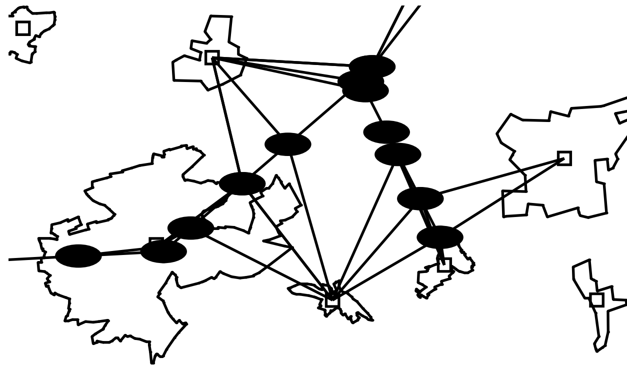
Diplomarbeit
Standortplanung von Bahnhöfen



Steffen Mecke[1]

März 2003

[1]Steffen.Mecke@uni-konstanz.de

Inhaltsverzeichnis

Einleitung **5**

1 Alpha **7**

1.1 Warum man neue Bahnhöfe baut 7
1.2 Modellierung 7
1.2.1 Grundlagen 7
1.2.2 Optimierungsprobleme 10
1.3 Komplexität 12
1.4 Was bisher geschah 12
1.5 Überblick 13

2 Diskretisierung **15**

2.1 Eine endliche dominierende Menge für CSLP 16
2.2 CSLP als Set Covering 17
2.3 Konsequenzen für die Lösung der gestellten Probleme 18
2.4 Zwischenbemerkung zur Eingabegröße 19

3 Lösungsansätze für CSLP **21**

3.1 Definitionen 21
3.2 Zerlegung 22
3.3 Kompression 25
3.3.1 Definition 25
3.3.2 Kompressionsalgorithmen 29
3.3.3 Kompression II 31
3.4 Exakte Lösungsmethoden 32
3.4.1 Bekannte Verfahren 32
3.4.2 Instanzen mit Intervall-Eigenschaft 32
3.4.3 Real-World-Instanzen 38

3.4.4 Die Kombinationsmethode . . . 40
3.4.5 Heuristiken bei der Auswahl der Kombinationskandidaten . . . 44
3.5 Näherungsverfahren . . . 45
3.5.1 Heuristische Zerlegungsmethoden . . . 45

4 Implementation und Ergebnisse **49**
4.1 Hilfsmittel . . . 49
4.2 Daten und Parameter . . . 49
4.2.1 Reale Daten . . . 49
4.2.2 Parameter . . . 50
4.2.3 Sonstige Daten . . . 51
4.3 Vergleich der Instanzen . . . 52
4.4 Kompression . . . 55
4.5 Komponenten und Sortierung . . . 57
4.6 Performance der Kombinationsalgorithmen . . . 66

Omega **69**

Einleitung

Rund um die Verkehrsplanung und insbesondere beim Zugverkehr gibt es eine Fülle von lohnenden Fragestellungen. Sie reichen etwa von der Linienplanung über die Fahrplanauskunft bis zum Verspätungsmanagement. Am Anfang steht die Planung des Streckennetzes. Dabei wird man sich naturgemäß selten in der Situation befinden, das komplette Netz neu aufzubauen, sondern es wird in der Regel eine bestehende Infrastruktur zu erweitern sein.

Gegenstand dieser Arbeit wird die Standortplanung neuer Bahnhöfe sein. Neue Bahnhöfe zu bauen hat z.T. gegensätzliche Effekte zur Folge, die den Gewinn beeinflussen, den die Betreiber des Bahnnetzes erwirtschaften können.

Als Erstes wenden wir uns deshalb verschiedenen Möglichkeiten zu, diese Effekte zu modellieren. Praktisch alle interessanten Modellierungen resultieren in **NP**-schweren Problemen. Wir werden eine dieser Möglichkeiten genauer betrachten und diese zunächst vereinfachen, indem wir die Zahl der möglichen Positionen für neue Halte einschränken. Trotzdem bleibt das zugrundeliegende Problem weiterhin **NP**-schwer. Danach entwickeln wir einige einfache Regeln, um die Größe der Probleme zu verringern. Wir werden beobachten, dass diese so gut funktionieren, dass anschließend für die meisten praktischen Probleme eine Lösung mit sehr geringem Aufwand gefunden werden kann. Zum Schluss des theoretischen Teils geben wir einige exakte und näherungsweise Lösungsverfahren an.

Im praktischen Teil werden wir die Leistungsfähigkeit der vorgestellten Methoden an einigen Datensätzen testen, darunter „echte“ Bahndaten und einige Beispiele aus der Literatur. Wir werden sehen, dass unsere Methoden geeignet sind, die Instanzen, die auf realen Daten beruhen, vollständig und exakt zu lösen, obwohl das zugrundeliegende Problem **NP**-schwer ist.

Ich möchte an dieser Stelle der Betreuerin meiner Diplomarbeit, Prof. Dr. D. Wagner für die fortwährende Unterstützung bei meiner Arbeit herzlich danken. Ebenfalls möchte ich den Mitarbeitern ihres Lehrstuhls an der Universität Konstanz danken, sowie meinen Kommilitonen, die mir bei auftretenden Problemen jederzeit hilfreich zur Seite standen. Nicht zuletzt geht mein Dank an meine meine Familie, für ihr Interesse und ihre Geduld und ihre Hilfe während meines Studiums.

Steffen Mecke
Konstanz, den 24. Juli 2003

Kapitel 1

Alpha

1.1 Warum man neue Bahnhöfe baut

Ein Eisenbahnnetz ist nicht statisch. Häufig werden Bahnhöfe (oder „Halte“) stillgelegt und (heutzutage weniger häufig) neue Bahnhöfe eröffnet. Welches ist die optimale Anzahl an Bahnhöfen und wo sollten sie sich befinden? Um diese Frage beantworten zu können müssen wir uns zunächst überlegen welche Auswirkungen das Einrichten neuer Halte hat.

Zum einen erhöht sich die Attraktivität der Bahn, denn potentielle Kunden werden sich eher für die Bahn entscheiden, wenn sie einen kürzeren und schnelleren Weg zum nächsten Bahnhof haben. Auf der anderen Seite verringert sich die Attraktivität, da sich die Reisezeit (anderer Kunden) durch die zusätzlichen Halte verlängert. Außerdem verursachen Halte Kosten für die Bahngesellschaft. Diese Effekte beeinflussen die Attraktivität der Bahn und letztlich den erzielbaren Gewinn.

1.2 Modellierung

Wir wollen nun einige Optimierungsprobleme definieren. Als Grundlage dienen [SHLW02] und [KPS$^+$02].

1.2.1 Grundlagen

Sei $P \subset \mathbb{R}^2$ die (endliche) Menge der ***Siedlungsflächen***, $G = (V, E)$ ein (endlicher) Graph[1] mit Knotenmenge V und Kantenmenge E und geradliniger Einbettung $\varphi\colon V \to \mathbb{R}^2$ in die Ebene. Dadurch wird das Bahnnetz repräsentiert: Die Knoten sind vorhandene Haltestellen oder “optische Punkte”, d.h. Knicke oder Weichen, die Kanten heißen auch *Strecken*. Zur Vereinfachung unterscheiden wir häufig nicht zwischen dem Graphen und seiner Einbettung und bezeichen auch letztere mit G.

[1] Wir gehen davon aus, dass der Leser über zumindest die rudimentäre Kenntnisse über Graphentheorie und Algorithmen verfügt, wie sie etwa in [CLR97] oder beinahe jeder anderen Einführung in Algorithmen vermittelt werden.

