: Macht `v` und `w` äquivalent. Gibt 1 zurück, falls die beiden Elemente vorher nicht derselben Klasse angehörten, sonst 0.

`int find(int v, int w)`
: Gibt 1 zurück, falls `v` und `w` nicht äquivalent sind, sonst 0.

Zur Implementierung faßt man die Information über die Äquivalenzklassen in N in einem gerichteten Graphen $D = (N, A)$ mit der Eigenschaft $\gamma_D^+(v) \leqslant 1$ für alle $v \in N$ zusammen. Wegen letzterer Bedingung kann man diesen Graphen in einem Array `TEQ::dad` speichern, für welches gilt:

$$\gamma_D^+(v) = 1 \Leftrightarrow \texttt{dad[}v\texttt{]} = w, w \in \mathrm{adj}_D^+(v)$$
$$\gamma_D^+(v) = 0 \Leftrightarrow \texttt{dad[}v\texttt{]} = -\left|\{w | w \xrightarrow[D]{*} v\}\right| \leqslant 0 \,.$$

Aufgrund der Speicherung in einem Array der Länge $|N|$ hat man einen Speicherplatzbedarf von $O(|N|)$ für `TEQ`.

Es seien nun $v, w \in N$. Dann wird während einer `find(v,w)`-Aktion getestet, ob die folgenden beiden (eindeutig bestimmten) Knoten v' und w' gleich sind, um herauszufinden, ob v und w zur selben Äquivalenzklasse gehören:

$$v' \in N \text{ mit } v \xrightarrow[D]{*} v' \text{ und } \gamma_D^+(v') = 0$$
$$w' \in N \text{ mit } w \xrightarrow[D]{*} w' \text{ und } \gamma_D^+(w') = 0 \,.$$

Außerdem wird innerhalb von `find` Pfadkompression („path compression“) angewandt, d.h. für alle Knoten $\hat{v}$ mit $v \xrightarrow[D]{*} \hat{v} \xrightarrow[D]{*} v'$ setzt man `dad[`$\hat{v}$`]:=`v'; genauso werden für alle $\hat{w}$ mit $w \xrightarrow[D]{*} \hat{w} \xrightarrow[D]{*} w'$ die zugehörigen Einträge im `dad`-Array geändert durch `dad[`$\hat{w}$`]:=`w'.

Das Verfahren der Pfadkompression wird ebenso bei jedem Aufruf der Funktion `unio(v,w)` angewandt; außerdem folgt man, falls sich v und w nicht in derselben Äquivalenzklasse befinden, dem Prinzip des Gewichtsausgleichs („weight balancing“), d.h. es wird stets die Klasse mit weniger Elementen an die größere angehängt:

$$\begin{array}{lll} \mathrm{dad}[v] < \mathrm{dad}[w]: & \mathrm{dad}[v] := \mathrm{dad}[v] + \mathrm{dad}[w] - 1; & \mathrm{dad}[w] := v \\ \mathrm{dad}[v] \geqslant \mathrm{dad}[w]: & \mathrm{dad}[w] := \mathrm{dad}[w] + \mathrm{dad}[v] - 1; & \mathrm{dad}[v] := w \end{array}$$

Man kann mit den beiden ebengenannten Maßnahmen für die Elementfunktionen `unio` und `find` die bestmögliche Komplexität erreichen, nämlich fast $O(1)$; Sedgewick verweist in dieser Frage auf eine Arbeit von Leeuwen und Tarjan [LeT86]. Wir werden später nicht die genaue Schranke für die Komplexität verwenden, sondern nur die sichere Abschätzung $O(\log |N|)$.

3.1.2 Priority-Queues

In nahezu allen Verfahren, die in dieser Arbeit vorkommen, müssen Pfade auf Graphen oder Tupel, die Pfade beschreiben, dynamisch ihrer Länge entsprechend sortiert werden. Das heißt, daß man das kleinste Element einer Menge von Tupeln herausnimmt, der Menge neue Elemente hinzufügt, wieder ein kleinstes Element sucht usw. Eine dieser Anforderung angemessene Datenstruktur sind Priority-Queues, wie sie etwa Sedgewick beschreibt.

Wir werden in diesem Unterabschnitt zwei Klassen-Templates (engl. für Muster, Schablone) kennenlernen, die solche Datenstrukturen realisieren helfen. Das erste Template heißt `TPQ<T>`, und es ist in der Lage, Priority-Queues für von `TPQItem` abgeleitete Klassen zu unterhalten; diese Klassen stellen ihrerseits Tupel der Form $q = (p, \dots)$ dar, wobei $p \in \mathbb{R}$ die dem Tupel q zugehörige Priorität sei. Sie wird dazu verwandt, die Tupel in der Schlange anzuordnen.

Der Begriff Template, der erst seit etwa 1993 in Compilern für die Sprache C++ verfügbar ist, bedarf einer kurzen Erläuterung. Eine solche Deklaration wird hier verwendet, weil der Typ <T> der Daten, welche mit Hilfe von `TPQ<T>` verwaltet werden sollen, noch nicht bekannt ist. Vielmehr kennt man den „kleinsten gemeinsamen Nenner“, nämlich das Vorhandensein einer Priorität p. Vererbt man nun etwa durch die folgende Deklaration die Eigenschaften der Klasse `TPQItem` auf eine Klasse mit irgendwelchen zusätzlichen Möglichkeiten und Namen `TPQMyItem`,

```
class TPQMyItem : public TPQItem
    {      (...)      };
```

so veranlaßt die folgende Deklaration

```
TPQ<TPQMyItem> a_priority_queue;
```

den C++-Übersetzer, die Instanz `a_priority_queue` einer Klasse zu erzeugen, welche eine Priority-Queue mit Elementen des Typs `TPQMyItem` zu verwalten imstande ist. Man bezeichnet Templates auch als Typ-parametrisierte oder generische Klassen und spielt damit auf ihre Eigenschaft an, eine Familie verwandter Datentypen zu spezifizieren. Für eine genaue Darstellung sei auf das Buch von Stroustrup [Str92] verwiesen.

Anstelle von `TPQ<T>` werden wir im folgenden auf die Klassenschablone `TPQKey<T>` näher eingehen, die die Funktionalität von `TPQ<T>` erweitert, und jeweils die Einschränkungen darstellen, denen `TPQ<T>` im Vergleich zu `TPQKey<T>` unterliegt. Letztere Schablone soll (mit $N = \{1, \dots, n\}$) eine Menge der Form

$$Q \subseteq \{\, (p, k, \dots) \mid p \in \mathbb{R}, k \in \mathbb{N} \,\} \text{ mit}$$
$$\left| \{\, (p, k, \dots) \mid k = k_0 \,\} \right| \leqslant 1, \quad \forall k_0 \in N$$

verwalten können. Die $p \in \mathbb{R}$ dienen wieder der Anordnung der Tupel in Q, die $k \in N$ sind Schlüssel, die die Tupel eindeutig identifizieren; sie könnten z.B. dazu dienen, auf Knoten in einem Graphen oder in ein externes Feld zu verweisen. Wir nehmen weiter an, daß die Tupel $q \in Q$ durch Instanzen von `TPQKeyItem` oder davon abgeleitete Klassen dargestellt werden; die Funktionalität dieser Klasse beschränkt sich, wie in Abbildung 3.2 auf Seite 59 zu sehen, im wesentlichen auf die Bereitstellung von Vergleichsoperatoren für die Prioritäten und Schlüssel. Die Deklaration der Schablone `TPQKey` ist in Abbildung 3.3 dargestellt (Erweiterungen im Vergleich zu `TPQ` sind durch „(+)“, Auslassungen durch „(...)“ markiert), eine Erläuterung der enthaltenen Methoden folgt:

`TPQKey()` und `void reset(void)`
: Stellen eine leere Struktur ohne Funktionalität bereit.

`TPQKey(int amn)` und `void reset(int amn)`
: Bereiten eine Struktur der Größe $n =$ `amn` zur Aufnahme von maximal n Instanzen des Typs <T> vor.

```
class TPQKeyItem
    {
    private:
        FPTYPE  pr;
        int     ky;
    public:
        TPQKeyItem();
        TPQKeyItem(FPTYPE apr, int aky);
        virtual ~TPQKeyItem();

        FPTYPE  getpr(void) const   {return pr; }
        FPTYPE &getpr(void)         {return pr; }
        int     getky(void) const   {return ky; }
        int    &getky(void)         {return ky; }

        int operator< (const TPQKeyItem &q);
        int operator<=(const TPQKeyItem &q);
        int operator> (const TPQKeyItem &q);
        int operator>=(const TPQKeyItem &q);
        int operator< (const FPTYPE &q);
        int operator<=(const FPTYPE &q);
        int operator> (const FPTYPE &q);
        int operator>=(const FPTYPE &q);

        int operator==(const TPQKeyItem &q);
        int operator!=(const TPQKeyItem &q);

        friend ostream &operator<<(ostream&, const TPQKeyI-
tem& );
        friend istream &operator>>(istream&, TPQKeyItem&);
    };
```

Abbildung 3.2: Deklaration der Klasse `TPQKeyItem`

`int isempty(void) const` und `int isfull(void) const`
: Liefert 1, falls die Queue leer ist, sonst 0, und umgekehrt.

`int insert(const T &item)`
: Fügt die durch `item` referenzierte Instanz der Klasse <T> in die Queue ein. Liefert -1, falls die Queue schon voll ist, sonst 1, da eine Änderung an der Datenstruktur ausgeführt wurde.

`int change(const T &item)`
: Fügt die durch `item` referenzierte Instanz der Klasse <T> in die Queue ein bzw. ändert eine schon vorhandene Instanz mit demselben Schlüssel auf die Werte von `item`. Liefert -1 bei falschem Schlüsselwert, 0, falls keine Prioritätsänderung eintrat oder 1 sonst.

`int update(const T &item)`
: Fügt die durch `item` referenzierte Instanz der Klasse <T> in die Queue

```
template <class T> class TPQKey
    {
    private:
        int n, mn, *p;
        T *a;
    public:
        TPQKey();
        TPQKey(int amn);
        ~TPQKey();

        void reset(void);
        void reset(int amn);
        int isempty(void) const;
        int isfull(void)  const;
        int getnum(void)  const;

        int insert(const T &item);
        int change(const T &item);      //  (+)
        int update(const T &item);      //  (+)
        int downdate(const T &item);    //  (+)
        int exists(int ak);             //  (+)
        int exists(T &item);            //  (+)
        int remove(int ak);             //  (+)
        int remove(T &item);

        void clear(void);

        friend ostream &operator<<(ostream &, const TPQKey &);
        friend istream &operator>>(istream &, TPQKey &);
    };
```

Abbildung 3.3: Deklaration des Templates `TPQKey`

ein bzw. ändert eine schon vorhandene Instanz mit demselben Schlüssel auf die Werte von `item`, falls die Priorität von `item` kleiner ist als die der schon enthaltenen Instanz. Rückgabewerte wie bei `change`.

`int downdate(const T &item)`
Fügt die durch `item` referenzierte Instanz der Klasse <T> in die Queue ein bzw. ändert eine schon vorhandene Instanz mit demselben Schlüssel auf die Werte von `item`, falls die Priorität von `item` größer ist als die der schon enthaltenen Instanz. Rückgabewerte wie bei `change`.

`int exists(int ak)`
Liefert 0, falls keine Instanz von <T> mit Schlüssel `ak` in der Queue vorhanden ist, ansonsten einen Wert größer 0.

`int exists(T &item)`
Liefert 0, falls keine Instanz von <T> mit gleichem Schlüssel wie `item`

in der Queue vorhanden ist. Ansonsten wird ein Wert größer 0 geliefert und die entsprechende Instanz in `item` kopiert.

`int remove(int ak)`
Löscht eine Instanz von <T> mit Schlüssel gleich `ak` aus der Queue. Rückgabewerte wie bei `change`.

`int remove(T &item)`
Löscht das Element mit kleinstem Schlüssel und liefert es in `item`. Gibt -1 im Falle einer schon leeren Queue zurück, sonst 0.

`void clear(void)`
Leert die Queue durch Löschen aller Einträge.

Die Implementierung der Templates `TPQKey` und `TPQ` erfolgt mit Hilfe eines Heaps `a`, in dem Tupel des Typs <T> angelegt werden. Das Array `a` wird zu einem Heap durch die folgenden Eigenschaften, wenn `n` die Anzahl der gerade im Array abgelegten Tupel bezeichne:

$$\begin{array}{ll} \mathtt{a}[j] \leqslant \mathtt{a}[2j] & \forall 0 < 2j \leqslant \mathtt{n} \\ \mathtt{a}[j] \leqslant \mathtt{a}[2j+1] & \forall 0 < 2j+1 \leqslant \mathtt{n} \end{array}$$

Weil man mit Hilfe von Methoden wie etwa `change` oder `remove` des Templates `TPQKey` auch direkt (d.h. über den eindeutigen Schlüssel) auf Tupel zugreifen will, benötigt man dort zusätzlich ein Positionsfeld `p`, das jedem Schlüssel die aktuelle Position des zugehörigen Tupels im Heap zuordnet. Sowohl für `TPQ` als auch für `TPQKey` erhält man einen Speicherplatzbedarf proportional zu $O(|N|)$.

Jede Aktion, die Tupel in den Heap einfügt, daraus löscht oder die Priorität eines Tupel im Heap ändert, muß gefolgt werden von der Ausführung von Funktionen, welche die Heap-Eigenschaft des Arrays `a` durch Umordnen der Elemente gegebenenfalls wiederherstellen. Diese Funktionen sind als `private` deklariert; sie werden automatisch von denjenigen Elementfunktionen, die die Heap-Struktur zerstören können, aufgerufen. Ihre Komplexität von $O(\log|N|)$ bestimmt die Komplexitäten aller nichttrivialen Elementfunktionen der besprochenen Templates; daher haben die Methoden `insert`, `change`, `update`, `downdate` und `remove` selbst auch die Komplexität $O(\log|N|)$.

3.2 Graphentheoretische Algorithmen

In diesem Abschnitt werden die Klassen bzw. Templates `TEQ`, `TPQKey<T>` und `TPQKeyItem` dafür verwendet, drei bekannte Algorithmen aus der Graphentheorie in Klassen zu fassen.

3.2.1 Algorithmus von Dijkstra

Die Klasse `TDijkstra` wird alle kürzesten Pfade von einem Knoten $v \in V$ zu allen anderen Knoten in $V \setminus \{v\}$ in einem Graphen $G = (V, E, d, g)$ liefern, indem sie den Algorithmus von Dijkstra in der von Sedgewick in [Sed92]

```
class TDijkstra
    {
    private:
        FPTYPE *val;
        int *dad;
        TPQKey<TPQKeyItem> pq;
        const TGr *gr;
        // (...)
    public:
        TDijkstra();
        TDijkstra(const TGr &g);
        ~TDijkstra();

        void reset(void);
        void reset(const TGr &g);

        int SPT  (int v, TGr &spt);
        int SPT  (int v, TGr &spt, const TGr &g);
        int SPTnw(int v, TGr &spt);
        int SPTnw(int v, TGr &spt, const TGr &g);
    };
```

Abbildung 3.4: Deklaration der Klasse `TDijkstra`

auf Seite 461ff. dargestellten Form implementiert. Der Speicherbedarf der Klasse `TDijkstra` ist proportional zu $|V|$, denn es wird eine „Priority-First-Search“ (siehe die Erläuterungen von Sedgewick zu diesem Begriff) auf einer Priority-Queue der Länge $|V|$ mit Hilfe zweier Arrays, die auch jeweils $|V|$ Elemente haben, ausgeführt. Die überladenen Elementfunktionen `SPT(...)` und `SPTnw(...)` implementieren den Algorithmus mit einer Komplexität von $O\big((|E|+|V|)\log|V|\big)$, wobei erstere kürzeste Pfade auf $(V,E,d,0)$ (Knotengewichte bleiben unbeachtet) , letztere kürzeste Pfade auf (V,E,d,g) berechnet. Den Interface-Teil der Klasse entnimmt man Abbildung 3.4; die Elementfunktionen haben im einzelnen folgende Aufgaben und Parameter:

`TDijkstra()` und `void reset(void)`
: Stellen eine leere Struktur ohne Funktionalität bereit.

`TDijkstra(const TGr &g)` und `void reset(const TGr &g)`
: Bereiten eine Instanz der Klasse dafür vor, das Kürzeste-Pfade-Problem auf dem Graphen, der durch `g` referenziert wird, zu lösen.

`int SPT(int v, TGr &spt)`
: Löst das Problem kürzester Pfade von `v` zu allen anderen Knoten des Graphen $(V,E,d,0)$, der im Konstruktor oder mit `reset` spezifiziert wurde, und speichert einen „Shortest Paths Tree“ in der durch `spt` referenzierten Instanz der Klasse `TGr`. Der Rückgabewert ist -1, falls ein Fehler auftritt (wenn `v` $\notin \{1,\ldots,|V|\}$ oder vorher kein Graph festgelegt wurde, auf dem das Problem gelöst werden soll), und 0 sonst.

`int SPT(int v, TGr &spt, const TGr &g)`

Verhält sich wie die vorhergehende Funktion, nur daß der Problemgraph `G` (neu) definiert wird; der Aufruf dieser Methode für eine Instanz `dij` ist also gleichbedeutend mit:

```
dij.reset(g); dij.SPT(v,spt);
```

`int SPTnw(int v, TGr &spt)`

Entspricht `SPT(int v, TGr &spt)` mit dem Unterschied, daß hier die Knotengewichte beachtet werden, d.h. (V, E, d, g) zugrundeliegt.

`int SPTnw(int v, TGr &spt, const TGr &g)`

Entspricht `SPT(int v, TGr &spt, const TGr &g)` mit dem Unterschied, daß hier die Knotengewichte beachtet werden.

Nach Durchführung der Methode `SPTnw(v,spt)` bzw. ihrer überladenen Variante befindet sich in einer durch `spt` referenzierten Instanz der Klasse `TGr` ein sogenannter „Shortest Paths Tree“ (SP-Baum); dies ist ein gerichteter Graph $D = (N, A, d')$ mit zusätzlicher Bewertung $g : N \to \mathbb{R}_{\geqslant 0}$ auf den Knoten, für den neben $N = V$ folgendes gilt:

$$\begin{aligned}
\operatorname{adj}_D^+(w) &= 1, && \forall w \in V\setminus\{v\}, \\
\operatorname{adj}_D^+(v) &= 0 && \\
d'_{(w,w')} &= d_{\min(G)}(w, v), && \forall \{w, w'\} \in A \\
g'_w &= d_{\min(G)}(w, v), && \forall w \in V\setminus\{v\} \,.
\end{aligned}$$

Weiter läßt sich für jedes $w \in V\setminus\{v\}$ ein $n_w \in \mathbb{N}$ wählen, sodaß ein Pfad P_w existiert mit den folgenden Eigenschaften:

$$\begin{aligned}
P_w = (w_1, \ldots, w_{n_w}) &\in \mathcal{P}_D(w, v) && \\
d(P_w) + g(P_w) &= d_{\min(G)}(w, v) && \\
\{w_i, w_{i+1}\} &\in A, && \forall 1 \leqslant i < n_w \,.
\end{aligned}$$

Damit lassen sich aus `spt` alle kürzesten Pfade mit v als Anfangsknoten rekonstruieren.

Die Funktion `SPT(v,spt)` und ihre überladene Variante liefern Ergebnisse vom gleichen Format für den zugrundeliegenden Graphen $(V, E, d, 0)$.

3.2.2 Algorithmus von Floyd-Warshall

Dieser Unterabschnitt behandelt die Klasse `TFlWa`, die den Algorithmus von Floyd-Warshall zur Lösung des Kürzeste-Pfade-Problems zwischen allen Knoten eines Graphen $G = (V, E, d, g)$, realisiert; Einzelheiten über den Algorithmus, dessen Speicherplatzbedarf proportional zu $|V|^2$ ist und dessen Komplexität $O(|V|^3)$ beträgt, findet der Leser bei Sedgewick auf den Seiten 476ff.

Man betrachte die Deklaration der Klasse `TFlWa` in Abbildung 3.5; die einzelnen Elementfunktionen haben die folgenden Bedeutungen.

`TFlwa()` und `void reset(void)`

Stellen eine leere Instanz ohne Funktionalität bereit.

```
class TFlWa
    {
    private:
        // (...)
    public:
        TFlWa();
        ~TFlWa();

        void reset(void);
        void reset(const TGr &g);
        void resetnw(const TGr &g);

        FPTYPE SPL(int i, int j) const;
        int    SP (int i, int j) const;

        // (...)
    };
```

Abbildung 3.5: Deklaration der Klasse `TFlWa`

`void reset(const TGr &g)`
: Führt den Algorithmus von Floyd-Warshall für den durch `g` spezifizierten Graphen $(V, E, d, 0)$, d.h. ohne Beachtung der Knotengewichte, durch.

`void resetnw(const TGr &g)`
: Führt den Algorithmus von Floyd-Warshall für den durch `g` spezifizierten Graphen (V, E, d, g), d.h. unter Beachtung der Knotengewichte, durch.

`FPTYPE SPL(int i, int j) const`
: Liefert (nach Durchführung einer der `reset(...)`-Funktionen) die Länge eines kürzesten Pfades zwischen den Knoten `i` und `j`.

`int SP(int i, int j) const`
: Liefert (nach Durchführung einer der `reset(...)`-Funktionen) einen Nachbarknoten von `i`, der auf einem kürzesten Pfad von `i` nach `j` liegt. Mit Hilfe dieser Funktion können kürzeste Pfade iterativ rekonstruiert werden.

3.2.3 Algorithmen von Prim und Kruskal

Die Klassen `TPrim` und `TKruskal` werden wir dafür einsetzen, einen minimalen aufspannenden Baum eines bewerteten Graphen $G = (V, E, d, g)$ zu bestimmen. Wie im vorangehenden Unterabschnitt werden auch hier Implementierungen verwendet, welche Sedgewick im oben schon angesprochenen Buch auf den Seiten 454ff. (für die Methode nach Prim) bzw. 458ff. (für die Methode nach Kruskal) darstellt.

Die Realisierung von Prims Algorithmus erfolgt mit Hilfe von „Priority First Search“ auf einer Instanz der Klasse `TPQKey<TPQKeyItem>` (siehe da-

```
class TPrim
    {
    private:
        FPTYPE *val;
        int *dad;
        TPQKey<TPQKeyItem> pq;
        const TGr *gr;
        // (...)
    public:
        TPrim();
        TPrim(const TGr &g);
        ~TPrim();

        void reset(void);
        void reset(const TGr &g);

        int MST(int v, TGr &mst);
        int MST(int v, TGr &mst, const TGr &g);
    };
```

Abbildung 3.6: Deklaration der Klasse `TPrim`

zu Unterabschnitt 3.1.2) der Größe $|V|$ unter Zuhilfenahme zweier Arrays von jeweils derselben Länge. Der Speicherbedarf von `TPrim` beträgt also $O(|V|)$; die Komplexität der überladenen Methode `TPrim::MST(...)` wird von Sedgewick angegeben zu $O((|E| + |V|)\log|V|))$.

Kruskals Algorithmus benutzt eine Priority Queue desselben Typs wie die, welche in `TPrim` vorliegt; sie wird hier jedoch zu Speicherung von Kantenlängen benutzt und hat deshalb, genauso wie zwei zusätzlich eingesetzte Arrays, einen zu $|E|$ proportionalen Bedarf an Speicher. Außerdem wird ein Union Find-Algorithmus (siehe Seite 55) für die Knoten aus G benützt. Man hat für `TKruskal` daher insgesamt einen Speicherbedarf von $O(|V| + |E|)$. Die Komplexität der Methoden `TKruskal::MST(...)` beträgt, so Sedgewick auf Seite 460, $O(|E|\log|E|)$.