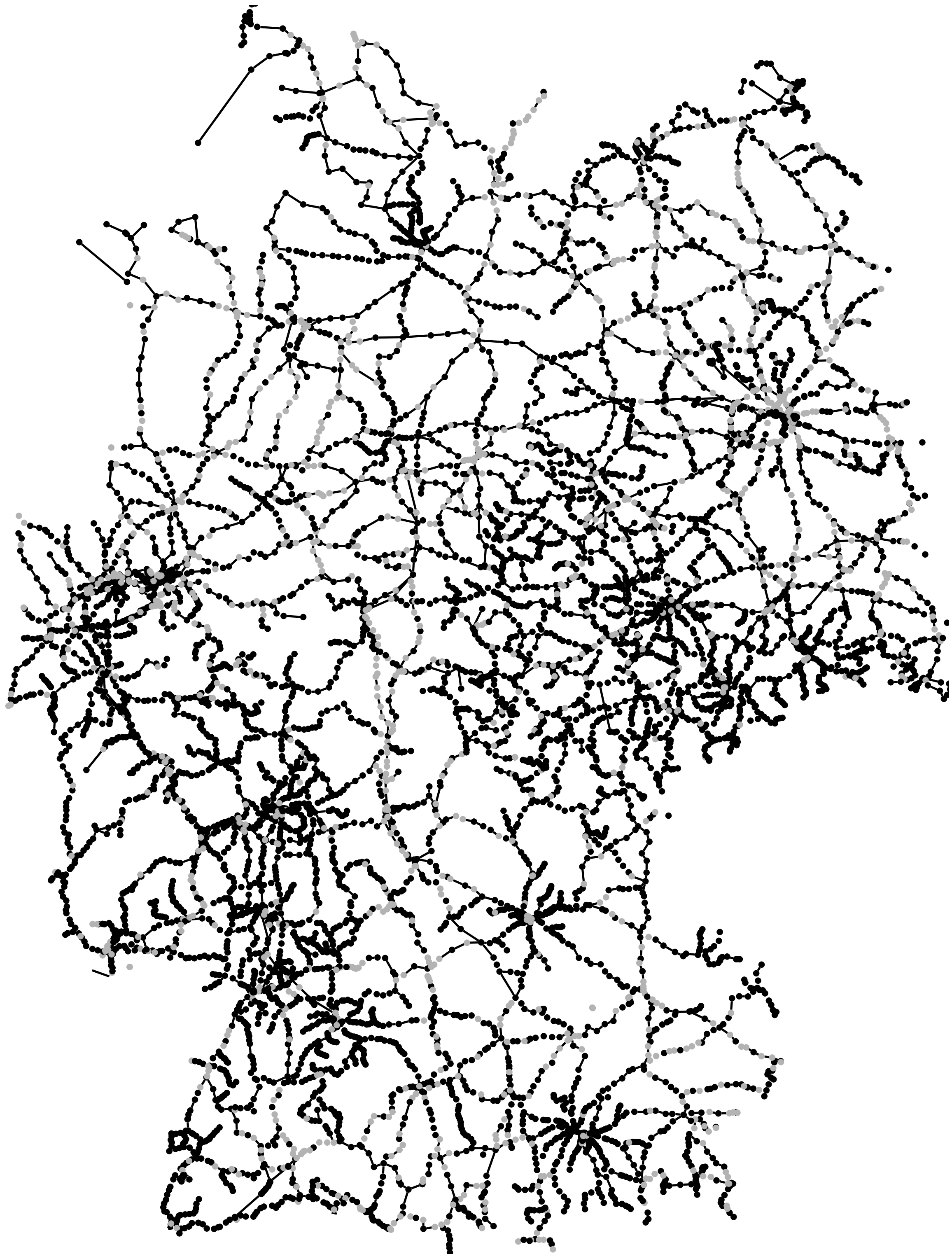


Abbildung 1.1: Bahnnetz, Deutschland

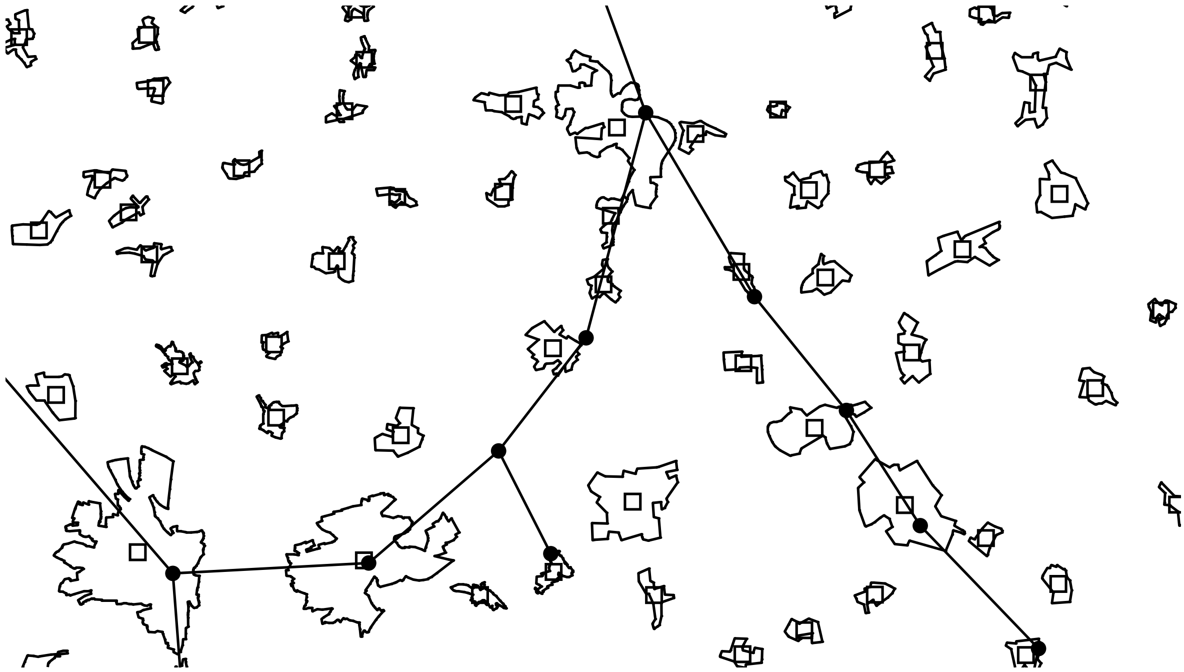


Abbildung 1.2: Bahnnetz, Ausschnitt, mit Siedlungsflächenpolygonen und -schwerpunkten

Abbildung 1.1 zeigt den Bahngraphen der Deutschen Bahn, in Abbildung 1.2 sind das Bahnnetz sowie die Umrisse der Siedlungsflächen eines Ausschnitts zu sehen. Die Schwerpunkte dieser Umrisse dienen uns als Siedlungsflächenmenge P.

Jedem Knoten v bzw. jeder Strecke e sei ein Gewicht c_v bzw. c_e zugeordnet. (Dies könnte zum Beispiel die Anzahl der Fahrgäste sein, die einen bestimmten Punkt im Netz pro Tag überqueren.)

Für einen Punkt $s \in G$ (genauer: $s \in \varphi(G)$) sei

$$g(s) := \begin{cases} s, \text{ falls } s \in V \\ e, \text{ falls } s \in e, s \notin V \end{cases} .$$

die Umkehrung der Einbettung φ.

Die ***Kosten*** einer (endlichen) Teilmenge $S \subset G$ seien durch

$$c(S) := \sum_{s \in S} c_{g(s)}$$

gegeben.