Den Darstellungen 3.6 und 3.7 entnimmt man, daß die beiden Klassen dieselbe Struktur wie schon `TDijkstra` besitzen; im folgenden werden die Elementfunktionen erklärt, zunächst die der Klasse `TPrim` :

`TPrim()` und `void reset(void)`
: Stellen eine leere Instanz ohne Funktionalität bereit.

`TPrim(const TGr &g)` und `void reset(const TGr &g)`
: Bereiten eine Instanz von `TPrim` darauf vor, einen minimalen aufspannenden Baum auf dem durch `g` referenzierten Graphen zu lösen.

`int MST(int v, TGr &mst)`
: Sucht nach einem minimalen aufspannenden Baum auf dem Graphen, der bei Aufruf des Konstruktors oder mit `reset` spezifiziert wurde. Der Algorithmus von Prim bedingt die Angabe eines `v`, bei dem die „Priority First Search“ startet. Das Ergebnis wird in der durch `mst`

```
class TKruskal
    {
    private:
        TPQKey<TPQKeyItem> pq;
        TEQ eq;
        int *v, *w;
        const TGr *gr;
        // (...)
    public:
        TKruskal();
        TKruskal(const TGr &g);
        ~TKruskal() {}

        void reset(void);
        void reset(const TGr &g);

        int MST(TGr &mst);
        int MST(TGr &mst, const TGr &g);
    };
```

Abbildung 3.7: Deklaration der Klasse `TKruskal`

bezeichneten Instanz der Klasse `TGr` gespeichert. Der Rückgabewert ist -1, falls ein Fehler auftritt (wenn $\mathtt{v} \notin \{1, \ldots, |V|\}$ oder vorher kein Graph festgelegt wurde, auf dem das Problem gelöst werden soll), und 0 sonst.

`int MST(int v, TGr &mst, const TGr &g)`
Verhält sich wie die vorhergehende Funktion, nur daß der Problemgraph `G` (neu) definiert wird; der Aufruf dieser Methode für eine Instanz `prim` ist also gleichbedeutend mit:

```
prim.reset(g); prim.MST(v,mst);
```

Jetzt folgt eine Erklärung der Methoden in `TKruskal`:

`TKruskal()` und `void reset(void)`
Stellen eine leere Instanz ohne Funktionalität bereit.

`TKruskal(const TGr &g)` und `void reset(const TGr &g)`
Bereiten eine Instanz von `TKruskal` darauf vor, einen minimalen aufspannenden Baum auf dem durch `g` referenzierten Graphen zu lösen.

`int MST(TGr &mst)`
Sucht nach einem minimalen aufspannenden Baum auf dem Graphen, der bei Aufruf des Konstruktors oder mit `reset` spezifiziert wurde. Das Ergebnis wird in der durch `mst` bezeichneten Instanz der Klasse `TGr` gespeichert. Der Rückgabewert ist -1, falls ein Fehler auftritt (wenn vorher kein Graph festgelegt wurde, auf dem das Problem gelöst werden soll), und 0 sonst.

`int MST(TGr &mst, const TGr &g)`

Verhält sich wie die vorhergehende Funktion, nur daß der Problemgraph `G` (neu) definiert wird; der Aufruf dieser Methode für eine Instanz `kruskal` ist also gleichbedeutend mit:

```
kruskal.reset(g); kruskal.MST(mst);
```

Wir gehen davon aus, daß mit Hilfe der gerade besprochenen Klassen ein minimaler aufspannender Baum zum Graphen $G = (V, E, d, g)$ berechnet werden sollte. Interpretiert man den an der durch `mst` definierten Stelle als Ergebnis gespeicherten gerichteten Graphen gemäß Gleichung (1.3) (siehe Seite 14) als Graphen $T = (V, E', d, g)$, so ist T gerade ein minimaler aufspannender Baum von G.

3.3 Heuristiken für das Steiner-Problem

In diesem Abschnitt werden die Klassen `TDNH` und `TSPH` erklärt, die zur Implementierung der Distance Network-Heuristik mit impliziter Pfadberechnung bzw. der verschiedenen Versionen der Shortest Path-Heuristik dienen. Außerdem werden Funktionen erläutert, mit deren Hilfe man die Schritte 4 und 5 von PD-Heuristiken durchführen kann.

Zunächst werden wir auf die Gemeinsamkeiten der Klassen `TDNH` und `TSPH` eingehen. Das Problem USTP'$_{G,Z}$, das es ja näherungsweise zu lösen gilt, wird spezifiziert durch

- einen zusammenhängenden, ungerichteten Graphen $G = (V, E, d, g)$, der gemäß (1.2) in einen Digraphen transformiert und in einer Instanz `g` der Klasse `TGr` abgelegt worden ist, und
- ein Feld `sn` der Länge $|V| + 1$, das die Information über die Menge $Z \subseteq V$ enthält. Für die Feldelemente von `sn` gilt:

$$\begin{aligned} \texttt{sn[0]} &= |Z| \\ \texttt{sn}[i] &= 1 \Leftrightarrow i \in Z, \quad \forall i \in V \ . \end{aligned}$$

Wir gehen dabei wie immer in dieser Arbeit, wenn die algorithmische Behandlung von Graphen zur Debatte steht, davon aus, daß ohne Einschränkung $V = \{1, \dots, n\}$ mit einem $n \in \mathbb{N}$ gilt.

Als Ergebnis liefern die Heuristiken einen Digraphen $D = (N, A, \hat{d})$ mit $N = V$ und der zusätzlichen Knotenbewertung g, in welchem die errechnete Näherungslösung eingebettet ist. Um diese zu extrahieren, interpretiere man den gerichteten Graphen D als erstes via (1.3) wieder als ungerichteten Graphen $G' = (V, E', d', g)$; es gilt dabei $d|_{E'} = d'$. Als zweites „lösche" man nun noch alle Steiner- Knoten aus G', welche in der Näherungslösung nicht benützt werden, d.h. man setze

$$\begin{aligned} T &:= (V', E', d', g') \quad \text{mit} \\ V' &:= \{\, v \in V \mid \gamma_{G'}(v) > 0 \,\}, \\ g' &:= g|_{V'} \ . \end{aligned} \tag{3.1}$$

Dadurch erhält man die gesuchte Näherungslösung T.

Funktionsname	Komplexität
SPHz	$O(\|V\|\|Z\| \log \|Z\|)$
SPHzz	$O(\|V\|\|Z\| \log \|Z\|)$
SPHZ	$O(\|V\|\|Z\|^2 \log \|Z\|)$
SPHV	$O(\|V\|^2\|Z\| \log \|Z\|)$
SPHzZ	$O(\|V\|\|Z\|^2 \log \|Z\|)$
SPHZZ	$O(\|V\|\|Z\|^3 \log \|Z\|)$

Tabelle 3.1: Komplexitäten der Methoden von TSPH

3.3.1 Klasse TSPH

Innerhalb der Klasse TSPH werden die Heuristiken aus Abschnitt 2.2 durch Methoden realisiert. Die Komplexitäten der einzelnen Funktionen wurden in jenem Abschnitt weiter vorn schon angegeben; in Tabelle 3.1 sind sie noch einmal aufgelistet. Bevor eine der Methoden durchlaufen wird, muß die Funktion `TDijkstra::spt` für jedes $z \in Z$ aufgerufen werden; dies geschieht bei der Übergabe der Problemdaten an die Klasse TSPH mit einer Gesamtkomplexität von $O((|V| + |E|) \log |V|)$. Alternativ dazu kann bei Übergabe der Problemdaten auch eine Referenz auf ein `TFlWa`-Objekt mitgeliefert werden, das schon alle Pfadinformationen des Problemgraphen enthält. Dann brauchen innerhalb von TSPH freilich keine kürzesten Pfade mehr errechnet werden.

Werden mehrere der in der Tabelle aufgeführten Funktionen für dasselbe Problem durchlaufen (z.B. zum Zwecke des Vergleichs), so können natürlich alle diese Methoden die einmal durch `TDijkstra` bzw. `TFlWa` gelieferten Pfadinformationen benutzen, ohne sie jeweils neu zu berechnen. Aus der Deklaration der Klasse TSPH ersieht man, daß klassenweit die Felder `dad`, `val` und `spt` benutzt werden. Alle drei haben eine Länge von $|Z|$ Elementen. In `spt` werden die vom Dijkstra-Algorithmus gelieferten Shortest Paths-Bäume, deren Größe je $O(|V|)$ beträgt, abgelegt. Da innerhalb von SPHz bzw. SPHzz als weiteres von der Problemgröße abhängiges Datum nur noch eine Priority Queue der Länge $O(|V|)$ benutzt wird, haben diese beiden Funktionen insgesamt eine Speicherintensität von $O(|V||Z|)$. Im Falle der Benutzung einer Instanz von `TFlWa` für die Pfadinformationen beträgt die Speicherintensität insgesamt $O(|V|^2)$.

Dieselben Werte gelten auch für alle iterierten Versionen; der anschließend mögliche Aufruf der Funktion `trimming` (siehe 3.3.3) vergrößert den Speicherbedarf um die für den Algorithmus von Kruskal bzw. Prim geltenden Terme.

Es bleibt noch, die Elementfunktionen im einzelnen zu erläutern:

`TSPH()` und `void reset(void)`
Stellen eine leere Instanz ohne Funktionalität bereit.

`TSPH(const TGr &g, const int *sn)` und
`void reset(const TGr &g, const int *sn)`
Bereiten eine Instanz von TSPH darauf vor, das durch `g` und `sn` gegebene Problem $\mathrm{USTP'}_{G,Z}$ näherungsweise zu lösen. Es erfolgen $|Z|$

```
class TSPH
    {
    private:
        int *dad;
        FPTYPE *val;
        TGr *spt;
        const TGr *gr;
        const int *sn;
        // (...)
    public:
        TSPH();
        TSPH(const TGr &g, const int *sn);
        TSPH(const TGr &g, const int *sn, const TFlWa &fw);
        ~TSPH();

        void reset(void);
        void reset(const TGr &g, const int *sn);
        void reset(const TGr &g, const int *sn,
                   const TFlWa &fw);

        FPTYPE  SPHz (int v, TGr &spht, int ftrim=0);
        FPTYPE  SPHzz(int v, int w, TGr &spht, int ftrim=0);

        FPTYPE  SPHZ (TGr &spht, int ftrim=0);
        FPTYPE  SPHV (TGr &spht, int ftrim=0);
        FPTYPE  SPHzZ(int v, TGr &spht, int ftrim=0);
        FPTYPE  SPHZZ(TGr &spht, int ftrim=0);
    };
```

Abbildung 3.8: Deklaration der Klasse `TSPH`

Aufrufe des Dijkstra-Algorithmus; die Ergebnisse kommen ins `spt`-Array.

`TSPH(const TGr &g, const int *sn, const TFlWa &fw)` und `void reset(const TGr &g, const int *sn, const TFlWa &fw)`

Bereiten eine Instanz von `TSPH` darauf vor, das durch `g` und `sn` gegebene Problem USTP'$_{G,Z}$ näherungsweise zu lösen. Die Referenz `fw` der Klasse `TFlWa` enthalte die Pfadinformationen des Graphen `g`.

`FPTYPE SPHz(int v, TGr &spht, int ftrim = 0)`

Wendet SPH-z mit Startknoten `v` an; wird für `ftrim` ein Wert $\neq 0$ angegeben, so ruft `SPHz` zusätzlich die `trimming`-Funktion mit dem Auswahlflag `ftrim` auf. Das Ergebnis kommt in die durch `spht` referenzierte Instanz von `TGr`. Funktionswert ist das Gewicht $d'(T) + g'(T)$ der Näherungslösung.

`FPTYPE SPHzz(int v, int w, TGr &spht, int ftrim = 0)`

Wendet SPH-zz mit den Startknoten `v` und `w` an; sonstige Funktionsweise wie bei `SPHz`.

FPTYPE SPHZ(TGr &spht, int ftrim = 0)
Iteriert SPHz(v,spht,ftrim) für alle $v \in Z$ und liefert die beste erhaltene Näherungslösung.

FPTYPE SPHV(TGr &spht, int ftrim = 0)
Iteriert SPHz(v,spht,ftrim) für alle $v \in V$ und liefert die beste erhaltene Näherungslösung.

FPTYPE SPHzZ(int v, TGr &spht, int ftrim = 0)
Iteriert SPHzz(v,w,spht,ftrim) für alle $w \in Z$ und liefert die beste erhaltene Näherungslösung.

FPTYPE SPHZZ(TGr &spht, int ftrim = 0)
Iteriert SPHzz(v,w,spht,ftrim) für alle $v, w \in Z$ und liefert die beste erhaltene Näherungslösung.

3.3.2 Klasse TDNIH

Man betrachte die zur Realisierung der Distance Network - Heuristik mit impliziter Pfadberechung dienende Klasse TDNIH in Abbildung 3.9. Ihr Speicheraufwand ist $O(|V| + |E|)$, weil

- die Felder le, pr und so jeweils $|V|$ Elemente aufnehmen,
- der Union Find-Algorithmus für jedes $v \in V$ einen Eintrag aufnimmt

und

- die Priority Queue pq maximal $|E|$ Einträge des Typs TPQItemDNIH aufzunehmen braucht (siehe Lemma 2.3.17). Die Klasse TPQItemDNIH ist Abkömmling von TPQItem; ihre Instanzen repräsentieren die innerhalb der DNI-Heuristik verwandten Tupel der Gestalt $(d, v, z, \bar{z}, \bar{v})$.

Die Schlüsselfunktion DNIH fordert darüberhinaus nur noch eine konstante Speichermenge. Die Methoden der Klasse TDNIH entsprechen dem nun schon gewohnten Muster und sind schnell erklärt.

TDNIH() und void reset(void)
Stellen eine leere Instanz ohne Funktionalität bereit.

TDNIH(const TGr &g, const int *sn) und
void reset(const TGr &g, const int *sn)
Bereiten eine Instanz von TDNIH darauf vor, das durch g und sn gegebene Problem USTP'$_{G,Z}$ näherungsweise zu lösen.

FPTYPE DNIH(TGr &dniht, int ftrim = 0)
Wendet die Algorithmen 6 und 7 an (siehe Seiten 43ff.). Wenn für ftrim ein Wert $\neq 0$ angegeben wird, dann ruft die Methode zusätzlich die trimming-Funktion mit dem Auswahlflag ftrim auf. Das Ergebnis kommt in die durch dniht referenzierte Instanz von TGr. Funktionswert ist das Gewicht $d'(T) + g'(T)$ der Näherungslösung.

```
class TDNIH
    {
    private:
        int *so, *pr;
        FPTYPE *le;
        TPQ<TPQItemDNIH> *pq;
        TEQ *eq;
        const TGr *gr;
        const int *sn;
        // (...)
    public:
        TDNIH(void);
        TDNIH(const TGr &g, const int *sn);
        ~TDNIH();

        void reset(void);
        void reset(const TGr &g, const int *sn);
        FPTYPE DNIH(TGr &dniht, int ftrim=0);
    };
```

Abbildung 3.9: Deklaration der Klasse `TDNIH`

3.3.3 Realisierung der PDH-Schritte 4 und 5

In den beiden vorangehenden Unterabschnitten wurde schon mehrmals die folgende Funktion erwähnt:

```
int trimming(
            TGr &st,
            const TGr &g,
            const int *sn,
            int ftrim
            );
```

Ihre Aufgabe besteht darin, einen von einer PD-Heuristik als Näherungslösung gelieferten Digraphen D (zur Interpretation dieses Digraphen als Näherungslösung siehe den Anfang von Abschnitt 3.3), welcher in `st` abgelegt sei, den Schritten 4 und 5 (siehe Seiten 26f.) zu unterziehen. Einige Autoren wie etwa Rayward-Smith und Clare ([RaC86]) nennen diese beiden Schritte „pruning" (engl. für abschneiden, zuschneiden). Die Parameter `g` und `sn` spezifizieren das von der Heuristik vorher bearbeitete Steiner-Problem; `ftrim` ist eine Flagge, die auch in den Parametersätzen der Heuristiken schon aufgetreten ist und von letzteren unverändert an `trimming` übergeben wird; sie steuert, welcher Algorithmus zur Generierung eines minimalen aufspannenden Baumes in PDH-Schritt 4 verwendet wird. Ist `ftrim` $= 1$, so kommt der Algorithmus von Kruskal zur Anwendung, in allen anderen Fällen wird der Algorithmus von Prim eingesetzt.

Wir betrachten die Komplexität der Funktion in jedem der drei wichtigen Teilschritte von `trimming`; $T = (V', E', d', g')$ sei die durch (3.1) gegebene, von einer Heuristik gelieferte Näherungslösung.

- Zunächst muß der durch V' induzierte Teilgraph $G'_{G,V'}$ von G (Definition 2.1.5) bestimmt werden. Weil für jede Kante in G entschieden wird, ob sie zu G' gehört oder nicht, hat dieser Teil eine Komplexität von $O(|E|)$.
- Die Bestimmung eines minimalen aufspannenden Baumes T' für $G'_{G,V'}$ hat, je nach eingesetztem Algorithmus (siehe Abschnitt 3.2.3), eine worst case-Komplexität von $O\big((|E| + |V|) \log |V|\big)$ bzw. $O\big(|E| \log |E|\big)$.
- Schließlich folgt in Schritt 5 das sukzessive Abschneiden von unnötigen Blättern aus T'; dieser Baum enthält schlimmstenfalls $|V| - 1$ Knoten; daher wird dieser Schritt in einer Zeit von $O\big(|V|\big)$ durchgeführt.

Insgesamt wird `trimming` also vom Zeitaufwand dominiert, der zur Berechnung der minimalen aufspannenden Bäume getrieben werden muß. Was die Speicher-Seite betrifft, werden nur zwei Graphen der Größe von G benützt.

Das von `trimming` gelieferte Ergebnis wird in `st` abgelegt und hat dasselbe Format wie die von einer PD-Heuristik gelieferten Näherungen. Insofern ist `trimming` den Heuristiken gegenüber völlig transparent.

Kapitel 4

Genetischer Algorithmus

Dieses Kapitel illustriert, wie man genetische Optimierungsalgorithmen zusammen mit den aus Kapitel 2 bekannten Heuristiken einsetzen kann, um die Ergebnisse, die etwa DNIH oder SPH-z liefern, den optimalen Werten weiter anzunähern. Im weiteren werden unter dem Begriff „genetische Algorithmen“ solche Suchmethoden zusammengefaßt, welche Holland in seinem Buch [Hol92] (in den Kapiteln 6 und 7) als „genetic plans“ bezeichnet. Eine im Verhältnis zu dieser ersten grundlegenden Monographie über genetische Algorithmen kürzer gefaßte Einführung sowie eine Reihe von Beispielproblemen, auf die diese Methoden schon angewandt worden sind, findet der Leser im Buch von Goldberg [Gol89].

Den Ansatz, welcher hier verfolgt wird, hat Esbensen in seinem Artikel [Esb95] im Zusammenhang mit dem klassischen Steiner-Problem auf Graphen dargestellt; der am Ende eingesetzte genetische Algorithmus wird sich aber in einem entscheidenden Punkt, nämlich der Art des Crossover-Operators, von Esbensens Methode unterscheiden. Bei der Wahl dieses Operators folgt der Autor den Verfassern von [Ran94], die einen genetischen Algorithmus verwenden, um Heuristiken zur (inkrementiellen) Partition von Graphen zu verbessern.

Die Grundidee des Einsatzes eines genetischen Algorithmus zusammen mit Heuristiken besteht darin, eines der dargestellten heuristischen Verfahren h statt nur auf das Grundproblem USTP'$_{G,Z}$ auf Ersatzprobleme der Form USTP'$_{G,Z\cup Z^+}$ mit einer Menge zusätzlicher spezieller Knoten $Z^+ \subseteq V\backslash Z$ anzuwenden. Die Bäume, welche die Heuristik h bei Anwendung auf USTP'$_{G,Z\cup Z^+}$ liefert, stellen natürlich erst recht zulässige Lösungen für das Problem USTP'$_{G,Z}$ dar; das Gewicht eines derartigen Baumes sei bezeichnet durch $\mathrm{res}_{h(G,Z\cup Z^+)}$. Dem genetischen Algorithmus fällt nun die Aufgabe zu, nach einem Minimum

$$\min\{\, \mathrm{res}_{h(G,Z\cup Z^+)} \mid Z^+ \subseteq V\backslash Z \,\}$$

zu suchen.

4.1 Repräsentierung der Ersatzprobleme

Wir nehmen wieder an, daß der dem Steiner-Problem zugrundeliegende Graph G von der Gestalt $G = (V, E, d, g)$ mit einer Menge $V = \{1, \dots, n\}$ von Knoten und $n \in \mathbb{N}$ sei. Ein Ersatzproblem wie oben mit $Z^+ \subseteq V \setminus Z$ wird dargestellt durch ein Tupel $g = (g_j)_{j \in V \setminus Z}$ mit $g_j \in \{0, 1\}$ für alle $j \in V \setminus Z$ und der charakterisierenden Eigenschaft:

$$g_j = 1 \Leftrightarrow j \in Z^+ \; .$$

Sind beispielsweise $V = \{1, \dots, 5\}$ und $Z = \{1, 3, 5\}$, so wird ein Ersatzproblem mit $Z^+ = \{2\}$ repräsentiert durch das Tupel $(1, 0)$.

Wir behalten die von der biologischen Genetik inspirierte Bezeichnungsweise Hollands bei und nennen deshalb das durch den Genotyp g repräsentierte Ersatzproblem USTP'$_{G, Z \cup Z^+}$ den zu g gehörigen Phänotyp. Die einzelnen Komponenten von g heißen Gene, das Tupel g als ganzes Chromosom. Für die Menge aller möglichen Genotypen schreiben wir:

$$\mathcal{G} = \left\{ \, (g_j)_{j \in V \setminus Z} \;\middle|\; g_j \in \{0, 1\} \, \right\} .$$

Wir identifizieren von nun an (wie auch Holland) Genotypen mit Phänotypen und schreiben deshalb kurz $h(g)$ statt $h(G, Z \cup Z^+)$ für einen von der Heuristik h gelieferten Baum, falls der Genotyp g das Ersatzproblem USTP'$_{G, Z \cup Z^+}$ repräsentiert.