Den ***Abstand*** zwischen zwei Punkten $x, y \in \mathbb{R}^2$ bezeichnen wir mit $\operatorname{dist}(x, y)$, den Abstand zwischen zwei (endlichen) Mengen mit $\operatorname{dist}(X, Y) := \inf\{\operatorname{dist}(x, y) : x \in X, \ y \in Y\}$. Obwohl große Teile unserer Ergebnisse auch bei allgemeineren Abstandsmaßen ihre Gültigkeit behalten (vgl. [SHLW02]), beschränken wir uns hier auf die euklidische Metrik.

Sei im Folgenden $r > 0$ ein fest vorgegebener Radius. Ein Punkt $p \in P$ heiße ***erschlossen*** durch einen Punkt $s \in G$, falls $\operatorname{dist}(p, s) \leq r$. Für eine Menge $S \subset G$ sei

$$\operatorname{cover}(S) := \{p \in P : \operatorname{dist}(p, S) \leq r\}.$$

Damit können wir die ***Einnahmen*** für eine Menge $S \subset G$ definieren. Dazu sei jedem $p \in P$ ein Gewicht w_p (zum Beispiel die Bevölkerungszahl) zugeordnet. Somit definieren wir für ein $P' \subset P$ zunächst

$$\text{gain}(P') := \sum_{p \in P'} w_p$$

und damit für eine Teilmenge $S \subset G$

$$\text{gain}(S) := \text{gain}(\text{cover}(S))$$

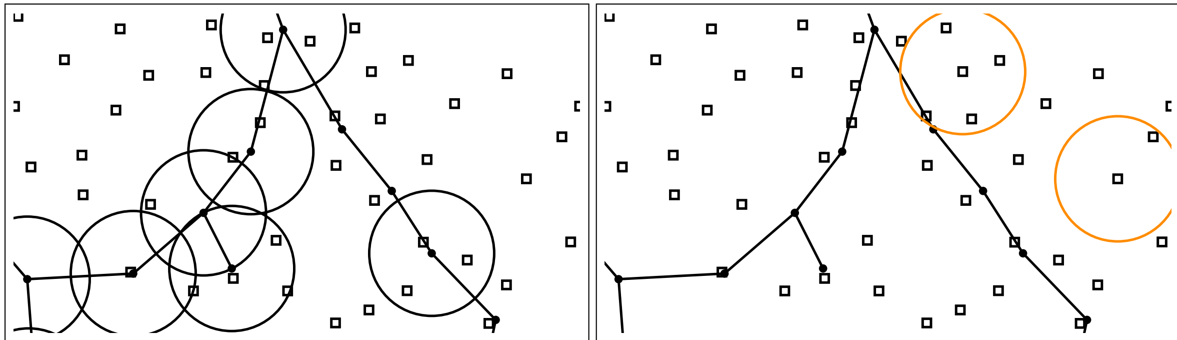


Abbildung 1.3: Überdeckung, Radius 3km

Beispiel 1.1: Der linke Teil der Abbildung 1.3 zeigt den schon bekannten Ausschnitt mit Bahngraph, Siedlungsflächen und den Kreisen mit Radius 3km um die bestehenden Bahnhöfe. Siedlungsflächen, die innerhalb dieser Kreise liegen, sind erschlossen. Der rechte Teil der Abbildung zeigt zwei Siedlungsflächen, von denen eine erschließbar ist (der Kreis mit Radius 3km um sie schneidet das Bahnnetz) und eine nicht. Abbildung 1.4 zeigt eine minimale Menge von Halten (Ellipsen), die alle erschließbaren Siedlungsflächen abdeckt. Eine solche zu finden ist das Hauptziel dieser Arbeit.

1.2.2 Optimierungsprobleme

Mit diesen Definitionen ist es uns möglich, die folgenden Optimierungsprobleme zu definieren.

CSLP oder MinStation

- Zulässige Lösung: $S \subset G$ mit $\text{cover}(S) = P$
- Minimiere $\text{cost}(S)$

Eine Variante hiervon ist CSLP′:

- Gegeben: Eine Menge $S' \subset G$ von existierenden Halten
- Zulässige Lösung: $S \subset G$ mit $\text{cover}(S \cup S') = P$
- Minimiere $\text{cost}(S)$

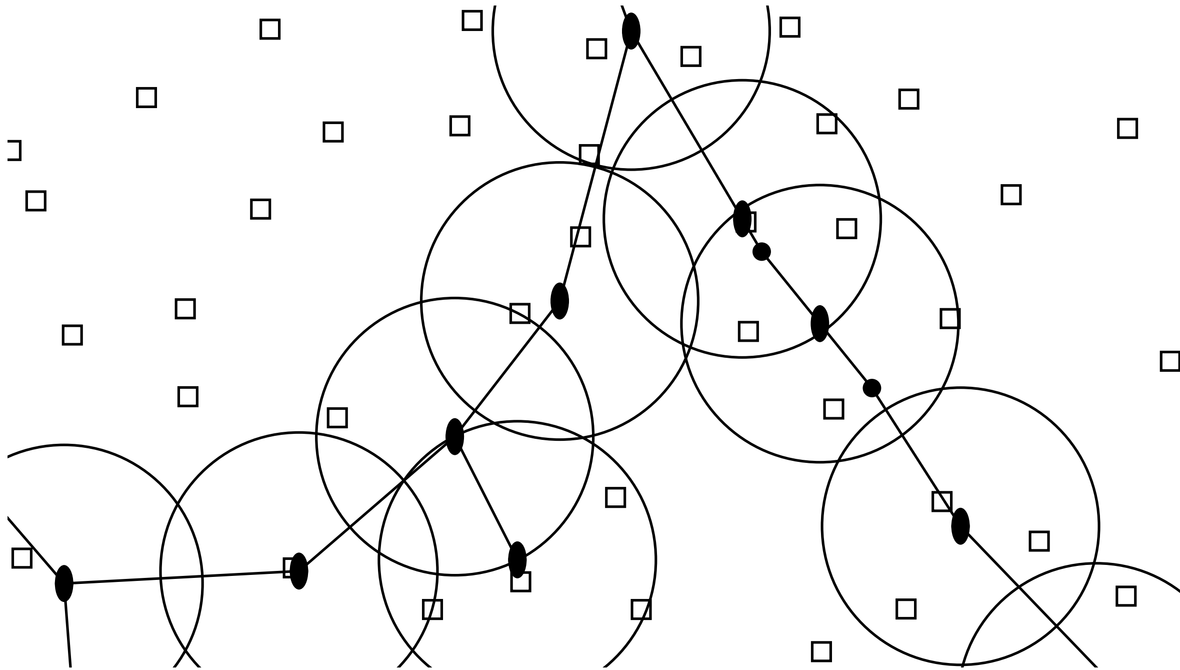


Abbildung 1.4: Optimale Lösung

Es ist jedoch leicht einzusehen, dass die beiden Probleme äquivalent sind: Um von CSLP′ zu CSLP zu kommen, braucht man nur $P = \emptyset$ zu setzen, für die umgekehrte Richtung ersetze man P durch $P \setminus \text{cover}(S')$. Trotzdem sind die beiden Probleme auf praktischen Daten sehr unterschiedlich schwer, wie wir im Ergebnisteil zeigen werden.

p-Cover

- Gegeben: Prozentsatz $0 \leq p \leq 1$
- Zulässige Lösung: $S \subset G$ mit ***Erschließungsgrad*** p, d.h. $|\,\text{cover}(S)| \geq p \cdot |P|$
- Minimiere $\text{cost}(S)$

Offensichtlich ist CSLP ein Spezialfall von p-Cover, denn es gilt 1-Cover = CSLP.

MaxGain

- Gegeben: Budget $B \in \mathbb{R}^+$
- Zulässige Lösung: $S \subset G$ mit $\text{cost}(S) \leq B$
- Maximiere $\text{gain}(S)$

Facility Location

- Zulässige Lösung: Jedes $S \subset G$
- Maximiere $\text{gain}(S) - \text{cost}(S)$

Mehrere Radien

Die 0-1-Entscheidung, ob eine Siedlungsfläche erschlossen ist oder nicht, scheint eine grobe Vereinfachung zu sein. Als mögliche Abhilfe könnte man etwa eine Siedlungsfläche als „zur Hälfte erschlossen“ bezeichnen, wenn sie weniger als 5km und als „ganz erschlossen“, wenn sie weniger als 2km vom nächsten Bahnhof entfernt ist.

Der Unterschied zu CSLP MaxGain oder Facility Location besteht also darin, dass es mehrere Radien $0 = r_0 < r_1 < \cdots < r_k$ sowie Prozentsätze $0 < \alpha_1 < \cdots < \alpha_k \leq 1$ gibt und

$$\begin{aligned} \text{cover}_{r_1 r_2}(S) &:= \{p \in P : \text{dist}(p, S) > r_1 \text{ und } \text{dist}(p, S) <= r_2\} \\ \text{gain}(S) &:= \sum_{i=1,\dots k} \text{gain}(\text{cover}_{r_{i-1} r_i}(S)) \cdot \alpha_i \end{aligned}$$

Stetige Maße

Als weitere Verallgemeinerung könnte man eine beliebige (evtl. stetige, monoton fallende) Funktion $\alpha(\cdot)$ definieren, und

$$\text{gain}(S) := \sum_{p \in P} w_p \cdot \alpha(\text{dist}(p, S)).$$

Ein ähnlicher Ansatz wurde bereits in [HLS+01] verfolgt (dort: Saved Travel Time).

1.3 Komplexität

Die meisten betrachteten Probleme sind **NP**-schwer. Für CSLP wurde dies in [HLS+01] und [SHLW02] gezeigt. p-Cover ist eine Verallgemeinerung von CSLP und mithin ebenfalls **NP**-schwer. MaxGain ist mit k-Median verwandt, das im Allgemeinen als **NP**-schweres Problem bekannt ist. Die **NP**-Vollständigkeit von Saved Travel Time wurde ebenfalls in [HLS+01] gezeigt.

1.4 Was bisher geschah

k-Median, k-Center und Facility Location Alle betrachteten Probleme sind Spezialfälle bereits bekannter Probleme, für die bereits eine Reihe von Lösungsansätzen existiert.

Neuere Arbeiten kommen von Shmoys, Charikar, Guha, Tardos et al. ([Shm95, STA97, GK99, CGTS99]), Jain und Vazirani ([JV99]) sowie Thorup ([Tho01]). Alle diese Arbeiten befassen sich jedoch mit diskreten Standortproblemen, die hier a priori nicht vorliegen.

MaxGain und CSLP Für diese Probleme, die sich spezieller mit dem Kontext der Haltestellenplanung für Bahnlinien befassen, wurden erst wenige vollständige Methoden gezeigt. CSLP und Saved Travel Time wurden zuerst in [HLS+01] eingeführt. [SHLW02] und [KPS+02] befassen sich vor allem mit der Lösung von Spezialfällen von CSLP bzw. MaxGain. In [SHLW02] ist auch ein Überblick über weitere verwandte Ansätze zu finden.

Auch [Ruf02] befasst sich mit CSLP. Es ist der erste uns bekannte vollständige Ansatz zur Lösung von CSLP.

1.5 Überblick

In den folgenden Kapiteln werden wir uns fast ausschließlich mit der Lösung von CSLP befassen. Im nächsten Kapitel werden wir sehen, wie das kontinuierliche Problem auf ein diskretes reduziert werden kann, also eines, bei dem nur eine endliche Menge von Punkten für neue Halte betrachtet werden muss. Damit werden wir es auf das Set Covering-Problem zurückführen.

Kapitel 3 gliedert sich in drei Hauptteile: Als erstes werden wir sehen, wie wir die Größe von CSLP-Instanzen durch Anwendung einiger einfacher Regeln verringern können. Anschließend befassen wir uns mit der Lösungverfahren zunächst für spezielle und dann für beliebige Instanzen. Den Abschluss bilden einige nicht exakte Verfahren, mit denen zu große Instanzen in handlichere Stücke zerlegt werden können.

Das letzte Kapitel ist der Anwendung der zuvor eingeführten Verfahren gewidmet. Wir werden die zur Verfügung stehenden Test-Instanzen analysieren und die Performance unserer Algorithmen in der Praxis testen.

Kapitel 2

Diskretisierung

Von Station Location zum Set Covering Problem

Wir werden uns im Folgenden auf die Lösung von CSLP beschränken. Abbildung 1.4 zeigt sinnvolle Kandidaten (Ellipsen) für neue Halte. In diesem Kapitel werden wir sehen, dass es genügt, eine endliche Menge von „Kandidaten“ als mögliche Standorte zu betrachten. Insbesondere folgt daraus, dass es stets eine endliche optimale Lösung für CSLP gibt. Damit wird es uns auch möglich sein, die hier gestellten Probleme auf bekannte, wie zum Beispiel Set Covering zurückzuführen. Als Basis diente [SHLW02].

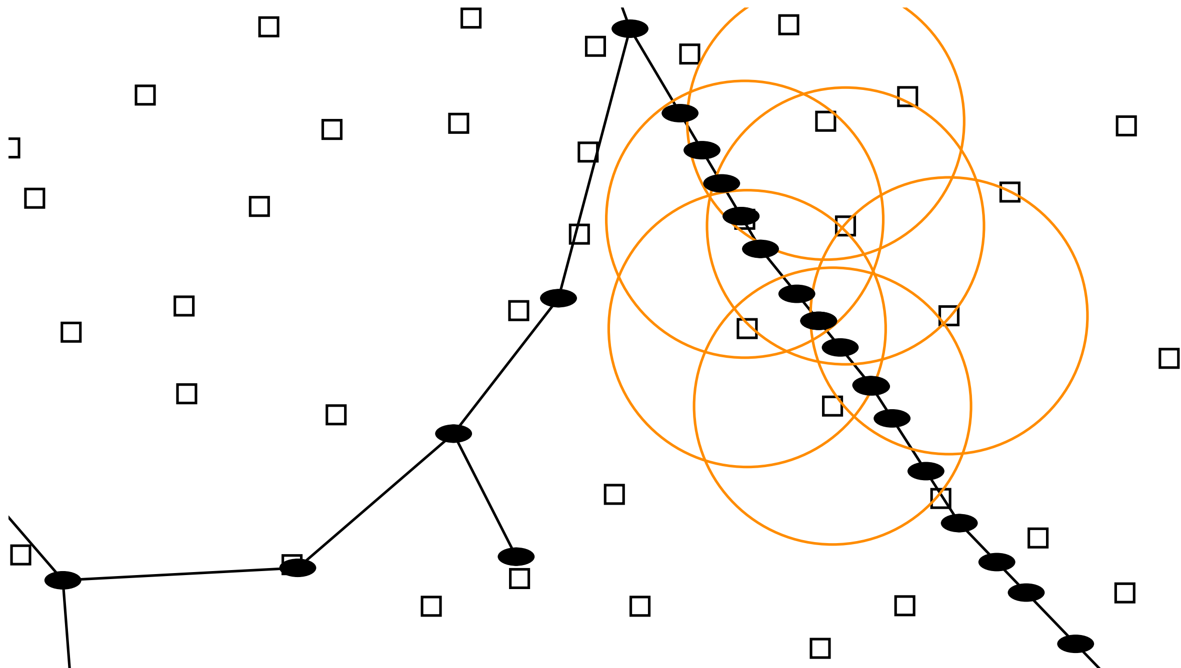


Abbildung 2.1: Sinnvolle Kandidaten für neue Halte

2.1 Eine endliche dominierende Menge für CSLP

Ziel dieses Abschnitts ist es, eine ***endliche dominierende Menge*** zu finden, also eine endliche Menge $C \subset G$ von *Kandidaten*, die eine optimale Lösung für CSLP enthält. Wir wollen uns im Rest des Kapitels auf solche Instanzen beschränken, die überhaupt eine Lösung besitzen, das heißt, für die $P \subset \text{cover}(G)$ gilt.