4.2 Fitness - Funktion

Der genetische Algorithmus wird eine Folge $\big(G(t)\big)_{t \in \mathbb{N} \cup \{0\}}$ sogenannter Populationen erzeugen, die die Eigenschaft $G(t) \subseteq \mathcal{G}$ für alle $t \in \mathbb{N} \cup \{0\}$ besitzen; die Populationsgröße $\big|G(t)\big| = N \in \mathbb{N}$ wird konstant gehalten. Genetische Operatoren (Mutation und Crossover) steuern, abhängig von einer Wahrscheinlichkeitsverteilung $P(t)$ auf $G(t)$, den Übergang zur neuen Generation $G(t+1)$. Die Verteilung $P(t)$ ihrerseits basiert auf den Werten $\mathrm{res}_{h(g)}$ für die Genotypen $g \in G(t)$. Man setzt voraus, daß die Genotypen in $G(t) := \{g^{(t,1)}, \dots, g^{(t,N)}\}$ angeordnet seien, d.h. es gelte:

$$\mathrm{res}_{h\left(g^{(t,1)}\right)} \geqslant \mathrm{res}_{h\left(g^{(t,2)}\right)} \geqslant \cdots \geqslant \mathrm{res}_{h\left(g^{(t,N)}\right)}$$

$g^{(t,N)}$ ist also der „beste“ in der Population $G(t)$ vorkommende Genotyp; dieser soll gemäß dem Darwin'schen Prinzip des „survival of the fittest“ mit der größten Wahrscheinlichkeit unter allen Genotypen in $G(t)$ versehen werden, von genetischen Operatoren verarbeitet zu werden und somit „Nachkommen“ in der nächsten Generation zu erhalten. Die diskrete Verteilung $P(t) := (p_i)_{i=1,\dots,N}$, die in diesem Fall nicht von t abhängt, definiert man nun durch:

$$p_i := \frac{2(N-i)}{N(N-1)}, \quad \forall\, i = 1, \dots, N \; .$$

Es gilt:

$$\sum_{i=1}^{N} p_i = \frac{2}{N(N-1)} \sum_{i=0}^{N-1} i = 1 \; .$$

$P(t)$ legt also für jedes Individuum einer Population fest, mit welcher Wahrscheinlichkeit es Nachkommen in der Folgegeneration erhält. Diese Wahrscheinlichkeit hängt hier nicht direkt vom Wert der Heuristik $h(g)$ für jedes Individuum $g \in G(t)$ ab, sondern vielmehr vom Rang des Individuums relativ zu seinen Mitkonkurrenten in $G(t)$. Dieses Bewertungsverfahren, das „ranking“ genannt wird, hat sich im Vergleich zu anderen Methoden als sehr wirkungsvoll erwiesen ([Whi89], [GoD91]) und wird sowohl in [Ran94] als auch in [Esb95] angewandt. Esbensen verwendet dieselben Wahrscheinlichkeiten p_i wie oben angeführt, bei denen das beste Individuum einer Population im Vergleich zum Median die doppelte Wahrscheinlichkeit besitzt, Nachkommen in der Folgegeneration zu erzeugen.

4.3 Genetische Operatoren

Der hier implementierte genetische Algorithmus hat den folgenden (schematischen) Aufbau. Die einzelnen Operatoren werden unten genauer besprochen, die Klasse TGA, welche zur eigentlichen Implementierung dient, in einem späteren Abschnitt:

- Erzeugung einer zufälligen Startpopulation $G(0)$ der Größe N;
  Man setze $t := 0$.
- Solange das Abbruchkriterium nicht erfüllt ist,
  - Aktualisierung der DKNUX-Wahrscheinlichkeiten;
  - Erstellung von N neuen Genotypen mit Hilfe des Crossover-Operators (d.h. DKNUX) und Mutation;
  - Auswahl einer neuen Population $G(t+1)$ der Größe N aus der vorhergehenden Population und den neu erzeugten Genotypen;
  - man setze $t := t + 1$.
- Ausgabe der besten während der Schleifendurchläufe erzielten Näherungslösung.

4.3.1 Erzeugung einer Startpopulation

Hier wird zufällig eine wie in 4.2 angeordnete Population $G(0)$ erzeugt. Für die einzelnen Genotypen $g^{(0,i)} = (g_j^{(0,i)})_{j \in V \setminus Z}$ darin (bis auf einen) gilt:

$$g_j^{(0,i)} = \begin{cases} 1 & \text{mit Wahrscheinlichkeit } P_{\text{Gen1}} \\ 0 & \text{mit Wahrscheinlichkeit } 1 - P_{\text{Gen1}} \end{cases}$$

Die Wahrscheinlichkeit $P_{\text{Gen1}} \in [0,1]$ muß vorgegeben werden. Mindestens einer der Genotypen in der Startpopulation wird mit $(0, \ldots, 0)$ vorbesetzt (man nennt dies „seeding“); er repräsentiert dadurch das Ausgangsproblem USTP'$_{G,Z}$.

4.3.2 Berechnung der DKNUX-Wahrscheinlichkeiten

Der DKNUX-Operator, der im nächsten Unterabschnitt beschrieben wird, ist definiert mit Hilfe der Tupel $q^{(t)} = (q_j^{(t)})_{j \in V \setminus Z}$ von Wahrscheinlichkeiten, wobei $t \in \mathbb{N} \cup \{0\}$; es gilt:

$$q_j^{(t)} \in [q_{\min}, q_{\max}]$$

mit vogegebenen Parametern $0 \leqslant q_{\min} \leqslant q_{\max} \leqslant 1$ und einem $q' > 0$. Ein weiterer Parameter $N_{\mathrm{DKNUX}} \in \mathbb{N}$ steuert, wie oft obige Tupel neu berechnet werden. Für $t \neq 0 \pmod{N_{\mathrm{DKNUX}}}$ wird einfach $q^{(t+1)} := q^{(t)}$ gesetzt. Ansonsten wird $q^{(t+1)}$ wie folgt neu berechnet. Zunächst sei $g(t)$ für $t \in \mathbb{N} \cup \{0\}$ der beste bis zu diesem Schleifendurchlauf gefundene Genotyp, d.h. es ist

$$\mathrm{res}_{h(g(t))} \leqslant \mathrm{res}_{h(g)}, \quad \forall g \in \bigcup_{t'=0}^{t} G(t') \; .$$

Mit $h(g(t))$ bezeichnen wir einen (ungerichteten) Baum mit Knotenmenge $V(t) \subseteq V$, den die Heuristik h als Näherungslösung für das durch $g(t)$ repräsentierte Ersatzproblem liefert. Für $j \in V \setminus Z$ definieren wir:

$$\bar{q}_j^{(t+1)} := q_{\min} + q' \left| \{ v \in \mathrm{adj}_G(j) | v \in V(t) \} \right| \; .$$

$\bar{q}_j^{(t+1)}$ hängt also von der Anzahl der Nachbarknoten von j in G ab, die in der Näherungslösung $h(g(t))$ enthalten sind. Damit definiert man nun weiter das Tupel $q^{(t+1)}$ durch

$$q_j^{(t+1)} := \begin{cases} q_{\min} & \text{falls } 0 \leqslant \bar{q}_j^{(t+1)} < q_{\min}, \\ \bar{q}_j^{(t+1)} & \text{falls } q_{\min} \leqslant \bar{q}_j^{(t+1)} \leqslant q_{\max}, \\ q_{\max} & \text{falls } q_{\max} < \bar{q}_j^{(t+1)} \leqslant 1 \; . \end{cases}$$

4.3.3 DKNUX-Crossover-Operator

Wir beschreiben jetzt den wichtigsten der beiden eigentlichen genetischen Operatoren (neben der Mutation), die innerhalb dieses Algorithmus eingesetzt werden. Sein Name steht für „Dynamic Knowledge Based Non-Uniform-Crossover“ (siehe [Ran94]). Dieser Operator stellt insofern eine Fortentwicklung des „Uniform Crossover-Operators“ (UX) dar, als er Informationen über die von der Heuristik gelieferten Näherungslösung in den Ablauf des genetischen Algorithmus einfließen läßt und ihn damit steuert; dies geschieht mittels der in Unterabschnitt 4.3.2 erläuterten DKNUX-Wahrscheinlichkeiten.

Wir gehen davon aus, daß eine wie in Abschnitt 4.2 geordnete Population $G(t)$ und eine Verteilung $P(t)$ ($t \in \mathbb{N} \cup \{0\}$) auf den Individuen dieser Population gegeben seien. Der Operator DKNUX erzeugt eine Menge $\bar{G}(t) = \{\bar{g}^{(t,1)}, \ldots, \bar{g}^{(t,N)}\}$ von neuen Individuen auf die folgende Art und Weise, abhängig vom Parameter $P_{\mathrm{DKNUX}} \in [0, 1]$:

- Für $i = 1, \ldots, N$
  - Wähle zwei Genotypen $g^{(t,i_1)}$ $g^{(t,i_2)}$ aus $G(t)$ mit $i_1 \neq i_2$ zufällig gemäß der Verteilung $P(t)$.
  - Wähle $r \in [0,1]$ zufällig gleichverteilt und unterscheide: falls $r > P_{\text{DKNUX}}$, setze $\bar{g}^{(t,i)} := g^{(t,i_1)}$; ansonsten sei $\bar{g}^{(t,i)} := (\bar{g}_j^{(t,i)})_{j \in V \setminus Z}$ mit

$$\bar{g}_j^{(t,i)} := \begin{cases} g_j^{(t,i_1)} & \text{falls } g_j^{(t,i_1)} = g_j^{(t,i_2)} \\ 0 & \text{sonst mit Wahrscheinlichkeit } 1 - q_j^{(t)} \\ 1 & \text{sonst mit Wahrscheinlichkeit } q_j^{(t)} \end{cases}$$

Die Verwendung der DKNUX-Wahrscheinlichkeiten im Crossover-Operator bedeutet, daß ein Knoten $v \in V \setminus Z$ umso eher in ein vom genetischen Algorithmus getestetes Ersatzproblem, d.h. in die Menge Z^+, aufgenommen wird, je mehr seiner Nachbarknoten schon in einem Ersatzproblem, welches ein gutes Ergebnis liefert, enthalten sind. Beim oben schon angesprochenen UX-Operator, der auch schon zur Lösung graphentheoretischer Probleme eingesetzt wurde (siehe [Sys89]), fließt dagegen keine problemspezifische Information ein; der Vektor $q^{(t)}$ aus Unterabschnitt 4.3.2 wird für alle $t \in \mathbb{N} \cup \{0\}$ und alle $j \in V \setminus Z$ konstant durch $q_j^{(t)} := q$ definiert (mit einem $0 < q < 1$). Dann haben alle Knoten aus $V \setminus Z$, auch solche, die von guten Lösungen weit entfernt liegen, die gleiche Chance, in ein Ersatzproblem aufgenommen zu werden.

4.3.4 Mutationsoperator

Dieser Operator wirkt auf die vorher von DKNUX erzeugte Menge $\bar{G}(t)$. Er ändert zufällig unter Verwendung der Wahrscheinlichkeit $P_{\text{MUT}} \in [0,1]$ einzelne Gene der Genotypen in dieser Menge. Jedes Gen wird mit einer Wahrscheinlichkeit $1 - P_{\text{MUT}}$ in seinem Zustand belassen und entsprechend mit Wahrscheinlichkeit P_{MUT} von 1 auf 0 bzw. von 0 auf 1 „umgedreht". P_{MUT} ist meist sehr klein und liegt in einer Größenordnung von etwa 0.01 oder weniger.

4.3.5 Erzeugung der neuen Population

Dieser letzte Schritt in jedem Schleifendurchlauf des gentischen Algorithmus erzeugt aus der „alten" Population $G(t)$ und der Menge $\bar{G}(t)$, die von den Crossover- und Mutations-Operatoren erstellt wurde, eine neue Generation $G(t+1)$. Sie besteht aus denjenigen Individuen g der beiden ebengenannten Mengen, welche die besten Zielfunktionswerte liefern. Der Anteil der Individuen in $G(t+1)$, welche aus $G(t)$ stammen, wird vom Wert des Parameters $N_{\max} \in \{0, \ldots, N\}$ gesteuert; für $G(t+1)$ gilt stets:

$$|\{g \in G(t+1) \mid g \notin \bar{G}(t)\}| \leqslant N_{\max} .$$

Maximal $N_{\max}$ Individuen werden demgemäß aus der alten Population in die neue kopiert.

4.3.6 Komplexität der Operatoren

Der Speicherbedarf des hier dargestellten genetischen Algorithmus hängt von der Zahl der Individuen der Populationen und von deren Länge ab; er beträgt $O(|V \setminus Z| N)$. Hinzu kommt freilich noch der von der jeweils zur Bewertung der Individuen verwendeten Heuristik benutzte Speicher.

Die Zeit-Komplexität des Algorithmus wird von drei Faktoren festgelegt, nämlich der Komplexität der genetischen Operatoren, der Komplexität der zur Anordnung der Individuen verwendeten Funktionen und der Komplexität der eingesetzten Heuristik. Bei den Tests am Ende dieser Arbeit wird offenbar werden, daß die auf die Durchführung der Heuristiken verwendete Zeit die anderen beiden Einflußgrößen bei weitem dominiert; die gewählte Heuristik wird für jede Generation $G(t)$ im schlimmsten Falle (wenn also kein Individuum unverändert von $G(t)$ nach $\bar{G}(t)$ übernommen wird) N-mal aufgerufen.

Die Individuen werden mit Hilfe von Priority Queues angeordnet; weil für jede neue Generation eine Priority Queue aufgebaut wird, erhält man einen Zeitverbrauch $O(N \log N)$ für die Anordnung der N Individuen in jeder Generation.

Die Erstellung der Startpopulation und der jeweils neuen Population, das Crossover und die Mutation bestehen jeweils nur aus Aufrufen des Generators für Pseudo-Zufallszahlen (siehe dazu die Seiten 92ff.) und Operationen, die die Werte einzelner Gene verändern. Dabei wird jedes Gen einer Population höchstens einmal behandelt, und man erhält für jeden der eben angesprochenen Operatoren eine Komplexität von $O(|V \setminus Z| N)$. Die erneute Berechnung der DKNUX-Wahrscheinlichkeiten bedarf des Durchlaufs aller Kanten des gegebenen Graphen G, um die von der Heuristik gelieferte Näherungslösung und die Nachbarn eines jeden Knotens $v \in V \setminus Z$ zu vergleichen. Daher hat diese Aktion die Komplexität $O(|E|)$.

4.3.7 Abbruchkriterien

Wir gehen jetzt noch auf verschiedene Abbruchkriterien ein, die verwendet werden, um die Iteration eines genetischen Algorithmus zu beenden. Wir werden, da hier der genetische Algorithmus getestet wird, eine feste Anzahl von Populationen vorgeben und während der dadurch begrenzten Evolution Informationen über die während der Iterationen aufgetretenen Ergebnisse sammeln.

Bei zeitkritischen Anwendungen ist jedoch daran zu denken, dem Algorithmus eine Zeitschranke zu setzen und solange zu iterieren, bis die gegebene Zeit abgelaufen ist. Dies ist möglich, weil die einzelne Anwendung eines heuristischen Verfahrens wie z.B. SPH-z oder DNIH, das zur Bewertung der Individuen dient, nur geringe Zeit benötigt. Gibt man dem genetischen Algorithmus also wenigstens soviel Zeit, bis $G(0)$ erzeugt und bewertet wurde (im wesentlichen die Zeit für N Durchläufe der jeweils gewählten Heuristik), so hat man etwa gegenüber einem branch-and-bound-Algorithmus den Vorteil, stets eine zulässige Lösung vorzufinden, egal, wann man die Evolution abbricht.

Eine andere häufig verwendete Alternative besteht darin, den Algorithmus abzubrechen, falls sich das beste bisher erhaltene Individuum während einer vorgegebene Anzahl von Generationen nicht mehr verändert hat, oder wenn die Population nur noch aus wesentlich weniger als N voneinander verschiedenen Individuen besteht. Wegen der fehlenden Diversität innerhalb der Population geht man in letzterem Fall davon aus, daß auch durch weitere Evolution (d.h. insbesondere durch weitere Anwendung eines Crossover-Operators) keine Fortschritte mehr zu erzielen sind. Dieses Kriterium wird besonders dann angewandt, falls genügend Zeit zur Verfügung steht, ein möglichst gutes Ergebnis erzielt werden soll und man nicht durch andere Methoden wenigstens abschätzen kann, wie weit das bisher erreichte Ergebnis noch vom eigentlichen Optimum entfernt liegt.

Besser sieht es schon aus, wenn eine untere Schranke für das gesuchte Minimum vorliegt. Man kann in diesem Falle abbrechen, sobald man dieser Schranke von oben „genügend nahe“ gekommen ist. Dieser zu erreichende Maximalabstand ist natürlich wiederum nur schätzbar und hängt vom zu erwartenden Abstand der unteren Schranke vom Optimum ab. Erreicht man innerhalb des genetischen Algorithmus allerdings die untere Schranke, so kann man sicher davon ausgehen, das Optimum auch gefunden zu haben. Diese Idee liegt Abschnitt 4.5 zugrunde.

4.4 Modifizierter Satz über Schemata

Wir werden in diesem Abschnitt zeigen, daß der bisher dargestellte genetische Algorithmus eine Eigenschaft aufweist, die auch der klassische, von Holland eingeführte Algorithmus besitzt, und die die Grundlage für die Wirksamkeit genetischer Algorithmen schlechthin bildet. Zunächst werden wir den Begriff „Schema“ klären, um dann das zentrale Ergebnis von Holland nachzuvollziehen und schließlich einen an den genetischen Algorithmus mit DKNUX-Operator angepaßten Satz über Schemata aufzustellen.

4.4.1 Schemata und implizite Parallelität

Die Individuen $g \in G(t)$, die während der Evolution eines genetischen Algorihtmus auftreten, existieren im allgemeinen nur wenige Iterationen lang. Eine genaue Untersuchung ist deshalb nur möglich mit Hilfe des jetzt einzuführenden Begriffs der Schemata.

Setzt man eine Codierung wie in Abschnitt 4.1 voraus, so ist ein Schema $s := (s_j)_{j \in V \setminus Z}$ ein Tupel aus der Menge $S := \{0, 1, \diamond\}^{|V \setminus Z|}$. Das Zeichen „$\diamond$“ wird als eine nicht definierte Position im Schema s interpretiert; die Anzahl der definierten Positionen bezeichnet man ([Gol89]) mit

$$o(s) := \left| \{ j \in V \setminus Z \mid s_j \neq \diamond \} \right| .$$

Betrachtet man ein Schema wie eben und einen Genotyp $g = (g_j)_{j \in V \setminus Z}$, so spricht man davon, daß g auf das Schema s paßt, in Zeichen $g \in s$, falls gilt:

$$g_j = s_j, \quad \forall j \in V \setminus Z \text{ mit } s_j \neq \diamond .$$

In diesem Sinne paßt der Genotyp $(1, 1, 0)$ auf das Schema $(\diamond, 1, 0)$, der Genotyp $(0, 1, 1)$ jedoch nicht.

Umgekehrt repräsentiert der Genotyp $(1,1,0)$ 2^3 verschiedene Schemata, etwa $(1,1,0)$, $(\diamond,1,0)$, $(1,\diamond,0)$ etc. Im allgemeinen steht ein Genotyp der Länge $|V\backslash Z|$ für $2^{|V\backslash Z|}$ verschiedene Schemata. Letztere Betrachtungsweise fördert das Verständnis für die Funktionsweise genetischer Algorithmen; man kann nämlich eine Population $G(t)$ von N Individuen als eine Menge von Stichproben für die zwischen $2^{|V\backslash Z|}$ und $N2^{|V\backslash Z|}$ verschiedenen Schemata interpretieren, die in $G(t)$ repräsentiert sind. Dieses Prinzip, daß die genetischen Operatoren also gleichzeitig auf die Individuen von $G(t)$ und auf die große Zahl implizit repräsentierter Schemata wirken, nennt Goldberg „implicit parallelism" ([Gol89]); Holland zieht neuerdings ebenfalls diese Bezeichnung vor, nachdem er in früheren Ausgaben seines Buches ([Hol92], siehe dazu insbesondere das Vorwort zur Ausgabe von 1992) stattdessen von „intrinsic parallelism" gesprochen hat. Hier werden wir dieses Prinzip deshalb in Übersetzung „implizite Parallelität" nennen.

Es seien, der Notation von [Gol89] folgend, für $t \in \mathbb{N} \cup \{0\}$ und $s \in S$:

$$m(s,t) := \big|\{\, g \in G(t) \mid g \in s \,\}\big|$$

die Anzahl an Individuen in $G(t)$, die auf ein Schema s passen. Weiter sei

$$p(s,t) := \frac{1}{m(s,t)} \sum_{\substack{g^{(t,i)} \in s \\ i \in \{1,\ldots,N\}}} p_i$$

die durchschnittliche (durch Ranking gegebene) Fitness von Genotypen der wie in 4.2 angeordneten Population $G(t) := \{g^{(t,1)},\ldots,g^{(t,N)}\}$, welche auf das Schema $s \in S$ passen.

Man betrachtet zunächst den genetischen Algorithmus ohne Crossover- und Mutationsoperatoren, d.h. allein die durch den Wahrscheinlichkeitsvektor $P(t)$ gesteuerte Reproduktion von Individuen von Generation $G(t)$ nach $G(t+1)$. Dann gilt für die zu erwartende Anzahl der in Population $G(t+1)$ auf das Schema s passenden Individuen:

$$\begin{aligned} m(s,t+1) &= N \sum_{\substack{g^{(t,i)} \in s \\ i \in \{1,\ldots,N\}}} p_i = \\ &= N\, m(s,t) p(s,t) = m(s,t) \frac{p(s,t)}{\bar{p}} \;, \end{aligned} \tag{4.1}$$

falls $\bar{p} = 1/N$ die konstante durchschnittliche Fitness der Individuen in $G(t)$ für $t \in \mathbb{N} \cup \{0\}$ bezeichnet, und wie bei Goldberg und Holland der Parameter $N_{\max} = 0$ vorausgesetzt wird.

Ein Schema s, das aufgrund der Stichprobe

$$\{g^{(t,i)} \in s \mid i \in \{1,\ldots,N\}\} \subset \{g \in s \mid g \in G\}$$

überdurchschnittliche Fitness zu besitzen scheint, wird also (wenn in den Folgegenerationen diese Überdurchschittlichkeit erhalten bleibt) durch eine exponentiell wachsende Anzahl von Individuen in den folgenden Populationen repräsentiert (hierzu und zur Optimalität dieses Verfahrens bei Anwendung auf das Problem „k-armiger Banditen" siehe in [Gol89] die Seiten 30ff.).

Die folgenden beiden Unterabschnitte werden nun die Einflüsse von Crossover und Mutation in die Gleichung (4.1) einbeziehen.

4.4.2 Satz von Holland

Innerhalb eines klassischen genetischen Algorithmus spielt ein etwas einfacherer Typ von Crossover-Operatoren als der hier dargestellte DKNUX-Operator die zentrale Rolle, aus einer Population $G(t)$ mit Verteilung $P(t)$ eine neue Menge $\bar{G}(t)$ von Individuen abzuleiten. Man nennt diesen zuerst von Holland verwendeten Operator den 1-Punkt-Crossover-Operator, kurz „X“; er ist abhängig von einem Parameter $P_X \in [0,1]$ wie folgt definiert, wenn man Genotypen der Länge $|V \backslash Z|$ voraussetzt.

$G(t)$ sei wie in Abschnitt 4.2 geordnet. Der 1-Punkt-Crossover-Operator erzeugt eine Menge $\bar{G}(t) = \{\bar{g}^{(t,1)}, \ldots, \bar{g}^{(t,N)}\}$ durch:

- Für $i = 1, \ldots, N$
  - Man wähle zwei Genotypen $g^{(t,i_1)}$, $g^{(t,i_2)}$ aus $G(t)$ mit $i_1 \neq i_2$ zufällig gemäß der Verteilung $P(t)$.
  - Man wähle $r \in [0,1]$ zufällig gleichverteilt und unterscheide: falls $r > P_X$, setze $\bar{g}^{(t,i)} := g^{(t,i_1)}$; ansonsten wähle man $n_X \in V \backslash (Z \cup \max_{V \backslash Z})$ zufällig gleichverteilt und setze $\bar{g}^{(t,i)} := (\bar{g}_j^{(t,i)})_{j \in V \backslash Z}$ mit

$$\bar{g}_j^{(t,i)} := \begin{cases} g_j^{(t,i_1)} & \text{falls } j \leqslant n_X \\ g_j^{(t,i_2)} & \text{sonst} \end{cases}$$

Sind zum Beispiel $V = \{1, \ldots, 5\}$, $Z = \{2, 4\}$, so ist $n_X \in \{1, 3\}$. Obiges Crossover von $(1,1,0)$ und $(0,0,1)$ mit $n_X = 1$ erbringt dann $(1,0,1)$ als neues Individuum.