Sei

$$F_1 := \{s \in G\colon\ \text{dist}(p, s) = r,\ \ p \in P\}$$

die (endliche) Menge der Schnittpunkte der Kreise mit Radius r um die Siedlungsflächen mit den Kanten des Graphen.

Satz 2.1:

$$F := V \cup F_1 \cup \{(v_1 + v_2)/2 : \{v_1, v_2\} \in E\}$$

ist eine endliche dominierende Menge.

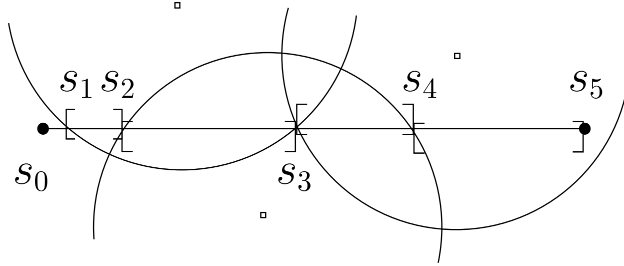


Abbildung 2.2: Intervalle

Um dies beweisen zu können, wählen wir für jede Kante $e = (v, w)$ eine Orientierung $v < w$ und diese induziert eine Ordnung für alle Punkte der Kante durch

$$s_1 <_e s_2 : \Longleftrightarrow\ \text{dist}(s_1, v) < \text{dist}(s_2, v).$$

Damit definieren wir das Intervall

$$]s_1, s_2[_e := \{s \in e |\ s_1 <_e s <_e s_2\} \subset e.$$

Wir betrachten dadurch auch die Punkte von F auf e als geordnet und bezeichnen sie mit $v = s_0 <_e s_1 <_e \cdots <_e s_{N_e} = w$. Mit Hilfe dieser Notation formulieren wir

Lemma 2.2: Sei $e \in E$, $s \in]s_j, s_{j+1}[_e$und $t \in [s_j, s_{j+1}]_e$ für ein $j \in \{0, \dots, N_e - 1\}$. Dann gilt

$$\text{cover}(s) \subset \text{cover}(t).$$

Beweis. Annahme: Sei $\text{cover}(s) \not\subseteq \text{cover}(t)$, d.h. es gibt ein $p \in P$ mit $\text{dist}(s, p) \leq r$ und $\text{dist}(t, p) > r$. Aber $\text{dist}(s, p) = r$ ist nach Wahl von s nicht möglich. Wegen der Stetigkeit von dist gibt es also (aufgrund des Zwischenwertsatzes) einen Punkt $z \in]s_j, s_{j+1}[_e$ mit $\text{dist}(z, p) = r$. Dies impliziert $z \in F_1$ im Widerspruch zur Definition von s_j und s_{j+1}. □

Beweis von Satz 2.1. Sei S eine optimale Lösung. Wir werden zeigen, dass jedes $s \in S \setminus F$ durch ein $t \in F$ ersetzt werden kann, ohne die Optimalität aufzugeben oder die Eigenschaft, eine Lösung zu sein. Ein solches s kann nicht aus V sein, muss also auf einer Kante $e = (v, w)$ liegen, genauer in einem Intervall $]s_i, s_{i+1}[_e$. Falls die Punkte s_i, s_{i+1} die Endpunkte von e sind, so kann s durch $t := (v + w)/2$ ersetzt werden. Falls (o.B.d.A) $s_i \in]v, w[$ gilt, ersetzen wir s durch $t := s_i$. In beiden Fällen gilt

$$\text{cover}(S \setminus \{s\} \cup \{t\}) \supset \text{cover}(S) \text{ und}$$

$$\text{cost}(S \setminus \{s\} \cup \{t\}) \leq \text{cost}(S),$$

was den Beweis abschließt. □

2.2 CSLP als Set Covering

Mit diesem Ergebnis können wir CSLP als das lange bekannte SET COVERING-Problem neu formulieren. Dazu sei $\gamma_f := \text{cover}(f)$ und $\text{cost}(\gamma_f) := \text{cost}(f)$ sowie

$$\Gamma := \{\gamma_f : f \in F\} \subset \mathbb{P}(P).$$

Gesucht ist $\Gamma_0 \subset \Gamma$ mit

$$\bigcup_{\gamma \in \Gamma_0} \gamma = P \tag{2.1}$$

und

$$\text{cost}(\Gamma_0) := \sum_{\gamma_f \in \Gamma_0} \text{cost}(\gamma_f) \tag{2.2}$$

minimal unter allen $\Gamma' \subset \Gamma$ mit (2.1).

Anders formuliert: Gegeben sei eine Matrix $A_{cov} := (a_{pf})$ mit

$$a_{pf} = \begin{cases} 1, & \text{falls } p \in \text{cover}(f) \\ 0 & \text{sonst,} \end{cases}$$

deren Zeilen mit den Siedlungsflächen und deren Spalten mit den Kandidaten korrespondieren und $c := (c_f)_{f \in F}$. Dann entspricht eine Lösung $S \subset F$ einem Vektor $x = (x_f)_{f \in F}$, wo

$$x_f = \begin{cases} 1, & \text{falls } f \in S \\ 0 & \text{sonst.} \end{cases}$$

Damit lässt sich das CSLP-Problem als lineares Programm formulieren:

$$\begin{array}{lll} \text{Minimiere} & c^T x, & \\ \text{so dass} & A_{cov} x & \geq \mathbf{1} \\ & x & \in \{0,1\}^{|F|}. \end{array} \tag{2.3}$$

Mit $\mathbf{1}$ ist dabei der Vektor $(1)_{p \in P}$ gemeint.

Beides sind Formulierungen des SET COVERING-Problems.

Beispiel 2.1: Bei der in Abbildung 2.3 skizzierten Instanz mit drei Siedlungsflächen, drei Knoten und zwei Strecken benötigen wir acht Kandidaten. Das SET COVERING-Problem ist

$$\begin{array}{lll} \text{Minimiere} & x_0 + x_1 + x_2 + x_3 + x_4 x_5 + x_6 + x_7, & \\ \text{so dass} & \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} x & \geq \mathbf{1} \end{array}$$

und die optimale Lösung ist $S = \{f_2\}$ bzw. $x = (0,0,1,0,0,0,0)^T$, vorausgesetzt, alle Knoten und Strecken haben gleiche Kosten.

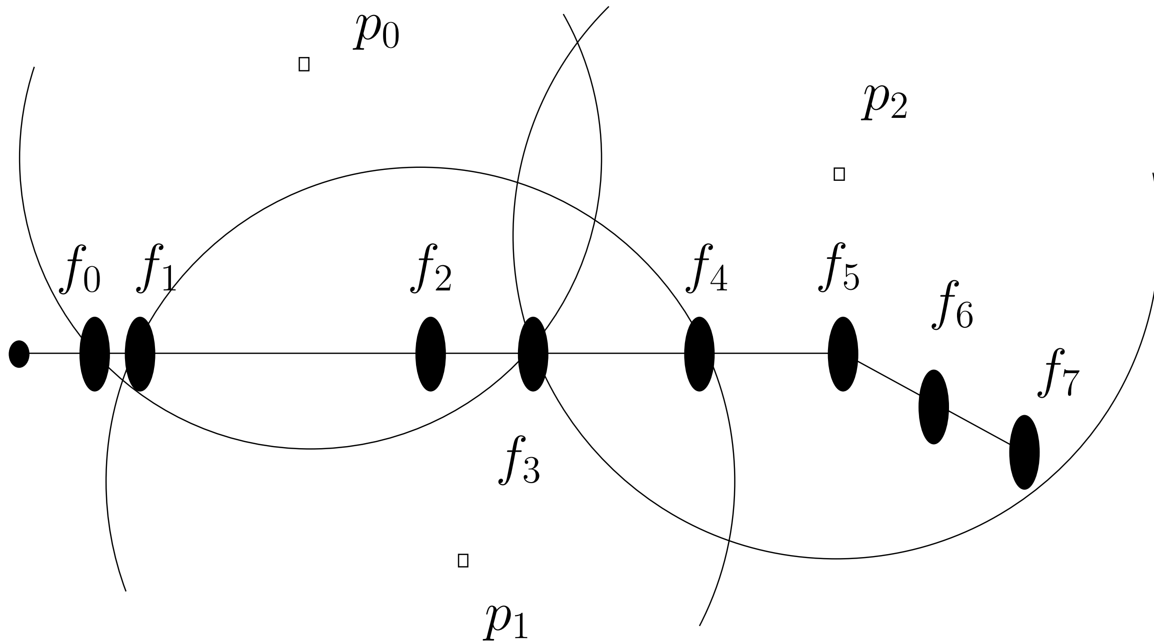


Abbildung 2.3: Beispiel einer Kandidatenmenge

2.3 Konsequenzen für die Lösung der gestellten Probleme

Die im vorigen Abschnitt entwickelte Kandidatenmenge F kann auch für MaxGain und mit leichten Modifikationen auch für die Mehr-Radien-Versionen verwendet werden.

Damit entspricht also CSLP dem Set Covering-Problem und MaxGain wird zu k-Median (oder B-Median). Für stetige Maße, also z.B. für Saved Travel Time lässt sich eine diskrete Kandidatenmenge jedoch nicht so einfach finden.

Leider ist Set Covering **NP**-schwer und nach Ergebnissen von Lund und Yannakakis ([LY94]) sowie von Arora et al. [ALMS92] wahrscheinlich nicht einmal auf effiziente Weise bis auf einen konstanten, von der Eingabegröße unabhängigen Faktor approximierbar. Im nächsten Kapitel werden wir dennoch verschiedene exakte und nicht exakte Lösungsverfahren analysieren.

Für k-Median und facility location existieren zwar effizientere Lösungsverfahren, jedoch nur für den Fall, dass das zugrunde liegende Maß eine Metrik ist und insbesondere die Dreiecks-Ungleichung erfüllt ([JV99, GK99, STA97, CGTS99]). In unserem Fall wäre die Metrik aber etwas wie

$$\delta(p, f) := \begin{cases} 1, & \text{falls } \operatorname{dist}(p, f) > r \\ 0 & \text{sonst} \end{cases}$$

und erfüllt somit die Dreiecks-Ungleichung nicht.

2.4 Zwischenbemerkung zur Eingabegröße

Es stellt sich an dieser Stelle die Frage, welcher Wert als Eingabegröße für die im Folgenden vorgestellten Algorithmen verwendet werden soll. Die ursprüngliche Eingabegröße ist $|P|+|G|$, die Anzahl der Siedlungsflächen plus die Größe des Bahnnetzes. Die Anzahl der Kandidaten für das SET COVERING liegt danach zwischen 1 (ein Kandidat für alle Siedlungsflächen) und $(|V|+2\cdot|E|)\cdot|P| = O(|G|\cdot|P|)$ (jeder Kreis um eine Siedlungsfläche schneidet jede Kante in zwei Punkten).

Realistischer ist die Annahme, dass jeder solche Kreis nur eine (von r abhängige) konstante Zahl von Kanten schneidet und somit $|F| = O(|P|)$ gilt, falls das Schienennetz nicht zu engmaschig ist.

Ein gutes Maß für die Eingabegröße des SET COVERING-Problems ist die Anzahl der Einsen in der Überdeckungsmatrix, denn diese entspricht der Anzahl der Terme im LP (2.3). Wir notieren dies als

$$\operatorname{mag} A := \#\{a_{ij} \in A : a_{ij} = 1, i \in P, j \in F\}.$$

Ferner notieren wir

$$\begin{aligned} m &:= |P| &&, \\ n &:= |F| && \text{und} \\ N &:= \operatorname{mag}(A). \end{aligned}$$

Wir werden in Zukunft davon ausgehen das die Überdeckungsmatrix als „doppelte Adjazenzliste“ gespeichert ist, d.h. für jede Zeile bzw. Spalte werden die Indizes der Einträge ungleich Null in der betreffenden Zeile bzw. Spalte der Matrix gespeichert. Dies erlaubt effizientes Iterieren über die Einträge, wie wir es im Folgenden oft brauchen werden, bei einem Speicherplatzverbrauch, der trotzdem nur linear (in $\operatorname{mag}(A)$) ist.

Kapitel 3

Lösungsansätze für CSLP

In diesem Kapitel wollen wir verschiedene Lösungsverfahren für die bisher vorgestellten Probleme diskutieren. Wir wollen uns dabei auf CSLP konzentrieren. Falls nicht ausdrücklich etwas anderes angegeben ist bezieht sich alles Gesagte darauf.

Weiterhin lassen wir den speziellen Charakter unserer praktischen Probleme zunächst außer Acht und betrachten stattdessen das allgemeine Set Covering-Problem.

3.1 Definitionen

Eine ***Instanz*** I von Set Covering ist vollständig bestimmt durch ein Tupel (P, F, c, A) bestehend aus einer Menge von Siedlungsflächen P $(\subset \mathbb{N})$, einer Menge von Kandidaten (engl. *facilities*) F $(\subset \mathbb{N})$, einem Vektor $c \in (\mathbb{R}^+)^{|F|}$ und einer Matrix $A = (a_{ij})$ der Dimensionen $|P| \times |F|$ mit $a_{ij} \in \{0, 1\}$. A heißt ***Überdeckungsmatrix***. Eine 1 in Zeile i und Spalte j bedeutet, dass die i-te Siedlungsfläche vom j-ten Kandidaten erschlossen wird:

Zu jeder Teilmenge $S \subset F$ notieren wir den charakteristischen Vektor von S mit $x_S = (\xi_f)$, wobei

$$\xi_f = \begin{cases} 1, & \text{falls } f \in S \\ 0 & \text{sonst.} \end{cases}$$

Es gilt $\text{cover}_A(S) = \{p \in P \mid (A \cdot x_S)_p \geq 1\}$. S heißt zulässige (Set Covering-)***Lösung*** für I, falls $A \cdot x_S \geq \mathbf{1}$ ist. Dies ist äquivalent zu $\text{cover}_A(S) = P$.

$$\begin{array}{lll} \text{Minimiere} & c^T x, & \\ \text{so dass} & A_{cov} x & \geq \mathbf{1} \\ & x & \in \{0,1\}^{|F|} \end{array} \tag{3.1}$$

Eine Instanz, deren Matrix Null-Zeilen oder -Spalten enthält heiße ***entartet***. Entartete Instanzen haben entweder keine Lösung oder enthalten überflüssige Kandidaten. Zur Vereinfachung der Notation beschränken wir uns bis auf Weiteres auf nicht entartete Instanzen.