Um den Einfluß des X-Operators auf Gleichung (4.1) angeben zu können, berechnet man eine untere Schranke für die Wahrscheinlichkeit, daß ein Schema s mit $g^{(t,i_1)} \in s$ durch diesen Operator nicht zerstört wird, d.h. dafür, daß dann auch noch für einen erzeugten Nachkommen $\bar{g}^{(t,i)} \in s$ gilt.

Zunächst benötigt man hierfür noch die sogenannte „definierende Länge“ $\delta(s)$ eines Schemas $s \in S$. Sie ist festgelegt als

$$\begin{aligned} \delta(s) := \big| \{ j \in V \backslash Z \mid & \exists j_1 \in V \backslash Z, j_1 \leqslant j, \text{ mit } s_{j_1} \neq \diamond, \\ & \exists j_2 \in V \backslash Z, j < j_2, \text{ mit } s_{j_2} \neq \diamond \} \big| \end{aligned}$$

und hat zum Beispiel für $s = (\diamond, 1, \diamond, \diamond, 0, 1)$ einen Betrag von $\delta(s) = 4$; für $s = (\diamond, 0, \diamond, \diamond, \diamond, \diamond)$ ist die definierende Länge $\delta(s) = 0$.

Ein Schema $s \in S$ kann nun, falls wegen $r \leqslant P_X$ überhaupt Crossover durchgeführt wird, nur zerstört werden, falls n_X zwischen zwei definierte Stellen von s fällt. Die Wahrscheinlichkeit $P_X(s)$, daß s das Crossover überlebt, ist also wegen der Gleichverteilung von n_X über alle Stellen des Schemas mindestens

$$P_X(s) \geqslant 1 - P_X \frac{\delta(s)}{|V \backslash Z| - 1} \ .$$

Damit nun das Schema auch in der neuen Generation $G(t+1)$ repräsentiert ist (bei $N_{\max} = 0$, wie es auch Holland und Goldberg annehmen), darf auch die Mutation an den definierten Stellen des Schemas keine Änderungen vornehmen; die Wahrscheinlichkeit $P_{\mathrm{MUT}}(s)$, daß s auch die Mutation unverändert übersteht, ist demnach

$$P_{\mathrm{MUT}}(s) = (1 - P_{\mathrm{MUT}})^{o(s)} \ .$$

Insgesamt erhält man den fundamentalen Satz über genetische Algorithmen von Holland jetzt für den Fall, daß Ranking als Verfahren zur Fitness-Bestimmung für die Individuen benützt wird:

Satz 4.4.1
Für die erwartete Anzahl an Repräsentanten eines Schemas $s \in S$ bei Verwendung von Ranking, 1-Punkt-Crossover und Mutation gilt die Abschätzung:

$$\begin{aligned} m(s, t+1) &= m(s,t)\frac{p(s,t)}{\bar{p}} P_X(s) P_{MUT}(s) \geqslant \\ &\geqslant m(s,t)\frac{p(s,t)}{\bar{p}}\left(1 - P_X \frac{\delta(s)}{|V \setminus Z| - 1}\right)(1 - P_{MUT})^{o(s)} \ . \end{aligned}$$

Dies bedeutet, daß ein überdurchschnittliches Schema $s \in S$ von kurzer definierender Länge $\delta(s)$ und niedriger Ordnung $o(s)$ in den Folgegenerationen eine exponentiell wachsende Anzahl von Repräsentanten erhält. Betrachtet man Schemata als Teillösungen des Problems, so erwartet man, daß gemäß dieser Aussage „gute“ Teillösungen während der Evolution mit Hilfe des Crossover-Operators zusammengesetzt werden. Man nimmt an, daß hierbei immer bessere Näherungslösungen entstehen; diese Annahme nennt man die „building blocks“-Hypothese für genetische Algorithmen.

4.4.3 Modifizierter Satz

Der in Abschnitt 4.3 dargestellte genetische Algorithmus unterscheidet sich vom klassischen Algorithmus lediglich darin, daß er den DKNUX-Operator anstatt des X-Operators anwendet. Deshalb kann man, wenn $P_{\mathrm{DKNUX}}(s,t)$ die Wahrscheinlichkeit bezeichnet, daß ein Schema $s \in S$ diesen Operator bei gegebenem DKNUX-Vektor $q^{(t)}$ unverändert übersteht, das Ergebnis des letzten Unterabschnitts sofort übertragen:

$$m(s,t+1) = m(s,t)\frac{p(s,t)}{\bar{p}} P_{\mathrm{DKNUX}}(s,t) P_{\mathrm{MUT}}(s) \ .$$

Es sei nun $s \in S$ ein Schema, das wegen $g^{(t,i_1)} \in s$ zur Bearbeitung durch DKNUX ansteht. Es sei mit $j \in V \setminus Z$

$$p_j(s,t) := P\left(\bar{g}_j^{(t,i)} = g_j^{(t,i_1)}\right)$$

die Wahrscheinlichkeit, daß die Stelle j im Genotyp durch DKNUX nicht verändert wird.

Man kann, abhängig von den DKNUX-Wahrscheinlichkeiten und dem Wert von $g_j^{(t,i_1)}$, wegen der Definition des DKNUX-Operators (siehe Unterabschnitt 4.3.3) weiter schreiben:

$$p_j(s,t) = \begin{cases} P\left(g_j^{(t,i_1)} = g_j^{(t,i_2)}\right) + P\left(g_j^{(t,i_1)} \neq g_j^{(t,i_2)}\right) q_j^{(t)} & \text{für } g_j^{(t,i_1)} = 1 \\ P\left(g_j^{(t,i_1)} = g_j^{(t,i_2)}\right) + P\left(g_j^{(t,i_1)} \neq g_j^{(t,i_2)}\right)(1 - q_j^{(t)}) & \text{sonst.} \end{cases}$$

Die Überlebensfähigkeit eines einzelnen Gens hängt also von den Werten der Gene der anderen Individuen der Population (diese Eigenschaft stimmt mit den anderen Typen von Crossover-Operatoren überein), aber eben auch von den (problemspezifischen) DKNUX-Wahrscheinlichkeiten ab.

Die Überlebensfähigkeit eines Schemas (in [Ran94] wird dies „survivability" genannt) wird ausschließlich von der Überlebensfähigkeit der Gene an den definierenden Positionen des Schemas $s = (s_j)_{j \in V \setminus Z}$ bestimmt, und man erhält deswegen:

$$P_{\text{DKNUX}}(s,t) \geqslant \prod_{\substack{s_j \neq \diamond \\ j \in V \setminus Z}} p_j(s,t) \ .$$

Damit hat man aber schon den modifizierten

Satz 4.4.2
Für die erwartete Anzahl an Repräsentanten eines Schemas $s \in S$ bei Verwendung von Ranking, DKNUX-Operator und Mutation gilt die Abschätzung:

$$\begin{aligned} m(s,t+1) = m(s,t)\frac{p(s,t)}{\bar{p}}P_{DKNUX}(s,t)P_{MUT}(s) \geqslant \\ \geqslant m(s,t)\frac{p(s,t)}{\bar{p}}P_{DKNUX}\prod_{\substack{s_j \neq \diamond \\ j \in V \setminus Z}} p_j(s,t)(1-P_{MUT})^{o(s)} \ . \end{aligned}$$

Die definierende Länge eines Schemas und die Anordnung der Gene auf den Genotypen haben also keinen Einfluß mehr auf die erwartete Anzahl an Repräsentanten eines Schemas. Vielmehr sagt der Satz jetzt, daß ein überdurchschnittliches Schema $s \in S$ von niedriger Ordnung $o(s)$ und hoher Überlebensfähigkeit („survivability") in den Folgegenerationen eine exponentiell wachsende Anzahl von Repräsentanten erhält. Dabei ändert sich die Überlebensfähigkeit der Individuen einer Population insbesondere dann, wenn ein neuer an einem Individuum, für das die Heuristik ein besseres Ergebnis erzielt, orientierter DKNUX-Vektor errechnet wird; dies kann höchstens alle N_{DKNUX} Iterationen während des Ablaufs des genetischen Algorithmus geschehen.

4.5 Untere Schranke

Im Unterabschnitt 4.3.7 wurde schon dargelegt, daß es nützlich sein kann, eine untere Schranke für $\min(\text{USTP'}_{G,Z})$ bzw. für $\min(\text{DSTP'}_{D_G,Z})$ (was nach Satz 1.3.4 ja gleichbedeutend ist) zu kennen. Damit kann man nämlich einerseits ein Abbruchkriterium für den genetischen Algorithmus aufstellen, und andererseits die Qualität von Ergebnissen, die die besprochenen Heuristiken für Probleme liefern, für die die Optimallösungen nicht bekannt sind, wenigstens grob einschätzen. Der sogenannte „Duale Aufstiegs-Algorithmus" (DAA, „dual ascent algorithm" in [Won82] und [Win87]) ist imstande, sowohl eine solche untere Schranke als auch - gleichsam als Nebenprodukt - eine zulässige Lösung für DSTP'$_{D_G,Z}$ zu liefern.

4.5.1 Formulierung als lineares Programm

Man geht davon aus, daß ein gemäß der Definition in (1.1) auf Seite 9 von einem Graphen G abgeleiteter Digraph $D_G = (N, A, d')$ und eine Menge $Z \subseteq N$ spezieller Knoten vorliegen. Ein Wurzelknoten $r \in Z$ sei ausgezeichnet; die Menge der übrigen speziellen Knoten werde mit $Z^+ := Z \backslash \{r\}$ bezeichnet. Dann läßt sich das Problem DSTP'$_{D_G,Z}$ nach [Won82] als ein ganzzahliges lineares Programm formulieren:

$$\min \sum_{(v,w)\in A} d'_{(v,w)} x_{v,w}$$

unter den Nebenbedingungen

$$x_{v,w} \geqslant y^z_{v,w} \qquad \forall (v,w) \in A, \forall z \in Z^+ \tag{4.2}$$

$$\sum_{(v,w)\in A} y^z_{v,w} - \sum_{(u,v)\in A} y^z_{u,v} = \begin{cases} 1 & \text{falls } v = r \\ -1 & \text{falls } v = z \\ 0 & \text{sonst} \end{cases} \qquad \forall z \in Z^+ \tag{4.3}$$

$$x_{v,w} \in \{0,1\} \qquad \forall (v,w) \in A \tag{4.4}$$

$$y^z_{v,w} \geqslant 0 \qquad \forall (v,w) \in A, \forall z \in Z^+ \tag{4.5}$$

Die $x_{v,w}$ sind binäre Variable, welche angeben, ob die Kante (v,w) in der Lösung enthalten ist ($x_{v,w} = 1$) oder nicht ($x_{v,w} = 0$). Eingebettet in das obige lineare Programm ist ein Netzfluß-Problem („multi commodity network flow problem“, [Won82]), bei dem $|Z^+|$ verschiedene Waren von $r \in Z$ aus auf die Reise durch den Digraphen geschickt werden und jede an genau einem Knoten $z \in Z^+$ ankommen soll (Gleichungen (4.3)). Die Variablen $y^z_{v,w}$ geben an, welche Menge der Ware $z \in Z^+$ sich auf der Kante $(v,w) \in A$ bewegt; Ungleichung (4.2) läßt einen Warenfluß nur auf Kanten zu, die in der Lösung enthalten sind. Die (Un-)Gleichungen (4.2) bis (4.5) bedingen also, daß eine zulässige Lösung einen gerichteten Pfad zwischen r und jedem Knoten $z \in Z^+$ besitzen muß. Wenn für alle Kanten $(v,w) \in A$ strikt $d'_{(v,w)} > 0$ gilt, so ist die Lösung des obigen linearen Programms auch eine Lösung von DSTP'$_{D_G,Z}$.

4.5.2 Dualisierung und Wongs Aufstiegsverfahren DAA

Dualisieren der linearen Relaxation des im letzten Unterabschnitt aufgestellten Programms (d.h. des obigen Programms ohne die Ganzzahligkeitsvoraussetzungen) ergibt das Programm:

$$\max \sum_{z\in Z^+} (x^z_z - x^z_r)$$

unter den Nebenbedingungen

$$x^z_w - x^z_v - y^z_{v,w} \leqslant 0 \qquad \forall (v,w) \in A, \forall z \in Z^+ \tag{4.6}$$

$$\sum_{z\in Z^+} y^z_{v,w} \leqslant d'_{v,w} \qquad \forall (v,w) \in A \tag{4.7}$$

$$x^z_v \geqslant 0 \qquad \forall v \in N, \forall z \in Z^+ \tag{4.8}$$

$$y^z_{v,w} \geqslant 0 \qquad \forall (v,w) \in A, \forall z \in Z^+ \tag{4.9}$$

Wong schlägt nun in [Won82] das folgende einfache Aufstiegsverfahren vor, um dieses duale Programm näherungsweise zu lösen. Hierbei wird ein Hilfsdigraph $D' = (N, A', d')$ schrittweise um weitere Kanten erweitert. Am Anfang ist $A' = \emptyset$.

Wir schreiben für je zwei Knoten $v, w \in N$:

$$v \underset{D'}{\overset{*}{\twoheadrightarrow}} w :\Leftrightarrow v \underset{D'}{\overset{*}{\rightarrow}} w, w \underset{D'}{\overset{*}{\not\rightarrow}} v \ ,$$

d.h. es gibt einen gerichteten Pfad von v nach w, aber nicht umgekehrt. Damit läßt sich nun formulieren:

Definition 4.5.1
Eine Zusammenhangskomponente $T \subseteq N$ von D' heißt Wurzelkomponente, falls $Z^+ \cap T \neq \emptyset$ und für alle Knoten $t \in T$ und $v \in N$ gilt:

$$v \underset{D'}{\overset{*}{\twoheadrightarrow}} t \Rightarrow v \notin Z \ .$$

Nach solchen Wurzelkomponenten wird in jeder Iteration von DAA zu suchen sein; wir verwenden für diese Aufgabe eine an die zusätzliche Bedingung, die Wurzelkomponenten erfüllen müssen, angepaßte Version des bekannten Algorithmus von Tarjan (siehe etwa [Sed92], Seiten 481ff.), den dieser 1972 veröffentlicht hat. Der Algorithmus dient in seiner ursprünglichen Fassung zum Auffinden der Zusammenhangskomponenten eines Digraphen.

Eine einfache Tiefensuche in D' wird eingesetzt, um während des Algorithmus von Wong die Mengen

$$C(w) := \{\, v \in N \mid v \underset{D'}{\overset{*}{\rightarrow}} w \,\}$$

und darauf aufbauend das sogenannte „cut set“ von w

$$\mathrm{Cut}(w) := \{\, (u,v) \in A \mid (u,v) \notin A', v \in C(w), u \notin C(w) \,\}$$

zu bestimmen.

Algorithmus 8 (Algorithmus von Wong: DAA)

Schritt 0: Initialisierung

D' wird generiert und die folgenden Variablen gesetzt:

$$\begin{aligned} x_v^z &:= 0 && \forall v \in N, \forall z \in Z^+ \\ y_{v,w}^z &:= 0 && \forall (v,w) \in A \\ S(i,j) &:= d'_{(v,w)} && \forall (v,w) \in A \end{aligned}$$

Schritt 1: Auswahl einer Wurzelkomponente

Wähle eine Wurzelkomponente T aus D'.

Falls keine existiert, wird der Algorithmus gestoppt.

Schritt 2: Aufstiegsschritt

Es sei $z \in Z^+ \cap T$ beliebig gewählt.

Weiter seien $\bar{v}, \bar{w}$ so gewählt, daß gilt:

$$S(\bar{v},\bar{w}) = \min\{\, S(v,w) \mid (v,w) \in \mathrm{Cut}(z) \,\} \ .$$

Man setze dann

$$\begin{aligned} x_v^z &:= x_v^z + S(\bar{v}, \bar{w}) & \forall v \in C(z) \\ y_{v,w}^z &:= y_{v,w}^z + S(\bar{v}, \bar{w}) & \forall (v,w) \in \mathrm{Cut}(z) \\ S(v,w) &:= S(v,w) - S(\bar{v}, \bar{w}) & \forall (v,w) \in \mathrm{Cut}(z) \end{aligned}$$

Schritt 3: Erweiterung des Hilfsgraphen
Setze $A' := A' \cup \{\bar{u}, \bar{v}\}$ und gehe zu Schritt 1.

Die Anfangslösung, die obiges Verfahren benützt (alle Variablen auf 0 gesetzt), ist zulässig; Wong beweist, daß die durch Schritt 2 gelieferte neue Lösung jeweils zulässig für das duale Problem bleibt. Weil in jedem Schritt eine Kante zum Hilfsgraphen D' hinzugefügt wird, terminiert der Algorithmus - so Winter in [Win87] - nach endlich vielen Schritten.

4.5.3 Heuristik DAH

Wong hat in seinem Artikel weiter darauf hingewiesen, daß man aus dem vom DAA aufgebauten Hilfsgraphen $D' = (N, A', d')$ eine zulässige Lösung für USTP'$_{G,Z}$ erstellen kann. Grundlage dafür ist die Beobachtung, daß am Ende von DAA für die Menge der Knoten

$$N' := \{ v \in N \mid \mathcal{P}_{D'}(r, v) \neq \emptyset \} ,$$

die über einen gerichteten Pfad in D' vom ausgezeichneten Wurzelknoten $r \in Z$ aus erreicht werden können, die Inklusion $Z \subseteq N'$ gilt. Deshalb ist der ungerichtete Graph $G' := (V', E', d, g)$ mit

$$V' := N' \text{ und } E' := \bigcup_{\substack{(v,w) \in A' \\ v,w \in N'}} \{\{v, w\}\}$$

eine zulässige Lösung für USP'$_{G,Z}$.

Wendet man auf G' nun noch die beiden (zunächst nur für PD-Heuristiken vorgesehenen) Schritte 4 und 5 aus Unterabschnitt 2.1.3 an, so erhält man einen Baum in G, welcher Z aufspannt, d.h. also eine zulässige Lösung für USTP'$_{G,Z}$.

Was das vollkommen analoge Vorgehen unter Beibehaltung von gerichteten Graphen (wenn also eine Näherungslösung für das Problem DSTP'$_{D_G,Z}$ gesucht wird) betrifft, sei der Leser auf die Bemerkungen von Winter verwiesen.

4.5.4 Beispiel

Das schon von den Heuristiken SPH und DNIH bekannt Beispielproblem löst die DA-Heuristik näherungsweise mit einem Wert von 8.8. Die Näherungslösung ist in Abbildung 4.1 zu sehen. Die vom dualen Aufstiegsalgorithmus berechnete untere Schranke für dieses Beispielproblem ist 8.0.

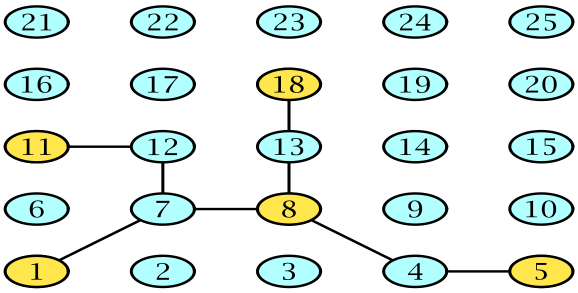

Abbildung 4.1: Lösung mit DAH

4.5.5 Klasse TWong

Die Klasse `TWong`, die die beiden Verfahren DAA und DAH realisiert, besitzt das in Abbildung 4.2 dargestellte Interface. Zentrale Methode ist `algo`, die je nach Wert der Flagge `flag` entweder nur DAA oder sogar DAH anwendet. Die Elementfunktionen im einzelnen:

`CTvisit(int)` und `CTDFS(int)`
: Führen rekursiv Tiefensuche im Hilfsgraphen durch wie in Unterabschnitt 4.5.2 angedeutet.

`RCvisit(int,int)` und `RCDFS(void)`
: Führen rekursiv den modifizierten Algorithmus von Tarjan aus.

`TWong()` und `reset(void)`
: Stellen eine leere Instanz ohne Funktionalität bereit.

`TWong(const TGr &g, int *sn)` und `reset(const TGr &g, int *sn)`
: Bereiten eine Instanz von `TWong` darauf vor, eine untere Schranke für das durch `g` und `sn` gegebenen Problems USTP'$_{D_G,Z}$ und ggf. auch eine Näherungslösung dafür zu berechnen. Die Strukturen `g` und `sn` haben dabei genau dieselbe Bedeutung wie bei den PD-Heuristiken; sie ist auf Seite 67 spezifiziert.

`FPTYPE algo(TGr &daht, FPTYPE &dahw, int flag=fLowerBound)`
: Diese Methode führt zunächst DAA durch und gibt die errechnete untere Schranke als Funktionwert zurück. Ist das `flag` auf den Wert von `fHeuristic` gesetzt, so wird aus dem in DAA aufgebauten Hilfsgraphen außerdem - wie auf Seite 86 beschrieben - eine zulässige Lösung für USTP'$_{G,Z}$ konstruiert. Ihr Wert wird dann in `dahw` abgelegt, der dazugehörende Digraph in `daht`. Die dazugehörige Näherungslösung kann man dann aus diesem Digraphen wie auf Seite 67 dargestellt (Gleichung (3.1)) rekonstruieren.

```
class TWong
    {
    private:
        // (...)

    protected:
        int  CTvisit(int);
        int  CTDFS(int);
        int  RCvisit(int,int);
        int  RCDFS(void);

    public:
        TWong();
        TWong(const TGr &g, int *sn);
        ~TWong();

        void reset(void);
        void reset(const TGr &g, int *sn);

        enum {fLowerBound, fHeuristic};
        FPTYPE algo (
                    TGr &daht,
                    FPTYPE &dahw,
                    int flag=fLowerBound
                    );

        friend ostream &operator<<(ostream &, const TWong &);
    };
```

Abbildung 4.2: Deklaration der Klasse TWong

4.6 Implementierung des genetischen Algorithmus

Dieser Abschnitt befäßt sich mit der Implementierung des bisher dargestellten genetischen Algorithmus. Zunächst wird die Klasse TGA eingeführt, die zur Verwaltung und Evolution von Populationen dient, die wie in Abschnitt 4.2 angeordnet sind. Die Methoden dieser Klasse werden die in den Unterabschnitten 4.3.1 bis 4.3.5 gezeigten Funktionen realisieren.

Zur Auswertung der Genotypen wird jede Instanz von TGA mittels eines (bei der Konstruktion zu setzenden) Zeigers auf einen Abkömmling der Klasse TGA_DecEval zugreifen. Letztere Klasse besitzt keine eigene Funktionalität; sie ist eine abstrakte Klasse und dient lediglich dazu, eine Schnittstelle zwischen den bereits implementierten Heuristiken (etwa TD-NIH) und TGA herzustellen.

Zuletzt werden wir darlegen, wie die Pseudozufallszahlen erzeugt werden können, die innerhalb der genetischen Operatoren ständig gebraucht werden. In vielen Compilern für die Sprache C++ sind zwar Generatoren für die Erzeugung solcher Zahlenfolgen vorhanden, doch wurde hier mit wenig

Aufwand explizit eine Klasse mit Namen `TRandom` deklariert, die diese Aufgabe übernehmen kann; dies geschah mit dem Ziel, nachvollziehbare und von Compiler-spezifischen Definitionen unabhängige Ergebnisse zu erhalten.

4.6.1 Klasse TGA der genetischen Operatoren

Die Methoden der in 4.3 angebildeten Klasse `TGA` haben die folgenden Aufgaben:

`TGA()` `void reset(void)`
Stellen eine leere Instanz ohne Funktionalität bereit.

`TGA(...)` und `void reset(...)`
Bereiten eine Instanz von `TGA` darauf vor, die genetischen Operatoren auf das durch `g` und `sn` wie in Abschnitt 3.3 spezifizierte Problem $\text{USTP'}_{G,Z}$ anzuwenden. Dabei wird eine Population der Größe `N` aufgebaut und gepflegt. Der Zeiger `ph` weist auf die Instanz einer von `TGA_DecEval` abgeleiteten Klasse, welche die innerhalb der genetischen Operatoren zu verwendende Heuristik realisiert.