Eine ***Teil-Instanz*** $I_1 \subset I$ ist gegeben durch das Tupel

$$P_1 \subset P, F_1 \subset F, c_1 := c_{\restriction} \text{ und } A_1 = (\alpha_{ij})_{i \in P_1, j \in F_1}, \text{ mit } \alpha_{ij} \leq a_{ij}$$

für alle $(i,j) \in P_1 \times F_1$. Den Kosten-Vektor c werden wir häufig auch weglassen, wenn keine Verwechslungen möglich sind.

Gilt $\alpha_{ij} = a_{ij}$, also $A_1 = A_{\upharpoonright P_1 \times F_1}$ so heißt I_1 ***vollständige Teil-Instanz***

Für $f \in F$ und $A = (a_{ij})$ ist

$$\text{cover}_A(f) := \{p \in P \mid a_{pf} = 1\}.$$

Für $p \in P$ ist

$$\text{precover}_A(p) = \{f \in F \mid a_{pf} = 1\}.$$

Falls aus dem Kontext klar ist, welches A gemeint ist, schreiben wir auch nur cover(f) bzw. precover(p).

Gilt $F_1 = \text{precover}_A(P_1)$ für eine Teil-Instanz (P_1, F_1, A_1), so sprechen wir von ihr als einer ***P-induzierten*** Teil-Instanz. Gilt $P_1 = \text{cover}_A(F_1)$ nennen wir sie ***F-induziert***. Eine Teil-Instanz, die sowohl P-induziert als auch F-induziert ist heißt ***Komponente***.

Für eine Teilmenge $F_1 \subset F$ ist

$$I(F_1) := (\text{cover}_A(F_1), F_1, c_\upharpoonright, A_\upharpoonright)$$

die von F_1 ***induzierte*** vollständige Teil-Instanz.

Für $P_1 \subset P$ ist

$$I(P_1) := (P_1, \text{precover}_A(P_1), c_\upharpoonright, A_\upharpoonright)$$

die von P_1 ***induzierte*** vollständige Teil-Instanz.

Sind $I_1, I_2, \ldots, I_n$ vollständige Teil-Instanzen und gilt

$$\text{precover}_A(P_i) \cap \text{precover}_A(P_j) = \emptyset$$

für alle $i \neq j$, so heißen sie ***unabhängig***.

Sind $I_1, I_2, \ldots, I_n$ Teil-Instanzen von I und ist $\bigcup_{i=1,\ldots,n} P_i = P$, so sprechen wir von einer ***Zerlegung*** von I. Eine Zerlegung heißt ***disjunkt*** falls sowohl die P_i als auch die F_i untereinander paarweise disjunkt sind.

In den nächsten Abschnitten möchten wir einige Methoden beschreiben, die eine Reduktion der Größe von Instanzen ermöglicht. Diese Methoden werden nicht nur beim Preprocessing verwendet, sondern sind auch integraler Bestandteil bei den weiteren Lösungsschritten.

3.2 Zerlegung

Ist $I = I_1 \cup I_2 \cup \cdots \cup I_k$ eine disjunkte Zerlegung in vollständige, unabhängige Komponenten, gilt also

$$\dot{\bigcup_{1 \leq i \leq k}} P_i = P, \quad \dot{\bigcup_{1 \leq i \leq k}} F_i = F \quad \text{und} \quad A_i = A_\upharpoonright$$

und darüber hinaus

$$\text{precover}_A(P_i) \cap \text{precover}_A(P_j) = \emptyset \text{ für alle } i \neq j,$$

so können die Teil-Instanzen unabhängig voneinander betrachtet werden.

Dieser Fall ist (bei geeigneter Nummerierung) charakterisiert durch eine Zerlegung von A in Blöcke:

$$A = \begin{pmatrix} A_1 & & & \\ & A_2 & & \\ & & \ddots & \\ & & & A_k \end{pmatrix} \tag{3.2}$$

Diesen Sachverhalt wollen wir im Folgenden kurz formalisieren.

Definition 3.1: Kann eine Matrix A allein durch Vertauschen von Zeilen und Spalten in eine Blockdiagonalgestalt wie in (3.2) gebracht werden (mit $k > 1$) oder enthält sie Zeilen oder Spalten, die keine Eins enthalten, so heißt sie ***zerlegbar***.

Eine Matrix, die nicht durch Vertauschen von Zeilen und Spalten in eine zerlegbare Form gebracht werden kann, heißt ***unzerlegbar***.

Definition 3.2: Eine Instanz, deren Matrix A zerlegbar (bzw. unzerlegbar) ist, heißt ebenfalls ***zerlegbar*** (bzw. ***unzerlegbar***).

Definition 3.3: Sei $I = (P, F, c, A)$ eine Instanz. Der bipartite Graph $G = (V, E)$ mit Knoten $V = P \dot{\cup} F$ und $(p, f) \in E \iff a_{pf} = 1$ heißt ***Überdeckungsgraph*** von I (vgl. Abb. 3.1).

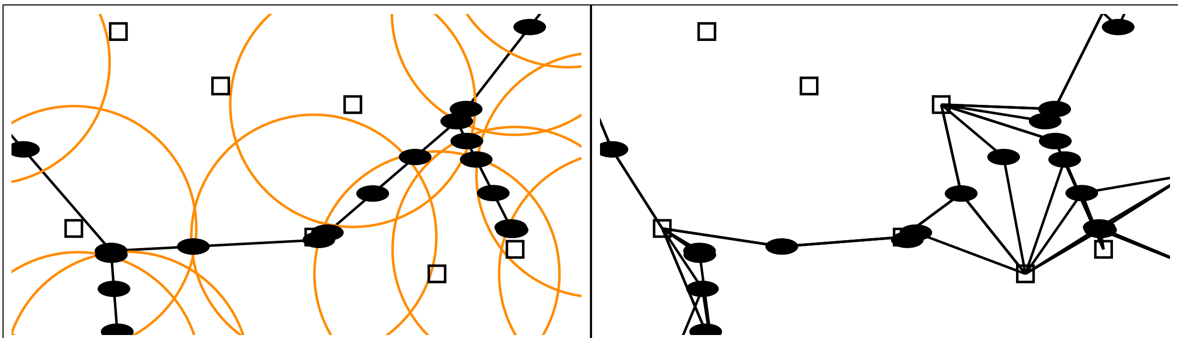


Abbildung 3.1: links: CSLP-Instanz, rechts: Überdeckgunsgraph

Satz 3.4: Es gibt eine eindeutige und disjunkte Zerlegung von I in Komponenten.

Beweis. Dazu muss man lediglich erkennen, dass die Komponenten von I den maximalen Zusammenhangskomponenten des Überdeckungsgraphen entsprechen. Jeder Graph zerfällt eindeutig in Zusammenhangskomponenten, also zerfällt auch I eindeutig in Komponenten. □

Lemma 3.5: Seien I_1 und I_2 zwei Komponenten von I. Dann sind sie unabhängig.

Beweis. Da I_1 und I_2 Komponenten sind, gilt $\text{precover}(P_1) \subset F_1$ und $\text{precover}(P_2) \subset F_2$, also $\text{precover}(P_1) \cap \text{precover}(P_2) = \emptyset$. Also sind sie unabhängig. □

Satz 3.6: Eine Zerlegung in unzerlegbare Komponenten lässt sich in $O(|P| + \text{mag}(A))$ berechnen.

Beweis. In völliger Analogie zur Suche von Zusammenhangskomponenten in Graphen (Breitensuche) liefert Algorithmus 1 die gewünschte Zerlegung. Er wird lediglich dadurch etwas komplexer, dass wir zwei verschiedenen Klassen von Knoten im Überdeckungsgraphen jede für sich behandeln müssen.