`int *getGenoBest1(void)`
Liefert einen Zeiger auf den Genotyp des besten bisher aufgefundenen Individuums.

`int *getGenoIndiv(int i)`
Liefert einen Zeiger auf den Genotyp des `i`-ten Individuums der aktuellen Population.

`getnPop(void)`
Liefert die Populationsgröße.

`getnGene(void)`
Liefert die Länge der Chromosomen.

`void calcKNUXinfo(int fknux=TGA::fKNUX1)`
Berechnet, abhängig vom Inhalt der Flagge `fknux`, die DKNUX-Wahrscheinlichkeiten neu. Gleicht der Wert der Flagge `TGA::fKNUX1`, wird der Wahrscheinlichkeitsvektor wie in Unterabschnitt 4.3.2 besetzt, ist der Wert gleich `TGA::fUX`, wird er an allen Stellen mit $1/2$ besetzt, so daß die Methode `KNUX` lediglich UX durchführt.

`void initialPopulation(...)`
Erzeugt eine Startpopulation mit $P_{\text{Gen1}} =$ `pGeneIs1`. Ist `fSeeding` auf 1 gesetzt, wird „seeding“ durchgeführt, ansonsten nicht.

`void thenextPopulation(FPTYPE oldPopFrac)`
Erzeugt eine neue Population; dabei ist `oldPopFrac` mit

$$N_{\max} \leqslant N\ \texttt{oldPopFrac} < N_{\max} + 1$$

der Anteil an Individuen, der höchstens direkt aus der alten in die neue Generation übernommen wird.

```
class TGA
    {
    private:
        // ( ... )
        TRandom rnd;
        TGA_DecEval *deceval;

    public:
        TGA();
        TGA (
            const TGr &g, const int *sn,
            int N,
            TGA_DecEval *ph
            );
        ~TGA();

        void reset(void);
        void reset  (
                    const TGr &g, const int *sn,
                    int N,
                    TGA_DecEval *ph
                    );

        int *getGenoBest1(void);
        int *getGenoIndiv(int i);
        int getnPop(void);
        int getnGene(void);

        enum {fKNUX1=0, fUX=1};
        void calcKNUXinfo(int fknux=TGA::fKNUX1);
        void initialPopulation( FPTYPE pGeneIs1,
                                int fSeeding=0 );
        void thenextPopulation(FPTYPE oldPopFrac);
        void KNUX(FPTYPE pX);
        void Mutate(FPTYPE pM);
        void Evaluate(void);

        friend ostream &operator<<(ostream &, const TGA &);
    };
```

Abbildung 4.3: Deklaration der Klasse TGA

```
class TGA_DecEval
    {
    public:
        TGA_DecEval() {};
        virtual ~TGA_DecEval() {};

        virtual FPTYPE DecEval  (
                                TGr &ht,
                                const int *gt,
                                int ng
                                ) = 0;
    private:
        TGA_DecEval(const TGA_DecEval &);
        TGA_DecEval &operator=(const TGA_DecEval &);
    };
```

Abbildung 4.4: Deklaration der abstrakten Klasse `TGA_DecEval`

`void KNUX(FPTYPE pX)`
: Realisiert den DKNUX-Operator mit $P_{\mathrm{DKNUX}} = \mathsf{pX}$.

`void Mutate(FPTYPE pM)`
: Realisiert die Mutation mit $P_{\mathrm{MUT}} = \mathsf{pM}$.

`void Evaluate(void)`
: Wendet die Heuristik auf all jene Individuen der aktuellen Generation an, für die noch keine Näherungslösung bekannt ist.

Man beachte, daß die Genotypen in TGA in einer etwas anderen Form als in Abschnitt 4.1 repräsentiert werden. Es sei $g = (g_j)_{j \in V \setminus Z} \in G$ ein Genotyp. Dieser wird innerhalb von TGA durch das Tupel $g' = (g'_j)_{j \in V \setminus Z}$ dargestellt mit

$$|g'_j| = j \text{ und}$$
$$g'_j < 0 \Leftrightarrow g_j = 1 \; .$$

Dadurch ist sofort klar, für welchen Knoten das Gen g'_j steht, was besonders die Dekodierung von Genotypen erleichtert.

4.6.2 Abstrakte Klasse `TGA_DecEval` und Abkömmlinge

Die Klasse `TGA_DecEval` ist – wie weiter oben schon erwähnt – lediglich Mittler zwischen den Anforderungen, die der genetische Algorithmus an eine Heuristik stellt, und der jeweiligen Implementierung einer Heuristik wie sie etwa in Abschnitt 3.3 vorgenommen wurde. Wichtigste Methode ist die rein virtuelle Funktion

`virtual FPTYPE DecEval (...) = 0`
: Diese Funktion wird innerhalb von `TGA_DecEval` nicht definiert. Die Abkömmlinge dieser Klasse implementieren in dieser Funktion eine

Heuristik für USTP'$_{G,Z}$, die zum Genotypen `gt` eine Näherungslösung `ht` berechnet; ihr Gewicht wird als Funktionswert zurückgeliefert. Im Parameter `ng` übergibt der genetische Algorithmus außerdem die Anzahl der Gene, die `gt` trägt.

Ein gemäß Abschnitt 4.3 aufgebauter genetischer Algorithmus hat also – jetzt mit Hilfe der Methoden der Klasse `TGA` formuliert – das folgende Aussehen, wenn die DNI-Heuristik zur Auswertung der Individuen benützt wird und die Klasse `TGA_DNIH` als Abkömmling von `TGA_DecEval` diese etwa unter Verwendung von `TDNIH` realisiert.

```
    // Initialisierung der Klassen-Instanzen //
TGA_DNIH ga_dnih(g,sn);
TGA ga(g,sn,N,&ga_dnih);

    // Genetischer Algorithmus //
ga.initialPopulation(pGeneIs1, fSeeding);
for (i=0; i<NumGen; i++)
    {
    if (i%KNUXInterval==0) ga.calcKNUXinfo();
    ga.KNUX(pX);
    ga.Mutate(pM);
    ga.thenextPopulation(oldPopFrac);
    }

    // Bestes erreichtes Individuum //
DNIHwgt = ga_dnih.DecEval(
                         DNIH,
                         ga.getGenoBest1(),
                         ga.getnGene()
                         );
```

Es wurden `NumGen` Generationen erzeugt und jede `KNUXInterval`-te Generation wurden die DKNUX-Wahrscheinlichkeiten neu erstellt. Die beste erreichte Näherungslösung liegt in `DNIH`, ihr Gewicht in `DNIHwgt`. Das Format der Lösung ist von der angewandten Heuristik abhängig.

4.6.3 Generator `TRandom` für Pseudo-Zufallszahlen

Bei jedem der vorangegangenen genetischen Operatoren mußten, genauso wie bei der Auswahl der Individuen zur Reproduktion, zufällig ganze Zahlen mit einer bestimmten Wahrscheinlichkeitsverteilung gezogen werden; wir stellen deshalb jetzt eine Methode vor, wie man eine Zahlenfolge $(x_k)_{k\in\mathbb{N}} \subset \{0,\dots,m-1\}$ mit $m \in \mathbb{N}$ erzeugen kann, die zufällig erscheint, d.h. viele Eigenschaften echter Zufallszahlen besitzt, und die der Gleichverteilung über dem Bereich $\{0,\dots,m-1\}$ genügt. Diese Zahlen x_k nennt man Pseudozufallszahlen. Ziel ist also, mit Hilfe dieser Pseudozufallszahlen ein Laplace-Experiment auf dem Wahrscheinlichkeitsraum $(\Omega, \mathcal{A}, P)$ zu simulieren :

$$\begin{aligned} \Omega &:= \{0, \tfrac{1}{m}, \ldots, \tfrac{m-1}{m}\}, \quad (m \in \mathbb{N}) \ , \\ \mathcal{A} &:= 2^{\Omega} \ , \\ P(\{\omega\}) &:= \tfrac{1}{m}, \quad \forall\, \omega \in \Omega \ . \end{aligned} \tag{4.10}$$

Dabei bedeute 2^{Ω} die Potenzmenge der Menge Ω.

Was aber im genetischen Algorithmus stets benötigt wird, ist ein Zufallsexperiment auf einem Raum $(\Omega', \mathcal{A}', P')$ mit

$$\begin{aligned} \Omega' &:= \{1, \ldots, n\}, \quad (n \in \mathbb{N}) \ , \\ \mathcal{A}' &:= 2^{\Omega'} \ , \\ P'(\{i\}) &:= p_i \geqslant 0, \quad \forall\, i \in \Omega' \ , \text{ wobei } \textstyle\sum_{i=1}^{n} p_i = 1 \ . \end{aligned} \tag{4.11}$$

Man wird nun versuchen, mit Hilfe des simulierbaren Experiments aus (4.10) und einer Zufallsvariablen $X : (\Omega, \mathcal{A}, P) \to (\Omega', \mathcal{A}')$ die Verteilung P' durch die Verteilung P_X im Bildraum zu nähern. Man definiert

$$p_i{}' := \sum_{k=1}^{i} p_k, \quad \forall\, i = 1, \ldots, n$$

und $p_0 := 0$ und damit die Zufallsvariable X durch:

$$X(\{\omega\}) := i, \text{ falls } p'_{i-1} \leqslant \omega < p_i{}' \ .$$

Für die durch X im Bildraum $(\Omega', \mathcal{A}')$ induzierte Verteilung P_X können wir für beliebiges $i \in \Omega'$ schreiben:

$$\begin{aligned} P_X(\{i\}) &= P(\, \omega \in [p'_{i-1}, p_i{}')\,) = \\ &= P(\, \tfrac{k}{m} \in [p'_{i-1}, p_i{}')\,) = \\ &= \tfrac{1}{m} \mid \mathbb{N} \cap [m p'_{i-1}, m p'_i) \mid \ . \end{aligned}$$

Für $x \in \mathbb{R}$ mit $\lfloor x \rfloor$ die größte ganze Zahl $n \in \mathbb{Z}$ bezeichnend, für die $n \leqslant x$ gilt, hat man also wegen $p'_i - p'_{i-1} = p_i$:

$$\lfloor m p_i \rfloor \leqslant m P_X(\{i\}) \leqslant \lfloor m p_i \rfloor + 1 \ ,$$

und damit

$$\mid P_X(\{i\}) - p_i \mid \leqslant \tfrac{1}{m}, \quad \forall\, i \in \Omega' \ . \tag{4.12}$$

Nach Ungleichung (4.12) kann man P' gut durch P_X nähern, besonders wenn $m \gg n$ gilt.

Es fehlt jetzt freilich noch das Verfahren, die gleichverteilten zufälligen $\omega \in \Omega$ möglichst gut zu simulieren; hierin kann man der sogenannten „Lineare Kongruenzmethode“ folgen, die seit ihrer Vorstellung durch Lehmer im Jahre 1951 in den überaus meisten Fällen, in denen ein „Zufallsgenerator“ vonnöten war, angewandt wurde, so Sedgewick, der diese Methode in [Sed92] auf den Seiten 509 ff. dargestellt; Sedgewick seinerseits bezieht sich u.a. auf Knuth [Knu81].

Man erzeugt bei dieser Methode gemäß der gleich folgenden Rekursionsformel eine Zahlenfolge (x_n), für die $(x_n)_{n\in\mathbb{N}} \subset \{0,\ldots,m-1\}$ gilt :

$$x_n := bx_{n-1} + c \pmod m, \quad \forall n \in \mathbb{N}. \tag{4.13}$$

Kritisch ist die Wahl der Parameter $x_0,\ b,\ c \in \{0,\ldots,m-1\}$ und $m \in \mathbb{N}$. Sedgewick wählt zunächst $c = 1$. Die Zahl m nun, die ja den Bereich bestimmt, in welchem die Pseudozufallszahlen liegen, will man einerseits möglichst groß wählen, andererseits muß man sich an den Datentypen orientieren, die zur Implementierung verfügbar sein werden; hier wird ein 32-Bit `int`-Typ verwendet werden. Deshalb kann man $m = 10^8$ wählen; möglich wäre es auch, m als die größte darstellbare ganze Zahl zu setzen, also etwa $m = 2^{32}$. Nach detaillierten Untersuchungen – so Sedgewick – habe Knuth festgestellt, es sei (wenn m wie oben definiert wurde) vorteilhaft, für b eine Zahl

$$b = 10^2\, b' + 21, \quad b' \in \mathbb{N}$$

zu nehmen mit $10^{-1}\, m \leqslant b < m$. Wir definieren $b = 31\,415\,821$ und haben des weiteren freie Wahl für x_0. Um nicht bei jedem Programmlauf von außen einen Startwert zufügen zu müssen, wird als x_0 oft eine Zahl gewählt, die sich aus der Systemzeit des Computersystems in 1/10 oder 1/100 Sekunden herleitet. Wichtig ist dabei nur, daß x_0 aus einem großen Wertebereich gewählt wird, und daß sich x_0 im Vergleich zur Laufzeit des Programmes schnell ändert, wenn man vermeiden möchte, daß in mehreren Programmläufen hintereinander wegen desselben Startwertes auch jedesmal dieselbe Folge von Pseudozufallszahlen verwendet wird.

Um nun eine Funktion, die die oben definierte Zahlenfolge liefert, in einer Sprache wie C++ zu programmieren, muß man die dort bestehenden Grenzen für ganze Zahlen beachten. Bei der Multiplikation gemäß (4.13) von bx_{n-1} darf kein Überlauf-Fehler auftreten. Daher rechnet man mit den Darstellungen (wir schreiben jetzt der Einfachheit halber x statt x_{n-1}):

$$\begin{aligned} b &= 10^4\, b_1 + b_0 \qquad \text{und} \\ x &= 10^4\, x_1 + x_0\ . \end{aligned}$$

Damit erhält man für das Produkt

$$\begin{aligned} bx &= (10^4\, x_1 + x_0)(10^4\, b_1 + b_0) = \\ &= 10^8\, b_1x_1 + 10^4\,(b_1x_0 + b_0x_1) + b_0x_0\ . \end{aligned}$$

Den ersten Term und die höchstwertigen vier Stellen des zweiten kann man wegen der Rechnung $\pmod{10^8}$ weglassen, und man kann schreiben:

$$bx = (b_1x_0 + b_0x_1 \bmod 10^4)\, 10^4 + b_0x_0\ .$$

Jetzt kann man an die Deklaration (siehe Abbildung 4.5) Implementierung der Klasse `TRandom` gehen, ohne einen overflow-error befürchten zu müssen. Die Methoden der Klasse `TRandom` haben im einzelnen die unterhalb angegebenen Bedeutungen.

```
class TRandom
    {
    private:
        enum
            {
            m  = 100000000,
            m1 = 10000,
            q0 = 5821,
            q1 = 3141
            };
        int p0,p1,x;
    public:
        TRandom();
        TRandom(int s);

        void seed(int s);
        void autoseed(void);

        int next(void);
        FPTYPE next01(void);
    };
```

Abbildung 4.5: Deklaration der Klasse `TRandom`

`TRandom()` und `void autoseed(void)`
: Starten den Zufallsgenerator mit automatisch aus der aktuellen Systemzeit abgeleitetem Startwert; er besteht aus dem Produkt der Register für Sekunden und Millisekunden der aktuellen Systemzeit.

`TRandom(int s)` und `void seed(int s)`
: Starten den Zufallsgenerator mit dem ganzzahligen Startwert `s`.

`int next(void)`
: Liefert eine neue Pseudo-Zufallszahl x, für die $0 \leqslant x < \mathtt{m}$ gilt.

`FPTYPE next01(void)`
: Liefert eine neue Pseudo-Zufallszahl x, umgerechnet auf den Bereich $0 \leqslant x < 1$.

Kapitel 5

Experimentelle Ergebnisse

In diesem Abschnitt präsentieren wir Ergebnisse, die bei Anwendung der Heuristiken aus Kapitel 2 und des genetischen Algorithmus aus Kapitel 4 auf bekannte Testprobleme von Beasley bzw. auf von uns erstellte Gitter-Probleme mit Sensorknoten erzielt wurden.

Die Algorithmen wurden wie angegeben in der Sprache C++ programmiert und mit Hilfe des optimierenden 32-Bit-Compilers „Watcom C++" (Version 10.5) für das Betriebssystem OS/2 übersetzt. Es kam ein Pentium 150-Computer mit einer Speicherkapazität von 32 MB zum Einsatz. Zum Vergleich: auf diesem System dauert die Durchführung des Algorithmus von Floyd-Warshall für einen Graphen mit 100 Knoten und 400 Kanten (Testproblem B18, siehe Abschnitt 5.1) etwa 1/2 Sekunde, für einen Graphen mit 1000 Knoten und 50000 Kanten (Testproblem D20) circa vier Minuten. Die zur Zeitmessung eingesetzten C-Funktionen entsprechen dem ANSI-Sprachstandard und sind daher unabhängig von Systemen und Compilern.

5.1 Testprobleme von Beasley

Beasley hat in [Bea84] und in [Bea89] insgesamt vier Gruppen (B,C,D und E) von Testproblemen für $\text{USTP}_{G,Z}$ eingeführt, welche später vielfach zum Vergleich von Heuristiken und anderen Algorithmen zur (näherungsweisen) Lösung dieser Aufgabe herangezogen wurden; siehe hierzu etwa in neuester Zeit die Aufstellungen von Testergebnissen in den Arbeiten von Esbensen ([Esb95]) und von Koch und Martin ([KoM96]). Dateien, welche die UST-Probleme von Beasley beschreiben, sind Teil der OR-Library des Imperial College in London. Diese Sammlung enthält eine Vielzahl von Problemen aus dem Bereich der Operations Research; ein Einstieg in die Library ist unter der Adresse

`http://mscmga.ms.ic.ac.uk/info.html`

möglich.

Wir haben unsere Methoden, die natürlich insbesondere zur approximativen Lösung von $\text{USTP}_{G,Z}$ geeignet sind, an den Problemen von Beasley ausprobiert, um den Zeitverbrauch mit anderen bekannten Methoden für das Steiner-Problem vergleichen zu können.

5.1.1 Definition der Testgraphen

Die Testprobleme für $USTP_{G,Z}$ wurden von Beasley gemäß dem Schema von Aneja ([Ane80]) erzeugt, indem zunächst ein aufspannender Baum über alle Knoten, die im Graphen $G = (V, E, d, g)$ enthalten sein sollen, gelegt wurde; weitere Kanten wurden dann zufällig hinzugefügt (damit blieb G zusammenhängend) und allen Kanten schließlich zufällige Gewichte aus dem Bereich $\{1, \ldots, 10\}$ zugewiesen ([Bea89]).

Die vier Gruppen B,C,D und E unterscheiden sich in der Größe der enthaltenen Graphen und im Anteil der speziellen Knoten an der jeweiligen Gesamtzahl von Knoten. Genaue Informationen über diese Daten sind den ersten fünf Spalten der Tabellen 5.9 und 5.10 zu entnehmen.

Die Problemgruppe B enthält 18 UST-Probleme mit $|V| \in \{50, 75, 100\}$, einer Anzahl an speziellen Knoten von $|V|/6$, $|V|/4$ bzw. $|V|/2$ und durchschnittlichen Knotengraden von je 2.5 bzw. 4.0.

Die drei restlichen Gruppen C,D und E enthalten jeweils 20 Graphen nur einer einzigen Problemgröße, nämlich Graphen mit 500, 1000 und 2500 Knoten. Die durchschnittlichen Knotengrade sind mit 2.5, 4, 10 bzw. 50 angegeben, und für die Menge der speziellen Knoten gilt:

$$|Z| \in \{5, 10, |V|/6, |V|/4, |V|/2\} .$$

Wegen der extremen Größe der Dateien haben wir die Probleme e16–e20 nicht getestet; wir führen deshalb in den folgenden Tabellen nur die Probleme e01–e15 der Gruppe E auf.

5.1.2 Heuristik DNIH

Die DNI-Heuristik wurde dreimal angewandt, nämlich in ihrer Reinform DNIH (d.h. nur die Schritte 1–3) und in der Form mit den zusätzlichen Schritten 4 und 5, einmal unter Verwendung des Algorithmus von Prim zur Bestimmung der minimalen aufspannenen Bäume in Schritt 4 (DNIH(P)), und einmal unter Verwendung des Algorithmus von Kruskal (DNIH(K)).

Die DNI-Heuristik konnte alle ihr vorgelegten Probleme von Beasley näherungsweise lösen.

Der gemessene Zeitverbrauch betrug immer wesentlich weniger als eine Sekunde. Die Probleme B waren stets in weniger als 0.05 Sekunden bearbeitet, die Probleme C in weniger als 0.1 Sekunden; Ausnahmen bildeten nur c16 und c20 mit je 0.2 Sekunden benötigter Bearbeitungszeit. Auch die Probleme D bis einschließlich d15 verbrauchten weniger als 0.1 Sekunden; die längeste Zeit dieser Gruppe wurde für d20 mit 0.3 Sekunden bei Bearbeitung durch den Algorithmus DNIH(K) gemessen. Die Probleme der Gruppe E mit den größten Anteilen an speziellen Knoten brauchten bei Bearbeitung durch die Heuristik DNIH(K), die insgesamt jeweils am meisten Zeit beanspruchte, maximal 0.34 Sekunden bis zur Ausgabe der Näherung.

Abweichung	Problemgruppen B	C	D	E
= 0%	38.8 %	5.0 %	5.0 %	0.0 %
< 1%	55.6 %	20.0 %	15.0 %	20.0 %
< 2%	66.7 %	30.0 %	35.0 %	26.7 %
< 5%	72.2 %	60.0 %	65.0 %	53.3 %
maximal	8.5 %	18.2 %	15.4 %	19.1 %

Tabelle 5.1: Gruppierung der Abweichungen vom Optimum für DNIH

Esbensen erzielt mit einer iterierten Version von DNH, die er leider nicht näher beschreibt, bei vorheriger Reduktion der zu bearbeitenden Graphen zwar bessere Ergebisse, doch erstaunt ein Vergleich der Laufzeiten. Während hier Bearbeitungszeiten in einer Größenordnung von 0.1 Sekunden gemessen werden, gibt Esbensen in seiner noch aktuellen Arbeit von 1995 Laufzeiten zwischen 100 und 100,000 (sic!) Sekunden an.

Wir vergleichen die beste jeweils von DNIH, DNIH(P) und DNIH(K) gelieferte Lösung mit der Optimallösung und erhalten für die Abweichungen die in Tabelle 5.1 aufgezeichneten Werte.

Aus den Tabellen 5.9 und 5.10 ersieht man, daß die Anwendung der Schritte 4 und 5 wesentliche Verbesserungen gegenüber dem Ergebnis nach sich ziehen kann, welches die „Reinform" von DNIH liefert; diese Verbesserungen fallen umso mehr auf, je größer $|Z|$ ist. Man betrachte hierzu etwa die Ergebnisse für die Probleme c05, c10, c15, d05, d10, e08 oder e09.

5.1.3 Heuristiken SPH

Die Heuristik SPH-z, die einfach iterierten Versionen SPH-Z und SPH-V und die doppelt iterierte Version SPH-ZZ wurden jeweils unter Anwendung der Schritte 4 und 5 (siehe Seiten 26ff.) getestet. Dabei kam in Schritt 4 der Algorithmus von Prim zur Bestimmung kleinster aufspannender Bäume zum Einsatz.

Die ersten drei Heuristiken waren in der Lage, alle von Beasley vorgeschlagenen Testprobleme bis auf e05, e06, e09, e10, e14 und e15 zu lösen; bei letzteren reichte der Speicherplatz nicht aus, um alle SP-Bäume (siehe Seite 63)–nämlich für jeden speziellen Knoten einen–abzulegen. Die doppelt iterierte Heuristik SPH-ZZ wurde wegen der für große Probleme mit großer Anzahl spezieller Knoten explodierenden Rechenzeit nur bis einschließlich d04 eingesetzt.

Die für jedes Problem gelieferten Ergebnisse sind zusammen mit der jeweils zur Lösung aufgewandten Rechenzeit in Sekunden in den Tabellen 5.11 und 5.12 auf den Seiten 111f. abgedruckt. Die Zeit, die der Algorithmus von Dijkstra auf die Berechnung kürzester Pfade verwendete, ist separat in der letzten Spalte mit der Bezeichnung „SP" ausgewiesen. Bemerkenswert ist, wie genau die tatsächlich gemessenenen mit den gemäß der Komplexitätsbetrachtung zu erwartenden Rechenzeiten übereinstimmen. Für SPH-z wurde bei Anwendung auf d10 z.B. eine Zeit von 0.22 Sekunden gemessen; weil bei diesem Problem 500 spezielle auf insgesamt

Abweichung	SPH-z B	SPH-z C	SPH-z D
= 0%	55.6 %	25.0 %	10.0 %
< 1%	66.7 %	45.0 %	35.0 %
< 2%	72.2 %	65.0 %	50.0 %
< 5%	88.9 %	85.0 %	70.0 %
maximal	6.0 %	11.1 %	10.3 %
	SPH-Z B	SPH-Z C	SPH-Z D
= 0%	88.9 %	50.0 %	30.0 %
< 1%	94.4 %	85.0 %	55.0 %
< 2%	100.0 %	90.0 %	80.0 %
< 5%	100.0 %	95.0 %	95.0 %
maximal	1.6 %	5.3 %	5.8 %
	SPH-V B	SPH-V C	SPH-V D
= 0%	94.4 %	55.0 %	40.0 %
< 1%	100.0 %	85.0 %	70.0 %
< 2%	100.0 %	90.0 %	90.0 %
< 5%	100.0 %	95.0 %	95.0 %
maximal	0.9 %	5.3 %	5.4 %

Tabelle 5.2: Gruppierung der Abweichungen vom Optimum für SP-Heuristiken

1000 Knoten verteilt sind, erwartet man deshalb eine Rechenzeit von etwa 110 Sekunden für SPH-Z bzw. von 220 Sekunden für SPH-V. Tatsächlich brauchten die Algorithmen etwa 103 bzw. 208 Sekunden.

Der Unterschied zwischen den hier gemessenen Zeiten für die SP-Heuristik und den von Esbensen für eine iterierte Version der Heuristik von Takahashi und Matsuyama (Esbensen verweist hier auf [WiM92]) angegebenen Zeiten ist genauso erstaunlich wie im Fall von DNIH. Man entnimmt Tabelle 5.2, daß die Ergebnisse von SPH-Z und SPH-V wesentlich besser sind als diejenigen, die Esbensen ermittelte.

5.1.4 DAA und DAH

In den Tabellen 5.9 und 5.10 sind in den letzten drei Spalten auch die Ergebnisse des dualen Aufstiegsalgorithmus DAA und der dazugehörigen Heuristik, die wir in Abschnitt 4.5 behandelt haben, für die Anwendung auf Beasleys Testprobleme aufgeführt. Die dafür jeweils aufgewendete Zeit läßt sich aus der letzten Tabellenspalte ablesen.

Die beiden Algorithmen von Wong konnten auf fast alle Testprobleme in B, C, D und E erfolgreich angewandt werden. Lediglich bei e05, e10 und e15 machte die Größe der Menge Z in ähnlicher Weise wie zuvor im Fall der SP-Heuristiken die Durchführung von DAA (und demzufolge auch DAH) unmöglich. Dies ist ein Fortschritt gegenüber den wenigen von Wong selbst

$\lvert Z\rvert$ bzw. $\lvert Z\rvert/\lvert V\rvert$	DAA C	DAA D	DAA E	DAH C	DAH D	DAH E
5	1.8 %	7.6 %	6.6 %	14.2 %	31.6 %	7.6 %
10	1.6 %	3.5 %	3.6 %	10.7 %	9.5 %	17.8 %
1/6	7.0 %	10.8 %	11.3 %	8.7 %	11.8 %	6.3 %
1/4	6.2 %	9.2 %	11.5 %	6.4 %	7.1 %	5.2 %
1/2	9.6 %	5.9 %	—	3.4 %	14.9 %	—
insgesamt	5.2 %	7.4 %	8.3 %	8.7 %	12.3 %	9.3 %

Tabelle 5.3: Durchschnittliche Abweichungen vom Optimum für die beiden Algorithmen von Wong

Abweichung	DAA B	DAA C	DAA D	DAH B	DAH C	DAH D
= 0%	44.4 %	20.0 %	5.0 %	44.4 %	5.0 %	5.0 %
< 1%	50.0 %	20.0 %	10.0 %	50.0 %	5.0 %	15.0 %
< 2%	66.7 %	20.0 %	10.0 %	66.7 %	15.0 %	15.0 %
< 5%	88.9 %	55.0 %	35.0 %	83.3 %	40.0 %	45.0 %
maximal	13.6 %	14.4 %	24.2 %	9.1 %	22.2 %	61.5 %

Tabelle 5.4: Gruppierung der Abweichungen vom Optimum für die beiden Algorithmen von Wong

in [Won82] getesteten Beispielen, welche maximal 60 Knoten bei 240 gerichteten Kanten enthielten. Leider vergrößert sich auch der Abstand zwischen Optimum und den von DAA bzw. DAH gelieferten unteren bzw. oberen Schranken, wenn die Größe der Graphen und die Größe von Z wachsen. Jener Abstand war bei Wongs kleinen Problemen praktisch nie vorhanden; die hier erzielten Ergebnisse (abhängig vom Anteil spezieller Knoten in den Graphen der Gruppen C, D und E) sind in Tabelle 5.3 abzulesen.

Gruppieren der Ergebnisse nach Größe der Abweichung vom Optimum wie in Tabelle 5.4 offenbart, daß DAH Ergebnisse liefert, die durchaus noch mit DNIH vergleichbar sind. Gerade die iterierten Versionen von SPH erzeugen aber wesentlich bessere Näherungen. Auch im Hinblick auf die Maximalabweichungen in den verschiedenen Problemgruppen schneidet DNH am schlechtesten ab.

5.1.5 Genetischer Algorithmus

Um die Leistung des genetischen Algorithmus GA mit DKNUX-Crossover-Operator (siehe Kapitel 4 auf Seite 73ff.) im Vergleich zum Algorithmus Esbensens, der nach klassischem Muster mit 2-Punkt-Crossover und Inversions-Operator arbeitet, beurteilen zu können, haben wir GA auf die Gruppe C der ORL-Testprobleme angewandt. Als Heuristik innerhalb des genetischen Algorithmus wurde die Heuristik DNIH(P) verwendet.

Mit einer Populationsgröße $N = 40$ wurden 10 Evolutionen mit je 150 Generationen durchgeführt, d.h. wir erzeugten 10 Folgen $(G(t))_{t\in\{0,\ldots,150\}}$.

Crossover- und Mutationswahrscheinlichkeiten wurden in klassischen Größenordnungen gesetzt zu $P_{\mathrm{DKNUX}} = 0.8$ und $P_{\mathrm{MUT}} = 0.001$. Wir verwendeten $N_{\max} = 0.1N$. Jede fünfte Generation wurden die DKNUX-Wahrscheinlichkeiten neu berechnet, d.h. es war $N_{\mathrm{DKNUX}} = 5$; für diese Berechnungen genauso wie für die Initialisierung der ersten Populationen setzten wir die Parameterwerte in Abhängigkeit vom Verhältnis der Anzahl spezieller Knoten zur Anzahl an Steiner-Knoten $S = V \setminus Z$:

$$
\begin{aligned}
q_{\min} &= \frac{1}{4}\frac{|Z|}{|S|} = P_{\mathrm{Gen1}} \\
q_{\max} &= \min\left\{\frac{3}{4}\frac{|Z|}{|S|}, 1\right\} \\
q' &= \frac{1}{8}\frac{|Z|}{|S|} \; .
\end{aligned}
$$

In der Spalte „$\mathrm{Opt}_{\mathrm{GA}}$“ von Tabelle 5.5 ist die jeweils beste während der 10 Evolutionen erzeugte Näherung angegeben. Für genau 50 % der Probleme wurde das von Esbensen angegebene Optimum erreicht, insgesamt 85 % der Näherungen lagen nicht weiter als 1 % von diesem Optimum entfernt.

In Spalte drei („$\mathrm{Avg}_{\mathrm{GA}}$“) ist der Durchschnitt über die Endergebnisse der je 10 Evolutionen abgedruckt. Die t_1-Spalte gibt über die durchschnittliche Zeit Auskunft, die jeweils zur Berechnung einer einzigen Generation benötigt wurde.

Die letzten Spalten geben die Anteile der Evolutionen an, bei denen nach 50, 100 bzw. 150 Generationen der Wert $\mathrm{Opt}_{\mathrm{GA}}$ erreicht wurde. So konnte z.B. nur eine Evolution für das Problem c09 den Wert 708 liefern; für c12 hingegen wurde $\mathrm{Opt}_{\mathrm{GA}}$ bei jeder der 10 Evolutionen erreicht.

5.1.6 Tabellarische Aufstellung der Ergebnisse

In den vier Tabellen 5.9 bis 5.12 auf den Seiten 109ff. führen wir sowohl die Daten der bearbeiteten Probleme als auch die Ergebnisse, die die von uns dargestellten Algorithmen liefern, auf.

Die Werte in der „Opt“-Spalte sind Optimallösungen, die Esbensen angibt. Diese stimmen mit den von Beasley genannten Ergebnissen zum größten Teil überein. Nur für e07, e12, e13 und e14 scheinen Esbensen neuere Erkenntnisse vorzuliegen; Beasley nämlich konnte diese Probleme lediglich näherungsweise lösen.

Fehlt in einer Spalte für Näherungen ein Wert, d.h. ist stattdessen „—“ eingetragen, so wurde der jeweilige Algorithmus wegen Zeit- oder Speicherplatzmangels nicht auf das Problem angewandt.

Keine Angabe in einer der Zeitspalten bedeutet, daß die Durchführung des Algorithmus weniger als 0.1 Sekunden beanspruchte.

5.2 Selbsterstellte Gitter

Eine zweite Gruppe von Testproblemen wurde vom Autor erstellt; die Graphen, auf denen diese Testbeispiele beruhen, sind realen Gitterstrukturen nachempfunden; im Abschnitt über sie SP-Heuristiken auf den Seiten 30ff.

Name	$\mathrm{Opt}_{\mathrm{GA}}$	$\dot{\mathrm{Avg}}_{\mathrm{GA}}$	$\dot{t}_1$	$\mathrm{Opt}_{\mathrm{GA}}$ nach … Gen'en 50	100	150
c01	85	85.20	0.39	40 %	90 %	90 %
c02	144	144.00	0.81	100 %	100 %	100 %
c03	755	756.20	0.82	0 %	10 %	50 %
c04	1081	1084.20	0.63	0 %	0 %	10 %
c05	1579	1579.40	0.75	30 %	50 %	70 %
c06	55	55.00	0.49	100 %	100 %	100 %
c07	102	102.90	0.55	70 %	70 %	70 %
c08	511	511.90	0.79	0 %	20 %	50 %
c09	708	711.10	0.90	0 %	10 %	10 %
c10	1093	1093.90	0.92	30 %	40 %	50 %
c11	32	32.10	0.93	90 %	90 %	90 %
c12	46	46.00	0.86	50 %	80 %	100 %
c13	262	263.80	1.25	10 %	10 %	20 %
c14	325	327.60	1.20	0 %	10 %	10 %
c15	556	556.20	1.66	60 %	70 %	80 %
c16	11	11.50	2.34	20 %	40 %	50 %
c17	19	19.00	2.11	100 %	100 %	100 %
c18	119	119.40	3.01	30 %	60 %	60 %
c19	150	151.00	3.02	20 %	20 %	20 %
c20	267	267.60	4.50	40 %	40 %	40 %

Tabelle 5.5: Ergebnisse des genetischen Algorithmus

und an anderen Stellen dieser Arbeit sind bereits Graphen aufgetreten, deren Struktur derjenigen der jetzt zur Diskussion stehenden Graphen entsprach. Wir werden zwei spezielle Gitter näher betrachten, eines davon in der Ebene (daher von einem planaren Graphen modelliert), das andere in drei Dimensionen.

5.2.1 Definition der Testgraphen

Die Testgruppen basieren auf zwei Referenzgraphen ($k = 1, 2$)

$$_kR = (_kV, {}_kE, d, 0) \ ,$$

die mit Hilfe der folgenden Definitionen ($i \in \{1, 2, 3\}$) festgelegt werden:

$$\begin{aligned} E_{i,V} &:= \{\, \{x, x + e_i\} \mid x, x + e_i \in V \,\} \\ E_{i,j,V} &:= \{\, \{x, x + e_i + e_j\} \mid x, x + e_i + e_j \in V \,\} \ ; \end{aligned}$$

hierbei seien e_i für $i \in \{1,2,3\}$ die Einheitsvektoren im $\mathbb{R}^3$. Wir definieren jetzt

$$
\begin{aligned}
{}_1V &:= \{1,\ldots,25\} \times \{1,\ldots,25\} \times \{1\} \\
{}_1E &:= \bigcup_{i=1}^{3} E_{i,{}_1V} \cup E_{1,2,{}_1V} \\
{}_2V &:= \{1,\ldots,10\} \times \{1,\ldots,5\} \times \{1,\ldots,20\} \\
{}_2E &:= \bigcup_{i=1}^{3} E_{i,{}_2V} \cup E_{1,2,{}_2V} \cup E_{1,3,{}_2V} \ .
\end{aligned}
$$

${}_1R$ interpretiert man als 25×25-Gitter in der Ebene mit 625 Knoten und 4704 Kanten, während ${}_2R$ dagegen ein dreidimensionales $10 \times 5 \times 20$-Gitter mit 1000 Knoten und 11600 Kanten modelliert. Für die Bewertung auf den Kanten der Graphen gilt

$$
d_e = \begin{cases} 1.0 & \text{falls } e \in \bigcup_{i=1}^{3} E_{i,\mathbb{Z}^3} \\ 1.4 & \text{sonst .} \end{cases}
$$

Jede der auf ${}_kR$ ($k \in \{1,2\}$) aufbauenden Testgruppen besteht aus 27 USTP'-Problemen, die sich voneinander unterscheiden durch

- den Anteil p_{R} der im Graphen zufällig (durch Multiplikation) mit einer neuen Bewertung versehenen Kanten, nämlich 0%, 10% oder 20%,
- den Anteil p_{SP} der speziellen Knoten bezüglich aller Knoten des Graphen, nämlich 1%, 5% oder 10%, und
- den Anteil p_{SN} der Sensorknoten bezüglich aller Knoten des Graphen, nämlich 1%, 5% oder 10%.

Die exakte Parameterzuordnung und die Reihenfolge, in denen die Probleme mit Namen ${}_1R_{111}$, ${}_1R_{112}$ bzw. ${}_2R_{111}$, ${}_2R_{112}$ usw. in den später folgenden Tabellen auftreten werden, entnehme man der Aufstellung in Tabelle 5.6.

5.2.2 Bewertung auf den Knoten

Wir benutzen die folgende Knotenbewertung, um in die von uns betrachteten Referenzgraphen der Gestalt $G' = (V,E,d,0)$ die Information über die Sensorknotenmenge Z' wie in Kapitel 1 auf Seite 7 gefordert einfließen zu lassen:

$$
g : V \to \mathbb{R}_{\geqslant 0};\, g_v := d_{\min(G)}(v,Z')^2 \ .
$$

Wir wenden dann die besprochenen Algorithmen auf das Problem USTP'$_{G,Z}$ an, wobei der Graph G als $G := (V,E,d,g)$ definiert sei. Die durch diese neue Knotenbewertung aus ${}_kR$ ($k \in \{1,2\}$) entstehenden Testprobleme nennen wir ${}_kG$ ($k \in \{1,2\}$).

Name	${}_1R, {}_1G$ n_X	n_Y	n_Z	${}_2R, {}_2G$ n_X	n_Y	n_Z	p_R	p_{SP}	p_{SN}
111	25	25	1	10	5	20	0.00	0.01	0.01
112	25	25	1	10	5	20	0.00	0.01	0.05
113	25	25	1	10	5	20	0.00	0.01	0.10
121	25	25	1	10	5	20	0.10	0.01	0.01
122	25	25	1	10	5	20	0.10	0.01	0.05
123	25	25	1	10	5	20	0.10	0.01	0.10
131	25	25	1	10	5	20	0.20	0.01	0.01
132	25	25	1	10	5	20	0.20	0.01	0.05
133	25	25	1	10	5	20	0.20	0.01	0.10
211	25	25	1	10	5	20	0.00	0.05	0.01
212	25	25	1	10	5	20	0.00	0.05	0.05
213	25	25	1	10	5	20	0.00	0.05	0.10
221	25	25	1	10	5	20	0.10	0.05	0.01
222	25	25	1	10	5	20	0.10	0.05	0.05
223	25	25	1	10	5	20	0.10	0.05	0.10
231	25	25	1	10	5	20	0.20	0.05	0.01
232	25	25	1	10	5	20	0.20	0.05	0.05
233	25	25	1	10	5	20	0.20	0.05	0.10
311	25	25	1	10	5	20	0.00	0.10	0.01
312	25	25	1	10	5	20	0.00	0.10	0.05
313	25	25	1	10	5	20	0.00	0.10	0.10
321	25	25	1	10	5	20	0.10	0.10	0.01
322	25	25	1	10	5	20	0.10	0.10	0.05
323	25	25	1	10	5	20	0.10	0.10	0.10
331	25	25	1	10	5	20	0.20	0.10	0.01
332	25	25	1	10	5	20	0.20	0.10	0.05
333	25	25	1	10	5	20	0.20	0.10	0.10

Tabelle 5.6: Namen und Parameterwerte der Gitter-Testprobleme

5.2.3 Meßgrößen

Wir vergleichen die Ergebnisse, die die verschiedenen Programme liefern, nicht nur anhand der Werte $\mathrm{res}_{h(G,Z)}$ für eine Heuristik h, sondern auch anhand zweier weiterer nun folgender Meßgrößen.

Es sei $T' := (V', E', d, g) := h(G, Z)$ der von einem Algorithmus h gelieferte Baum, welcher eine Näherungslösung für $\mathrm{USTP'}_{G,Z}$ darstellt. Als Meßgrößen für die Qualität von T' betrachten wir den größten Abstand d'_1 und den mittleren Abstand d'_2 des Baumes T' in G' von den Knoten der Menge Z':

$$d'_1 := \max\{d_{\min(G')}(z, V') \mid z \in Z'\} ,$$
$$d'_2 := \frac{1}{|Z'|} \sum_{z \in Z'} d_{\min(G')}(z, V') .$$

Mit Hilfe dieser Größen werden wir beurteilen können, inwieweit jeder der Algorithmen die Abstände zu den Sensorknoten verkleinern konnte, indem wir sie nämlich mit den entsprechenden Meßgrößen für Näherungen $h(G', Z)$ auf den Referenzgraphen ohne die Knotenbewertungen gemäß Unterabschnitt 5.2.2 vergleichen.

	DNIH(P)	SPH-z(P)	GA	DAH
${}_1R$	9.2 %	8.1 %	—	47.0 %
${}_1G$	12.7 %	12.1 %	9.9 %	18.6 %
${}_2R$	13.5 %	11.8 %	—	47.1 %
${}_2G$	9.2 %	8.9 %	5.8 %	19.1 %

Tabelle 5.7: Durchschnittliche Prozentuale Entfernungen von der unteren Schranke

5.2.4 Tabellarische Aufstellung der Ergebnisse

Auf den Seiten 113 bis 116 findet der Leser Tabellen, die die Ergebnisse der Algorithmen DNIH(P), SPH-z(P), der beiden Algorithmen von Wong und des genetische Algorithmus für die Sätze ${}_kR$ und ${}_kG$ ($k \in \{1, 2\}$) unserer Testprobleme zusammenfassen.

Der genetische Algorithmus wurde aus Zeitgründen jeweils einmal mit einer Populationsgröße von $N = 30$ über 120 Generationen mit dem Algorithmus SPH-z(P) als Heuristik durchgeführt. Alle übrigen Parameter blieben gegenüber Unterabschnitt 5.1.5 unverändert.

Die erste Spalte der Tabellen zeigt die vom dualen Aufstiegsalgorithmus gelieferte untere Schranke. Die folgende Gruppe von Spalten gibt die Gewichte der von den Heuristiken bzw. vom genetischen Algorithmus erstellten Näherungenlösungen wieder. Die nächsten Spalten („$g \equiv 0$“) zählen jeweils die Summen der Kantengewichte der Näherungen auf. Deshalb stimmen bei den Problemen der Gruppen ${}_kR$ ($k \in \{1, 2\}$) diese vier Spalten mit den Spalten „USTP"“ überein. Die Werte der Meßgrößen d'_1 und d'_2 folgen in den letzten beiden Spaltengruppen.

5.2.5 Vergleich der Ergebnisse

Zunächst wollen wir einen Eindruck darüber erhalten, wie weit die Gewichte der von den Heuristiken gelieferten Näherungslösungen von der mit DAA ermittelten unteren Schranke jeweils entfernt liegen. Tabelle 5.7 gibt das Mittel dieser Abstände prozentual bezüglich der unteren Schranke an. Zum Vergleich: bei Anwendung von DNIH(P) bzw. SPH-z(P) auf die Testprobleme der Gruppe D von Beasley, die ähnlich groß sind wie die hier betrachteten Gitter, kommt man auf entsprechende Werte von 13.7 % bzw. 11.9 %.

Vergleicht man die von den Heuristiken gelieferten Näherungslösungen untereinander, so kommt man zur Erkenntnis, daß DAH die schlechtesten Ergebnisse liefert. Der genetische Algorithmus erzeugt zwangsläufig (wegen des „seeding“, siehe Seite 75), immer mindestens so gute Näherungen wie SPH-z(P). SPH-z(P) hat im Vergleich zu DNIH(P) öfter die Nase vorn: bei Anwendung ersterer Heuristik auf unsere Testgruppen erhält man für ${}_1R$ bzw. ${}_2R$ 24 bzw. 26-mal mindestens so gute Ergebnisse wie mit DNIH(P), für ${}_1G$ und ${}_2G$ jeweils 16-mal.

Wir legen jetzt unser Augenmerk auf die Gewichte der Näherungen, wenn man nur die Kantenbewertungen beachtet (Spalten „$g \equiv 0$“), und auf die beiden in Unterabschnitt 5.2.3 definierten Messgrößen. In Tabelle 5.8 sind die prozentualen Veränderungen dieser Werte für ${}_kG$ gegenüber

		DNIH(P)	SPH-z(P)	DAH
d_1'	$_1G$	-15.8 %	-15.9 %	-18.7 %
	$_2G$	- 6.7 %	-10.0 %	- 9.5 %
d_2'	$_1G$	-39.1 %	-39.8 %	-48.7 %
	$_2G$	-29.1 %	-33.3 %	-39.1 %
$g \equiv 0$	$_1G$	20.4 %	20.5 %	1.7 %
	$_2G$	10.3 %	14.9 %	1.7 %

Tabelle 5.8: Veränderungen nach Einführung der Knotenbewertung aus Unterabschnitt 5.2.2

$_kR$ ($k \in \{1, 2\}$) angegeben. Es ist klar, daß die Verringerung von d_1' und d_2' stets eine Vergrößerung des Gewichts der Näherungslösung nach sich zieht. Von der eingeführten Knotenbewertung hängt es ab, wie sehr die Meßgrößen auf Kosten des Baumgewichts verbessert werden können.

5.2.6 Exemplarische Betrachtungen

Zum Abschluß stellen wir einige Näherungslösungen, die wir für die Probleme $_1R_{332}$ bzw. $_1G_{332}$ durch Anwendung von SPH-z(P) und mit Hilfe des genetischen Algorithmus erhalten haben, dar. In den Abbildungen sind keine Kanten dargestellt, sofern sie nicht zur in den Ausgangsgraphen eingebetteten Näherung gehören. Man denke sich alle Knoten des Ausgangsgraphen gemäß Unterabschnitt 5.2.1 als durch waagerechte, senkrechte und diagonale Kanten verbunden. Die dick gezeichneten Knoten des Graphen sind die Sensorknoten.

Die Abbildung 5.1 zeigt das Ergebnis von SPH-z(P) nach Anwendung auf $_1R_{332}$. Die Sensorknoten sind für die Heuristik noch „unsichtbar“; erst nach Hinzufügen der Bewertung auf den Knoten erhält man das Ergebnis aus Abbildung 5.2, das diese Knoten offensichtlich berücksichtigt. Die Kanten dieser Näherungslösung haben zwar zusammen ein Gewicht von 192.75 (statt 127.42), doch der mittlere Abstand der Sensorknoten vom Baum beträgt nurmehr 0.85 statt vorher 1.89. Der maximale Abstand hat sich von 4.80 auf 4.20 verringert. Die Anwendung des genetischen Algorithmus vergrößert die Beträge der Meßgrößen zwar wieder, doch schrumpft die Summe der Gewichte aller Kanten auf 172.25; die dazugehörige Näherungslösung zeigt Abbildung 5.3.

Um das obige Ergebnis des genetischen Algorithmus zu prüfen, wurde $_1G_{332}$ auch einmal mit etwas mehr Aufwand behandelt. Wir führten 10 Evolutionen mit je 200 Generationen zu je 100 Individuen durch und verwendeten die Heuristik DNIH(P) zur Dekodierung. Die untere Reihe von Punkten in Abbildung 5.4 zeigt den Durchschnitt über die besten Individuen jeder der 10 Evolutionen, die bis zur nach rechts abgetragenen Anzahl von Generationen jeweils erreicht wurden, die obere Reihe zeigt den Durchschnitt über alle Individuen, die in der entsprechenden Generation enthalten waren.

Es wird offenbar, daß der Wert 1055.46 aus Tabelle 5.13 sehr gut mit dem Durchschnitt über die besten Individuen nach 150 Generationen über-

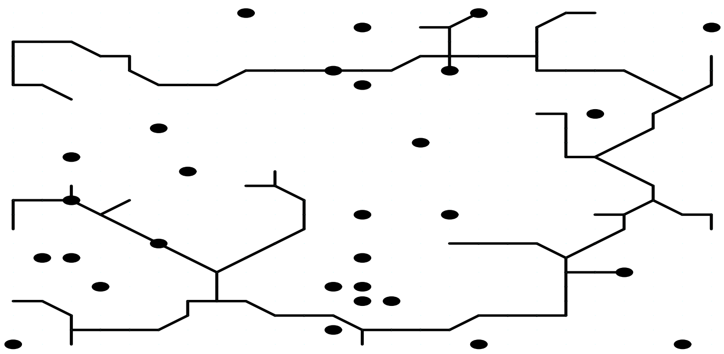

Abbildung 5.1: Ergebnis von SPH-z(P) für ${}_1R_{332}$

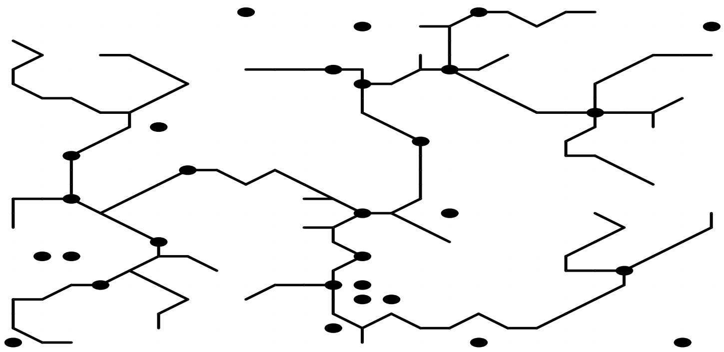

Abbildung 5.2: Ergebnis von SPH-z(P) für ${}_1G_{332}$

einstimmt, obwohl er mit anderen Parametern und einer anderen Heuristik innerhalb des genetischen Algorithmus erhalten wurde. Auch die Entscheidung, nach 150 Generationen abzubrechen, scheint vernünftig gewesen zu sein, denn später sind gemäß der Abbildung keine großen Verbesserungen mehr zu erwarten.

Durchschnittlich brauchte die Erzeugung von 30 Individuen bei Verwendung von DNIH(P) 1.7 Sekunden; wird stattdessen SPH-z(P) eingesetzt, reduziert sich diese Zeit auf etwa 0.4 Sekunden, allerdings muß dann vorher der Algorithmus von Floyd-Warshall zur Bestimmung aller kürzester Pfade durchgeführt werden, was 102 Sekunden benötigte.

5.3 Fazit aus den Tests

Wegen der Leistung heute allgemein verfügbarer Computersysteme können modifizierte Steiner-Probleme, wie sie in der vorliegenden Arbeit im Mittelpunkt standen, effizient näherungsweise gelöst werden. Dabei steht die Heuristik DNIH, was Laufzeit und Speicheraufwand betrifft, an der Spitze. Will man einen Kompromiß eingehen zwischen Laufzeitverhalten und Spei-

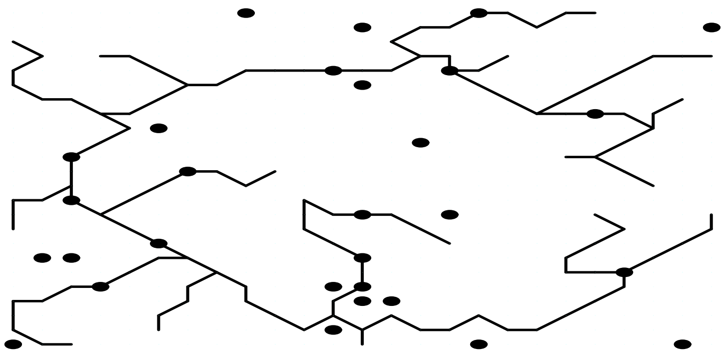

Abbildung 5.3: Ergebnis des genetischen Algorithmus für ${}_1G_{332}$

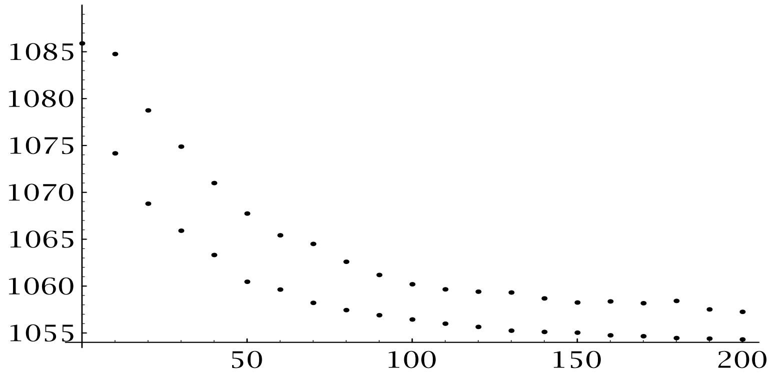

Abbildung 5.4: Durchschnittliche Entwicklung (10 Evolutionsläufe) des genetischen Algorithmus für ${}_1G_{332}$ mit 200 Generationen, 100 Individuen

cherverbrauch einerseits und der Ergebnisqualität andererseits, so denke man an einen Einsatz von SPH-Z.

Genetische Algorithmen liefern im Vergleich zu den Heuristiken die besten Ergebnisse, sowohl im Hinblick auf die Baumgrößen als auch auf die Meßgrößen. Natürlich muß man dafür längere Laufzeiten in Kauf nehmen. Betrachtet man allerdings Abbildung 5.4, so zeigt sich, daß schon die Durchführung weniger Generationen (etwa 50) spürbare Verbesserungen erbringt, was aber für obiges Problem mit 30 Individuen und DNIH als Heuristik auch schon 85 Sekunden dauern würde.

Will man SPH-z als Heuristik einsetzten, muß man vorher alle kürzesten Pfade im Graphen errechnen, verkürzt damit aber die zur Erstellung einer Generation notwendige Zeit (in obigem Beispiel auf 0.4 Sekunden). Dies kann sich durchaus lohnen, wenn verschiedene Steiner-Probleme mit denselben Sensorknoten und wechselnden speziellen Knoten gelöst werden sollen, oder wenn in mehreren Evolutionen hintereinander dasselbe Problem behandelt wird, um eine möglichst gute Näherung zu erreichen. Dann nämlich können die einmal errechneten Pfadinformationen in mehreren Durchläufen wiederverwendet werden.

Problemdaten					Näherungen			Dualer Aufstieg		
Name	$\|V\|$	$\|E\|$	$\|Z\|$	Opt	DNIH	DNIH(P)	DNIH(K)	DAA	DAH	Zeit
b01	50	126	9	82	82	82	82	82	82	—
b02	50	126	13	83	90	90	88	83	83	—
b03	50	126	25	138	140	140	138	138	138	—
b04	50	200	9	59	64	64	64	57	59	—
b05	50	200	13	61	62	61	61	58	62	—
b06	50	200	25	122	128	124	124	122	122	—
b07	75	188	13	111	111	111	111	111	111	—
b08	75	188	19	104	104	104	104	104	104	—
b09	75	188	38	220	221	221	221	213	220	—
b10	75	300	13	86	98	91	91	86	91	—
b11	75	300	19	88	92	92	92	87	96	—
b12	75	300	38	174	174	174	174	174	175	—
b13	100	250	17	165	175	175	175	162	174	—
b14	100	250	25	235	237	237	237	203	241	—
b15	100	250	50	318	325	318	318	300	324	—
b16	100	400	17	127	137	136	136	124	133	—
b17	100	400	25	131	132	132	132	129	134	—
b18	100	400	50	218	226	221	221	216	221	—
c01	500	1250	5	85	88	88	88	82	85	0.34
c02	500	1250	10	144	144	144	144	138	146	0.40
c03	500	1250	83	754	778	768	768	700	800	0.72
c04	500	1250	125	1079	1116	1110	1110	1022	1126	0.82
c05	500	1250	250	1579	1602	1586	1586	1472	1607	1.15
c06	500	2000	5	55	60	60	60	53	66	0.53
c07	500	2000	10	102	114	114	114	102	113	0.31
c08	500	2000	83	509	532	524	522	479	536	0.94
c09	500	2000	125	707	732	725	725	669	740	1.03
c10	500	2000	250	1093	1124	1104	1105	936	1116	1.79
c11	500	5000	5	32	37	37	37	32	38	2.19
c12	500	5000	10	46	49	49	49	45	50	0.87
c13	500	5000	83	258	275	271	270	246	280	1.84
c14	500	5000	125	323	343	337	337	315	330	1.44
c15	500	5000	250	556	570	558	557	484	584	2.54
c16	500	25000	5	11	13	13	13	11	13	7.97
c17	500	25000	10	18	19	19	19	18	22	6.94
c18	500	25000	83	113	129	123	124	101	130	17.07
c19	500	25000	125	146	163	154	155	129	167	8.43
c20	500	25000	250	267	269	268	268	256	279	8.69

Tabelle 5.9: Sätze B und C von Beasleys Testproblemen

Problemdaten					Näherungen			Dualer Aufstieg		
Name	$\|V\|$	$\|E\|$	$\|Z\|$	Opt	DNIH	DNIH(P)	DNIH(K)	DAA	DAH	Zeit
d01	1000	2500	5	106	107	107	107	100	131	1.03
d02	1000	2500	10	220	237	237	237	219	230	1.60
d03	1000	2500	167	1565	1648	1606	1609	1517	1603	2.54
d04	1000	2500	250	1935	2007	1969	1970	1853	1978	3.00
d05	1000	2500	500	3250	3313	3264	3264	3005	3276	4.13
d06	1000	4000	5	67	71	71	71	58	81	3.60
d07	1000	4000	10	103	116	116	116	103	103	0.97
d08	1000	4000	167	1072	1150	1123	1123	1013	1138	5.06
d09	1000	4000	250	1448	1547	1498	1497	1374	1478	4.15
d10	1000	4000	500	2110	2169	2144	2143	1986	2131	5.63
d11	1000	10000	5	29	31	31	31	28	35	6.57
d12	1000	10000	10	42	44	44	44	40	47	3.34
d13	1000	10000	167	500	530	519	519	448	543	6.78
d14	1000	10000	250	667	713	686	685	621	704	8.94
d15	1000	10000	500	1116	1153	1133	1134	1031	1140	10.22
d16	1000	50000	5	13	15	15	15	12	21	55.78
d17	1000	50000	10	23	23	23	23	21	28	32.18
d18	1000	50000	167	223	258	245	247	169	290	72.25
d19	1000	50000	250	310	352	332	329	246	367	47.31
d20	1000	50000	500	537	544	543	543	523	548	33.69
e01	2500	6250	5	111	125	125	125	105	118	8.87
e02	2500	6250	10	214	255	255	255	203	265	9.56
e03	2500	6250	417	4013	4222	4159	4156	3248	4254	24.50
e04	2500	6250	625	5101	5313	5191	5189	4821	5198	24.68
e05	2500	6250	1250	8128	8306	8173	8173	—	—	—
e06	2500	10000	5	73	86	86	84	71	83	21.31
e07	2500	10000	10	(157)	165	165	165	140	172	23.12
e08	2500	10000	417	2640	2819	2746	2742	2462	2758	27.69
e09	2500	10000	625	3604	3802	3702	3700	2921	3853	43.53
e10	2500	10000	1250	5600	5759	5650	5651	—	—	—
e11	2500	25000	5	34	39	39	39	30	35	37.63
e12	2500	25000	10	(67)	75	75	75	66	75	18.81
e13	2500	25000	417	(1290)	1374	1345	1348	1175	1386	56.75
e14	2500	25000	625	(1736)	1822	1789	1790	1554	1853	73.69
e15	2500	25000	1250	2784	2862	2809	2808	—	—	—

Tabelle 5.10: Sätze D und E von Beasleys Testproblemen

Problemdaten					Näherungen				Zeitbedarf				
Name	$\|V\|$	$\|E\|$	$\|Z\|$	Opt	SPH-z	SPH-Z	SPH-V	SPH-ZZ	SPH-z	SPH-Z	SPH-V	SPH-ZZ	SP
b01	50	126	9	82	82	82	82	82	—	—	—	—	—
b02	50	126	13	83	88	83	83	83	—	—	—	—	—
b03	50	126	25	138	138	138	138	138	—	—	—	—	—
b04	50	200	9	59	62	59	59	59	—	—	—	—	—
b05	50	200	13	61	61	61	61	61	—	—	—	—	—
b06	50	200	25	122	124	124	122	122	—	—	—	0.34	—
b07	75	188	13	111	111	111	111	111	—	—	—	—	—
b08	75	188	19	104	104	104	104	104	—	—	—	0.19	—
b09	75	188	38	220	220	220	220	220	—	—	0.13	1.44	—
b10	75	300	13	86	90	86	86	86	—	—	—	—	—
b11	75	300	19	88	88	88	88	88	—	—	—	0.18	—
b12	75	300	38	174	174	174	174	174	—	—	0.12	1.46	—
b13	100	250	17	165	165	165	165	165	—	—	0.12	0.19	—
b14	100	250	25	235	241	235	235	235	—	—	0.16	0.50	—
b15	100	250	50	318	318	318	318	318	—	—	0.28	3.75	—
b16	100	400	17	127	133	127	127	127	—	—	0.13	0.19	—
b17	100	400	25	131	132	131	131	131	—	—	0.16	0.53	—
b18	100	400	50	218	220	220	220	218	—	0.16	0.31	4.06	—
c01	500	1250	5	85	85	85	85	85	—	—	1.50	—	—
c02	500	1250	10	144	144	144	144	144	—	—	1.53	—	—
c03	500	1250	83	754	763	761	759	757	—	1.03	6.53	46.38	0.37
c04	500	1250	125	1079	1090	1087	1084	1082	—	2.62	10.84	175.69	0.57
c05	500	1250	250	1579	1582	1581	1581	1580	—	13.57	27.37	1748.90	1.16
c06	500	2000	5	55	55	55	55	55	—	—	1.50	—	—
c07	500	2000	10	102	102	102	102	102	—	—	1.63	0.19	—
c08	500	2000	83	509	516	511	509	509	—	1.06	6.56	47.32	0.43
c09	500	2000	125	707	715	713	713	711	—	2.06	10.66	173.62	0.69
c10	500	2000	250	1093	1098	1095	1094	1093	—	13.56	27.50	1762.85	1.35
c11	500	5000	5	32	34	32	32	32	—	—	1.72	—	—
c12	500	5000	10	46	48	46	46	46	—	—	1.93	0.22	—
c13	500	5000	83	258	268	262	261	260	—	1.10	6.91	49.75	0.59
c14	500	5000	125	323	331	325	325	324	—	2.50	10.31	166.05	0.90
c15	500	5000	250	556	560	556	556	556	—	14.12	28.50	1833.69	1.85
c16	500	25000	5	11	11	11	11	11	—	—	3.91	—	—
c17	500	25000	10	18	20	18	18	18	—	—	4.09	0.38	0.16
c18	500	25000	83	113	124	119	119	117	—	1.46	9.31	67.56	1.44
c19	500	25000	125	146	153	152	152	150	—	3.31	13.68	222.65	2.15
c20	500	25000	250	267	268	267	267	267	—	16.75	34.19	2341.06	4.34

Tabelle 5.11: Sätze B und C von Beasleys Testproblemen

Problemdaten					Näherungen				Zeitbedarf				
Name	$\lvert V \rvert$	$\lvert E \rvert$	$\lvert Z \rvert$	Opt	SPH-z	SPH-Z	SPH-V	SPH-ZZ	SPH-z	SPH-Z	SPH-V	SPH-ZZ	SP
d01	1000	2500	5	106	107	106	106	106	—	—	6.15	—	—
d02	1000	2500	10	220	232	220	220	220	—	—	6.84	0.34	0.12
d03	1000	2500	167	1565	1591	1587	1587	1585	—	7.00	42.78	618.31	1.75
d04	1000	2500	250	1935	1958	1947	1947	1945	—	17.97	72.69	2362.84	2.60
d05	1000	2500	500	3250	3258	3256	3253	—	0.28	108.25	219.00	—	5.85
d06	1000	4000	5	67	71	70	67	—	—	—	6.44	—	—
d07	1000	4000	10	103	103	103	103	—	—	—	5.91	—	0.12
d08	1000	4000	167	1072	1104	1093	1093	—	—	6.81	41.00	—	2.13
d09	1000	4000	250	1448	1486	1472	1470	—	0.10	18.53	74.97	—	3.03
d10	1000	4000	500	2110	2124	2121	2121	—	0.22	102.78	207.69	—	6.25
d11	1000	10000	5	29	31	30	29	—	—	—	7.19	—	—
d12	1000	10000	10	42	42	42	42	—	—	—	7.66	—	0.19
d13	1000	10000	167	500	518	509	509	—	—	6.69	40.72	—	2.75
d14	1000	10000	250	667	678	674	672	—	—	17.56	71.50	—	4.15
d15	1000	10000	500	1116	1127	1122	1122	—	0.22	108.72	221.90	—	8.57
d16	1000	50000	5	13	14	13	13	—	—	—	16.59	—	0.21
d17	1000	50000	10	23	24	23	23	—	—	0.19	17.07	—	0.41
d18	1000	50000	167	223	238	236	235	—	—	8.13	49.96	—	6.50
d19	1000	50000	250	310	333	325	323	—	—	19.35	79.59	—	9.81
d20	1000	50000	500	537	541	538	538	—	0.35	108.69	224.63	—	19.97
e01	2500	6250	5	111	115	115	111	—	—	—	40.62	—	0.15
e02	2500	6250	10	214	231	227	216	—	—	0.15	40.41	—	0.31
e03	2500	6250	417	4013	4096	4081	4080	—	7.00	101.19	603.50	—	17.56
e06	2500	10000	5	73	78	76	73	—	—	—	40.56	—	0.16
e07	2500	10000	10	(157)	157	149	145	—	—	0.19	41.37	—	0.37
e08	2500	10000	417	2640	2702	2689	2687	—	9.03	102.50	603.06	—	17.28
e11	2500	25000	5	34	39	34	34	—	—	—	48.69	—	0.25
e12	2500	25000	10	(67)	69	68	67	—	—	0.19	49.97	—	0.50
e13	2500	25000	417	(1290)	1322	1310	1309	—	7.82	92.84	545.50	—	22.59

Tabelle 5.12: Sätze D und E von Beasleys Testproblemen

USTP'					$g \equiv 0$				größter Abstand d'_1				mittlerer Abstand d'_2			
DAA	DNIH(P)	SPH-z(P)	GA	DAH	DNIH(P)	SPH-z(P)	GA	DAH	DNIH(P)	SPH-z(P)	GA	DAH	DNIH(P)	SPH-z(P)	GA	DAH
653.16	797.04	782.44	769.80	681.20	41.40	41.20	39.20	42.20	10.60	10.60	10.60	10.60	2.60	2.77	2.60	2.60
163.88	193.04	193.04	193.04	165.08	44.20	44.20	44.20	45.40	14.20	14.20	14.20	14.20	3.08	3.04	3.07	2.96
91.28	100.92	97.76	97.24	91.48	38.20	37.00	36.40	36.40	13.60	13.60	13.60	13.60	4.12	4.20	4.26	4.26
656.20	783.07	781.51	758.83	818.48	52.23	52.43	50.60	53.40	9.54	9.54	9.54	9.54	3.12	3.12	2.89	2.79
243.59	253.75	246.99	246.35	245.55	67.60	66.60	66.00	65.20	6.60	7.00	7.00	7.00	1.85	2.00	2.00	2.03
114.07	122.68	120.12	117.67	122.49	58.40	57.60	65.02	67.80	8.00	8.00	8.20	8.20	2.80	2.85	2.36	2.37
954.94	1068.68	1058.17	1018.85	1115.22	87.20	70.00	76.25	88.05	7.40	7.40	7.20	7.20	2.83	2.83	2.00	1.83
166.87	213.94	213.74	210.43	167.07	52.63	52.43	50.63	50.83	11.80	11.80	13.00	13.00	2.24	2.28	2.33	2.29
112.98	122.88	115.24	115.00	139.25	51.20	46.60	52.48	69.61	15.73	15.00	14.00	13.60	4.48	4.19	3.18	2.51
2792.08	2924.88	2917.64	2895.32	3335.84	109.00	108.00	106.60	132.60	1.40	1.40	1.40	1.40	0.57	0.57	0.57	0.23
613.76	642.32	636.16	622.84	692.48	119.40	124.20	116.00	145.20	4.40	4.00	4.00	2.00	1.04	0.85	1.03	0.46
274.80	322.32	324.48	314.12	327.08	113.40	112.60	108.20	120.80	5.60	5.60	5.60	5.60	0.97	0.98	1.08	0.91
2065.42	2266.17	2271.57	2241.04	2721.39	116.02	109.08	111.28	137.14	3.00	3.00	3.00	1.40	1.07	0.73	0.73	0.47
591.46	621.63	626.45	600.45	637.34	118.14	111.19	108.09	113.93	3.76	3.76	6.20	6.20	0.90	0.84	1.67	1.51
237.67	272.26	260.27	256.05	320.75	108.83	107.21	100.18	131.67	4.40	4.80	4.80	4.40	1.20	1.14	1.13	0.72
2425.75	2755.50	2749.64	2658.69	2814.92	133.47	142.16	127.51	136.00	2.40	2.40	2.40	2.40	0.73	0.73	0.73	0.57
534.08	566.63	569.36	556.01	582.20	128.39	126.77	123.58	138.15	2.80	3.80	2.80	2.40	0.52	0.55	0.57	0.39
272.25	299.67	304.49	295.27	329.64	99.69	105.30	93.37	123.66	6.81	6.80	7.40	6.80	1.73	1.58	1.86	1.28
3981.28	4950.15	4952.67	4863.63	6365.48	158.60	159.40	157.80	200.40	2.00	2.00	2.00	1.00	0.33	0.50	0.33	0.33
906.08	993.04	998.24	962.20	1046.44	155.80	161.20	152.00	185.60	2.00	2.00	2.00	2.40	0.52	0.50	0.54	0.20
366.08	404.04	402.68	399.28	455.40	139.60	141.00	138.60	175.60	6.00	6.00	6.00	6.00	1.26	1.27	1.32	0.83
5458.84	5725.46	5674.06	5662.66	6032.37	173.87	172.67	173.27	184.33	1.00	1.00	1.00	1.00	0.50	0.50	0.33	0.17
702.65	741.36	765.43	733.31	784.83	148.21	157.01	147.45	164.69	4.47	4.47	4.47	4.47	0.85	0.68	0.71	0.59
583.78	690.40	691.37	678.81	727.17	163.98	160.14	156.55	197.55	3.84	3.84	3.84	2.80	0.76	0.94	0.85	0.28
1697.98	2097.25	2091.50	2055.40	2395.53	143.57	143.35	144.12	164.93	1.40	1.00	1.00	0.00	0.23	0.50	0.17	0.00
863.57	1085.89	1089.03	1055.46	1325.35	183.33	192.75	172.25	212.19	4.79	4.20	4.79	4.80	1.03	0.85	1.05	0.35
644.90	679.43	684.46	675.85	716.30	149.39	154.80	149.39	177.68	3.40	3.40	3.40	3.40	0.89	0.90	0.93	0.63

Tabelle 5.13: Satz ${}_1G$ der Gitter-Testprobleme

USTP'				$g \equiv 0$			größter Abstand d'_1			mittlerer Abstand d'_2		
DAA	DNIH(P)	SPH-z(P)	DAH	DNIH(P)	SPH-z(P)	DAH	DNIH(P)	SPH-z(P)	DAH	DNIH(P)	SPH-z(P)	DAH
35.40	36.60	36.20	41.20	36.60	36.20	41.20	10.60	10.60	10.60	3.17	3.17	3.30
34.40	34.40	34.40	35.60	34.40	34.40	35.60	17.40	17.40	16.80	4.44	4.28	4.36
31.60	35.00	34.60	43.20	35.00	34.60	43.20	14.80	14.80	13.60	4.30	4.35	4.39
41.86	48.49	45.29	67.89	48.49	45.29	67.89	9.54	9.54	9.54	4.47	4.55	3.89
53.02	57.63	57.30	69.78	57.63	57.30	69.78	7.80	7.80	7.80	2.93	2.91	2.47
43.99	52.36	51.75	64.26	52.36	51.75	64.26	8.00	8.00	8.00	3.58	3.47	2.93
47.94	52.38	50.75	67.83	52.38	50.75	67.83	7.40	7.40	7.40	4.57	4.67	3.97
44.42	46.01	46.01	62.80	46.01	46.01	62.80	13.00	13.00	13.00	2.83	2.85	2.39
39.38	41.77	41.37	47.98	41.77	41.37	47.98	16.13	16.13	16.13	4.96	4.96	4.82
81.20	94.00	93.40	155.40	94.00	93.40	155.40	3.00	3.00	3.80	1.13	1.37	1.20
94.60	100.40	100.40	163.80	100.40	100.40	163.80	5.60	4.80	4.00	2.00	1.93	1.27
89.00	93.00	92.00	111.40	93.00	92.00	111.40	7.60	7.20	7.00	1.77	1.72	1.76
92.50	95.72	95.92	146.69	95.72	95.92	146.69	4.00	4.80	4.00	2.00	1.97	1.50
81.86	85.49	85.29	110.53	85.49	85.29	110.53	6.20	6.20	6.20	2.04	2.01	1.90
86.12	92.71	91.99	156.40	92.71	91.99	156.40	4.80	4.80	4.40	1.63	1.65	1.18
88.94	94.12	91.83	129.84	94.12	91.83	129.84	2.40	2.40	2.40	1.13	1.13	0.90
97.91	108.01	105.16	145.64	108.01	105.16	145.64	5.40	5.40	5.00	1.66	1.61	1.38
81.74	86.85	85.98	110.18	86.85	85.98	110.18	7.40	7.40	7.40	2.19	2.22	2.02
128.20	133.80	133.80	154.60	133.80	133.80	154.60	2.80	2.80	2.80	1.53	1.37	1.30
124.80	141.60	141.60	231.80	141.60	141.60	231.80	3.00	3.00	2.00	1.16	1.08	0.75
120.80	127.40	127.80	157.00	127.40	127.80	157.00	6.00	6.00	6.00	1.55	1.58	1.50
126.35	146.50	144.82	191.98	146.50	144.82	191.98	1.40	1.40	1.40	0.97	1.13	1.13
126.85	134.00	133.60	154.04	134.00	133.60	154.04	4.20	4.20	4.47	1.44	1.42	1.19
129.76	143.86	141.91	194.67	143.86	141.91	194.67	4.20	4.20	4.20	1.35	1.44	1.11
111.21	125.14	125.21	178.14	125.14	125.21	178.14	2.80	2.80	1.00	1.03	1.20	0.50
103.02	127.82	127.42	180.28	127.82	127.42	180.28	4.80	4.80	4.80	1.92	1.89	1.38
115.64	132.89	132.29	203.98	132.89	132.29	203.98	3.40	3.40	3.40	1.53	1.54	1.14

Tabelle 5.14: Satz ${}_1R$ der Gitter-Testprobleme

USTP'					$g \equiv 0$				größter Abstand d'_1				mittlerer Abstand d'_2			
DAA	DNIH(P)	SPH-z(P)	GA	DAH	DNIH(P)	SPH-z(P)	GA	DAH	DNIH(P)	SPH-z(P)	GA	DAH	DNIH(P)	SPH-z(P)	GA	DAH
521.28	580.80	569.24	543.12	583.28	40.80	40.80	42.20	47.00	4.80	5.80	4.80	3.80	2.51	2.67	2.49	2.13
101.04	112.40	111.24	108.84	117.44	41.60	45.20	43.80	51.40	6.40	6.60	6.20	6.20	2.36	2.40	2.39	2.27
70.84	75.80	74.80	73.60	95.40	42.40	42.40	40.20	58.80	5.40	6.00	6.00	5.40	2.43	2.45	2.53	1.77
605.81	665.61	658.91	645.03	657.16	42.80	51.20	49.40	53.57	8.46	6.67	7.47	7.47	3.99	2.70	3.01	3.20
99.89	114.53	108.85	108.85	117.65	45.60	43.00	43.00	53.60	6.80	6.31	6.31	6.31	2.38	2.42	2.42	2.13
62.88	68.80	67.24	65.80	90.08	35.23	33.63	32.23	50.70	9.01	9.01	9.01	9.01	3.32	3.42	3.52	3.12
222.45	259.56	254.45	253.91	243.33	39.82	39.03	36.69	45.42	4.80	4.80	4.80	4.80	1.33	1.33	1.80	1.27
114.79	122.10	120.43	118.07	123.27	46.36	50.44	48.20	47.35	5.59	5.20	5.20	5.59	2.22	1.98	2.03	2.19
63.77	68.33	67.67	67.46	71.70	35.38	37.26	36.75	41.39	8.51	8.51	8.51	8.51	2.83	2.76	2.76	2.64
1956.32	2157.28	2146.04	2051.80	2362.52	132.80	128.60	128.80	152.80	1.40	1.40	1.40	0.00	0.38	0.49	0.64	0.00
319.60	344.44	341.16	337.96	377.00	110.80	112.40	111.20	130.60	3.40	3.40	3.40	3.40	1.26	1.16	1.18	1.03
191.96	207.16	211.04	200.04	236.04	102.40	107.20	97.20	123.20	4.00	4.00	4.00	4.00	1.27	1.25	1.34	1.10
1407.00	1497.03	1498.53	1442.50	1452.40	116.06	122.06	115.24	123.27	1.40	1.00	1.40	1.40	0.42	0.11	0.31	0.16
325.58	366.08	361.27	345.53	391.02	123.31	119.49	112.64	133.25	4.20	4.20	4.20	4.20	1.10	1.14	1.31	1.07
220.22	245.76	246.64	236.64	288.95	113.14	115.74	110.94	142.34	4.40	4.40	4.40	4.40	1.44	1.40	1.47	1.21
1286.02	1448.15	1430.45	1359.23	1612.15	121.70	122.28	123.48	143.63	3.40	3.40	3.40	3.40	1.09	1.09	0.93	0.60
376.43	394.64	399.04	390.64	422.49	123.01	126.41	121.01	138.23	3.57	3.57	3.57	3.57	1.05	1.01	1.08	0.87
225.80	241.08	250.46	235.69	264.94	107.57	115.49	107.68	125.29	4.55	4.00	4.55	4.55	1.33	1.23	1.28	1.15
1892.00	2062.84	2107.48	2012.48	2287.92	179.20	189.00	174.20	212.40	2.40	2.40	2.40	2.40	0.64	0.64	0.87	0.49
528.84	567.32	575.32	558.56	637.80	170.00	171.00	164.20	200.60	2.80	2.80	3.80	3.80	0.71	0.71	0.80	0.59
389.56	411.60	410.00	400.56	433.76	171.20	169.60	163.20	185.40	3.00	3.00	3.00	3.00	0.96	0.99	1.07	0.90
1577.35	1701.97	1717.62	1670.67	1886.46	165.88	174.71	162.51	193.71	2.40	1.00	2.40	3.25	0.60	0.22	0.60	0.36
620.76	652.09	654.95	636.57	687.12	174.10	178.70	169.50	197.45	3.40	3.40	3.40	2.40	0.79	0.76	0.80	0.51
436.27	498.45	495.82	485.24	616.40	191.94	195.03	185.70	254.38	2.80	2.80	2.80	2.80	0.89	0.85	0.95	0.50
2291.75	2471.39	2478.75	2407.40	2757.85	188.71	192.31	181.85	205.14	2.40	2.40	2.40	1.40	0.42	0.42	0.42	0.16
645.90	694.41	694.03	685.30	750.08	184.92	188.68	181.64	210.34	3.40	3.40	3.40	3.74	0.86	0.73	0.88	0.54
428.73	475.10	475.98	461.40	521.26	183.08	186.60	180.30	218.68	3.40	3.40	3.40	3.40	0.89	0.88	0.89	0.69

Tabelle 5.15: Satz ${}_2G$ der Gitter-Testprobleme

	USTP'			$g \equiv 0$			größter Abstand d'_1			mittlerer Abstand d'_2		
DAA	DNIH(P)	SPH-z(P)	DAH	DNIH(P)	SPH-z(P)	DAH	DNIH(P)	SPH-z(P)	DAH	DNIH(P)	SPH-z(P)	DAH
31.20	36.40	35.40	56.60	36.40	35.40	56.60	5.40	5.40	5.40	3.36	3.27	3.49
35.80	38.00	38.00	53.00	38.00	38.00	53.00	6.60	6.40	6.20	2.65	2.58	2.29
33.40	40.60	39.00	58.80	40.60	39.00	58.80	6.40	6.40	5.80	2.70	2.84	2.39
28.37	35.32	35.04	56.93	35.32	35.04	56.93	8.86	9.40	8.02	4.46	4.57	4.29
35.68	40.63	40.11	56.81	40.63	40.11	56.81	7.20	7.20	6.31	2.74	2.66	2.40
27.81	32.98	31.58	37.41	32.98	31.58	37.41	8.61	9.01	9.09	3.50	3.55	3.58
30.48	34.77	34.17	43.25	34.77	34.17	43.25	4.80	4.80	4.80	2.40	2.13	2.22
31.53	39.05	37.85	65.64	39.05	37.85	65.64	6.95	6.95	6.95	2.86	2.96	2.68
31.84	33.47	33.47	41.91	33.47	33.47	41.91	8.31	7.84	8.51	2.95	2.95	2.94
102.20	116.20	113.60	137.40	116.20	113.60	137.40	2.40	2.80	2.00	1.40	1.27	1.13
90.80	101.80	101.40	139.00	101.80	101.40	139.00	3.80	4.00	4.00	1.63	1.77	1.50
85.40	96.40	97.60	114.20	96.40	97.60	114.20	4.00	4.00	4.00	1.51	1.48	1.38
95.70	103.54	103.54	134.14	103.54	103.54	134.14	2.00	2.00	1.40	1.10	1.10	0.88
88.44	106.59	105.93	155.24	106.59	105.93	155.24	3.80	3.80	3.80	1.58	1.56	1.31
95.15	106.09	105.44	133.17	106.09	105.44	133.17	4.40	4.40	4.40	1.59	1.62	1.49
92.91	104.99	101.26	127.11	104.99	101.26	127.11	3.40	3.40	3.40	1.91	1.96	1.64
99.11	111.19	108.19	132.01	111.19	108.19	132.01	3.57	3.57	3.57	1.55	1.52	1.39
89.32	101.20	100.12	127.75	101.20	100.12	127.75	4.40	4.40	4.40	1.51	1.53	1.44
146.00	163.00	160.00	185.60	163.00	160.00	185.60	3.00	3.00	3.00	1.73	1.73	1.56
144.60	156.40	155.00	187.20	156.40	155.00	187.20	3.80	3.80	3.80	1.17	1.14	1.00
149.40	164.80	161.00	186.60	164.80	161.00	186.60	3.00	3.00	3.00	1.22	1.23	1.10
139.84	151.65	150.05	173.65	151.65	150.05	173.65	3.25	3.25	3.25	1.17	1.18	1.17
148.23	162.44	158.20	179.82	162.44	158.20	179.82	2.80	2.80	3.40	1.16	1.21	1.13
151.66	177.81	174.74	236.29	177.81	174.74	236.29	2.80	2.80	2.80	1.18	1.22	1.02
143.05	163.79	161.62	204.70	163.79	161.62	204.70	2.40	2.40	2.40	1.09	1.05	1.16
149.37	168.09	167.57	216.81	168.09	167.57	216.81	3.74	3.74	3.74	1.22	1.24	1.17
158.19	174.45	174.10	208.12	174.45	174.10	208.12	3.40	3.40	3.40	1.12	1.10	1.02

Tabelle 5.16: Satz ${}_2R$ der Gitter-Testprobleme

Epilog

Es bleibt, am Ende dieser Diplomarbeit an all jene Menschen zu denken, die zu ihrem Gelingen beigetragen haben.

Mein Dank gilt zuallererst Herrn Prof. Dr. Günter Leugering, der mir bei der Bearbeitung des Themas jede Freiheit gab und es gleichzeitig verstand, ihm durch interessante Anregungen neue inhaltliche Facetten zu verleihen. Besonders lieben Dank meinen Eltern, die mich während meines Studiums stets unterstützt und mir hilfreich zur Seite gestanden haben.

Nicht zu vergessen sind das Bayerische Staatsministerium für Unterricht, Kultus, Wissenschaft und Kunst und die Universität Bayreuth, die mir ein Stipendium im Rahmen des Bayerischen Begabtenförderungsgesetzes zukommen ließen.

Literaturverzeichnis

[AHU85] A. V. Aho, J. E. Hopcroft, J. D. Ullman: *Data Structures and Algorithms,* Addison-Wesley, Reading (1985);

[Ane80] Y. P. Aneja: *An Integer Linear Programming Approach to the Steiner Problem in Graphs,* Networks 10 (1980), 167–178;

[Bal95] V. K. Balakrishnan: *Network Optimization,* Chapman & Hall, London (1995);

[BaP87] A. Balakrishnan, N. R. Patel: *Problem Reduction Methods and a Tree Generation Algorithm for the Steiner Network Problem,* Networks 17 (1987), 65–85;

[Bea84] J. E. Beasley: *An Algorithm for the Steiner Problem in Graphs,* Networks 14 (1984), 147–159;

[Bea89] J. E. Beasley: *An SST-Based Algorithm for the Steiner Problem in Graphs,* Networks 19 (1989), 1–16;

[Ber91] D. P. Bertsekas: *Linear Network Optimization: Algorithms and Codes,* MIT Press, Cambridge (1991);

[ChB97] P. C. Chu, J. E. Beasley: *Constraint Handling in Genetic Algorithms: The Set Partitioning Problem,* The Management School, Imperial College, London (1997);

[DrW72] S. E. Dreyfus, R. A. Wagner: *The Steiner Problem in Graphs,* Networks 1 (1972), 195–207;

[DuV87] C. W. Duin, A. Volgenant: *Some Generalizations of the Steiner Problem in Graphs,* Networks 17 (1987), 353–364;

[DuV89] C. W. Duin, A. Volgenant: *Reduction Tests for the Steiner Problem in Graphs,* Networks 19 (1989), 549–567;

[Esb95] H. Esbensen: *Computing Near-Optimal Solutions to the Steiner Problem in a Graph Using a Genetic Algorithm,* Networks 26 (1995), 173–185;

[Fis81] M. L. Fisher: *The Lagrangian Relaxation Method for Solving Integer Programming Problems,* Management Science 27 (1981), 1–17;

[FrT87] M. L. Fredman, R. E. Tarjan: *Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms,* Journal of the Association for Computing Machinery 34/3 (1987), 596–615;

[GaJ77] M. R. Garey, D. S. Johnson: *The Rectilinear Steiner Tree Problem is NP-complete,* SIAM Journal on Applied Mathematics 32 (1977), 826–834;

[GaJ79] M. R. Garey, D. S. Johnson: *Computers and Intractability: A Guide to the Theory of NP-Completeness,* Freeman, San Francisco (1979);

[Geo71] A. M. Geoffrion: *Duality in Nonlinear Programming: A Simplified Applications-Oriented Development,* SIAM Review 13 (1971), 1–37;

[Geo74] A. M. Geoffrion: *Langrange Relaxation for Integer Programming,* Mathematical Programming Study 2 (1974), 82–114;

[GiP68] E. N. Gilbert, H. U. Pollak: *Steiner Minimal Trees,* SIAM Journal on Applied Mathematics 16,1 (1968);

[Gol89] D. E. Goldberg: *Genetic Algorithms in Search, Optimization and Machine Learning,* Addison-Wesley, Reading (1989);

[GoD91] D. E. Goldberg, K. Deb: *A Comparative Analysis of Selection Schemes Used in Genetic Algorithms,* Foundations of Genetic Algorithms (1991, hrsg. G. Rawlins);

[GoM93] M. X. Goemans, Y. Myung: *A Catalog of Steiner Tree Formulations,* Networks 22 (1993), 19–28;

[Hak71] S. L. Hakimi: *Steiner's Problem in Graphs and Its Implications,* Networks 1 (1971), 113–133;

[Hol92] J. H. Holland: *Adaptation in Natural and Artificial Systems,* MIT Press, Cambridge (1992);

[Kar72] R. M. Karp: *Reducibility among Combinatorial Problems,* Complexity of Computer Computations, Plenum Press, New York (1972), 85–103;

[KMB81] L. Kou, G. Markowsky, L. Berman: *A Fast Algorithm for Steiner Trees,* Acta Informatica 15 (1981), 141–145;

[Knu81] D. E. Knuth: *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms,* Addison-Wesley, Reading (1981^2);

[KoM96] T. Koch, A. Martin: *Solving Steiner Tree Problems in Graphs to Optimality,* Preprint 1996;

[KRS93] A. Kapsalis, V. J. Rayward-Smith, G. D. Smith: *Solving the Graphical Steiner Tree Problem Using Genetic Algorithms,* Journal of the Operations Research Society 44/4 (1993), 397–406;

[Kru56] J. B. Kruskal, Jr.: *On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem,* Proceedings of the American Mathematical Society, 7 (1956), 48–50;

[LeT86] J. van Leeuwen, R. E. Tarjan: *Worst-Case Analysis of Set-Union Algorithms,* Journal of the Association for Computing Machinery, 1986;

[Meh88] K. Mehlhorn: *A Faster Approximation Algorithm for the Steiner Problem in Graphs,* Information Processing Letters 27 (1988), 125–128;

[Pri57] R. C. Prim: *Shortest Connection Networks and Some Generalizations,* Bell System Technical Journals 36 (1957), 1389–1401;

[Ran94] S. Ranka, H. S. Maini, K. G. Mehrota, C. K. Mohan: *Genetic Algorithms for Graph Partitioning and Incremental Graph Partitioning,* Center for Science and Technology, Syracuse University, (1994);

[Ray83] V. J. Rayward-Smith: *The Computation of Nearly Minimal Steiner Trees in Graphs,* International Journal of Mathematical Education in Science and Technology 14 (1983), 15–23;

[RaC86] V. J. Rayward-Smith, A. Clare: *On Finding Steiner Vertices,* Networks 16 (1986), 283–294;

[Sed92] R. Sedgewick: *Algorithms in C++,* Addison-Wesley, Reading (1992);

[SFG82] M. L. Shore, L. R. Foulds, P. B. Gibbons: *An Algorithm for the Steiner Problem in Graphs,* Networks 12 (1982), 323–333;

[Seg87] A. Segev: *The Node-Weighted Steiner Tree Problem,* Networks 17 (1987), 1–17;

[Sys89] G. Syswerda: *Uniform Crossover in Genetic Algorithms,* Proceedings of the Third International Conference on Genetic Algorithms (1989), 2–9;

[Str92] B. Stroustrup: *The C++ Programming Language,* Addison-Wesley, Reading (1992^2);

[Tar72] R. Tarjan: *Depth-First Search ans Linear Graph Algorithms,* SIAM Journal on Computation 1 (1972), 146–160;

[Tar77] R. E. Tarjan: *Finding Optimum Branchings,* Networks 7 (1977), 25–35;

[ThS92] K. Thulasiraman, M. N. Swamy: *Graphs: Theory and Algorithms,* John Wiley & Sons, New York (1992);

[TaM80] H. Takahashi, A. Matsuyama: *An Approximate Solution for the Steiner Problem in Graphs,* Mathematica Japonica 24 (1980), 573–577;

[Whi89] D. Whitley: *The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best,* Proceedings of the Third International Conference on Genetic Algorithms (1989), 116–121;

[Wid86] P. Widmayer: *On Approximation Algorithms for Steiner's Problem in Graphs,* Lecture Notes in Computer Science 246 (1986), 17–28;

[Win87] P. Winter: *Steiner Problem in Networks: A Survey,* Networks 17 (1987), 129–167;

[WiM92] P. Winter, J. MacGregor-Smith: *Path-Distance Heuristics for the Steiner Problem in Undirected Networks,* Algorithmica 7 (1992), 309–327;

[Won82] R. T. Wong: *A Dual Ascent Approach for Steiner Tree Problems on a Directed Graph,* Mathematical Programming 28 (1984), 271–287;

[WWW86] Y. F. Wu, P. Widmayer, C. K. Wong: *A Faster Approximation Algorithm for the Steiner Problem in Graphs,* Acta Informatica 23 (1986), 223–229;

Erklärung

Hiermit erkläre ich, daß ich die vorliegende Diplomarbeit selbst verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Bayreuth, den 1. Juli 1997 ....

(Clemens Frey)