Jede Zeile der Matrix und jede Eins wird nur einmal berührt, daraus ergibt sich die behauptete Laufzeitgarantie. □

Algorithmus 1: Komponentenzerlegung

Eingabe : Matrix A

Ausgabe : Familie A_i von nicht leeren Komponenten

```
Start
    Fuer alle Zeilen r wiederhole
        Falls r nicht erledigt dann
            Beginne neue Komponente
            Q_row.push(r)
            Solange Q_row nicht leer wiederhole
                Solange Q_row nicht leer wiederhole
                    r_akt := Q_row.pop()
                    Falls r_akt nicht erledigt dann
                        Markiere r_akt als erledigt
                        Füge r_akt zur aktuellen Komponente hinzu
1                       Fuer alle f ∈ precover(r_akt) wiederhole
                            Falls f nicht erledigt dann
                                Q_col.push(f)
                Solange Q_col nicht leer wiederhole
                    f_akt := Q_col.pop()
                    Falls f_akt nicht erledigt dann
                        Markiere f_akt als erledigt
                        Füge f_akt zur aktuellen Komponente hinzu
2                       Fuer alle r ∈ cover(f_akt) wiederhole
                            Falls r nicht erledigt dann
                                Q_row.push(r)
Ende
```

Zum Abschluss formulieren wir den

Satz 3.7: Sei $I = \dot{\bigcup}_{1 \leq i \leq k} I_i$ die Zerlegung in die vollständigen Komponenten von I. Dann ergibt die Vereinigung der optimalen Lösungen der Teil-Instanzen eine optimale Lösung des gesamten Problems.

Beweis. Die Vereinigung S der Teillösungen S_i ist Lösung von I, da es sich um eine Zerlegung handelt. Da die Teil-Instanzen P-induziert sind, induziert die Optimallösung S^* von I jeweils

eine Lösung $S_i^* = S^* \cap F_i$ für jede Teil-Instanz I_i. Da die S_i optimale Teillösungen sind, gilt $\text{cost}(S_i) \leq \text{cost}(S_i^*)$. Sowohl die S_i als auch die S_i^* sind disjunkt, da die Teil-Instanzen disjunkt sind. Also gilt $\text{cost}(S) = \sum \text{cost}(S_i) \leq \sum \text{cost}(S_i^*) = \text{cost}(I)$. Also ist S optimal. □

Es genügt also, wenn wir uns im Folgenden auf die unzerlegbaren Fälle konzentrieren.

Beispiel 3.1: Ein Beispiel für eine zerlegbare Instanz liefert der Fall, dass jede Siedlungsfläche nur von Kandidaten auf einer einzigen Kante erschlossen wird:

I zerfällt dann in $I_1, \ldots, I_{|E|}$ mit $I_e = I(F_e) = I(\{f \in F | f \in e\})$ für alle Kanten $e \in G$. □

3.3 Kompression

3.3.1 Definition

Definition 3.8: Für einen Vektor $v = (v_1, \ldots, v_n) \in \{0,1\}^n$ sei

$$\text{Tr}(v) := \{i | \ v_i = 1\} \quad (\text{„Träger“ von } v).$$

Für einen Matrix $A = (a_{ij}) \in \{0,1\}^{m \times n}$ seien

$$z_i := z_i(A) := (a_{ij})_{j=1,\ldots,n} \qquad \text{bzw.}$$
$$s_j := s_j(A) := (a_{ij})_{i=1,\ldots,m}$$

die i-te Zeile und die j-te Spalte von A. Dementsprechend seien $\text{Tr}(z_i, A)$ bzw. $\text{Tr}(s_j, A)$ die Träger der entsprechenden Zeilen- bzw. Spaltenvektoren.

Definition 3.9 (Dominanz von Kandidaten):

1. Seien j und ℓ zwei Kandidaten. Falls alle Siedlungsflächen, die von j abgedeckt werden auch von ℓ abgedeckt werden und mindestens eine solche Siedlungsfläche existiert, also

   $$\emptyset \neq \text{cover}(j) \subset \text{cover}(\ell)$$

   gilt, so sagen wir, j werde von ℓ ***dominiert*** und schreiben $j \lesssim \ell$.

2. Seien a_j und a_ℓ zwei Spalten einer Matrix. Falls

   $$\emptyset \neq \text{Tr}(a_j) \subset \text{Tr}(a_\ell)$$

   gilt, so enthält die Spalte ℓ höchstens mehr Einsen und wir sagen, a_j werde von a_ℓ ***dominiert*** und schreiben $a_j \lesssim a_\ell$.

Da ℓ hier alle Siedlungsflächen abdeckt, die j abdeckt, macht es keinen Sinn j gegenüber ℓ vorzuziehen.

Definition 3.10 (Dominanz von Siedlungsflächen):

1. Seien i und k zwei Siedlungsflächen. Falls jeder Kandidat, der k abdeckt auch i abdeckt und es mindestens einen solchen Kandidaten gibt, falls also

$$\emptyset \neq \text{precover}(k) \subset \text{precover}(i)$$

gilt, so sagen wir, die Siedlungsfläche i werde von der Siedlungsfläche k ***dominiert*** und schreiben $i \lesssim k$.

2. Seien a_i und a_k zwei Zeilen einer Matrix. Falls

$$\emptyset \neq \text{Tr}(a_k) \subset \text{Tr}(a_i)$$

gilt, so sagen wir, die Zeile a_i (der Überdeckungsmatrix) werde von der Zeile a_k (die höchstens weniger Einsen enthält) ***dominiert*** und schreiben $a_i \lesssim a_k$.

Da jede Lösung von Set Covering k abdecken muss, bedeutet dies, dass i keine zusätzliche Anforderungen an die optimale Lösung bedingt. Es ist also intuitiv klar, dass dominierte Zeilen und Spalten überflüssig sind.

Zur Klarstellung: Es sind die *kleineren* Spalten und die *größeren* Zeilen, die überflüssig sind.

Bemerkung 3.11: Die Relation $\lesssim$ ist transitiv.

Definition 3.12:

1. Ein ***Kompressionsschritt*** ist ein Tripel $(M, i, k) \in \{row, col\} \times \mathbb{N} \times \mathbb{N}$.

Sei $I = (P, F, c, A)$ eine Instanz und

2. Zeile i werde von Zeile k dominiert und $k \neq i$. Eine solche Instanz heißt ***zeilenkomprimierbar***. Sei $\sigma = (row, i, k)$. Wir bezeichnen mit σA die Matrix, die aus A durch Auf-Null-Setzen aller Einträge der Zeile i entsteht und es sei

$$\sigma I := (P, F, c, \sigma A).$$

3. Spalte j werde von Spalte ℓ dominiert und $j \neq \ell$. Dann heißt I (ungewichtet) ***spaltenkomprimierbar***. Gilt zusätzlich $c_j > c_\ell$, so heißt sie ***gewichtet*** spaltenkomprimierbar. Sei $\tau = (col, j, \ell)$. Es sei τA die Matrix, die durch Auf-Null-Setzen aller Einträge der Spalte j entsteht und

$$\tau I := (P, F, c_{\restriction}, \tau A).$$

4. Eine Instanz, die weder zeilen- noch (gewichtet) spaltenkomprimierbar ist, heißt ***(gewichtet) komprimiert***.

Bemerkung 3.13: Da bei der Komprimierung Null-Spalten und -Zeilen auftreten, müssen wir hier solchermaßen entartete Instanzen wieder zulassen. Wenn wir im Folgenden von einer optimalen Lösung einer entarteten Instanz I sprechen, so meinen wir eigentlich die Lösung der Instanz $I(P \setminus P_0)$ mit $P_0 := \{p \in P |\, \text{precover}(A) = \emptyset\}$.

In der Praxis werden wir später gestrichene Zeilen und Spalten ganz entfernen, so dass sich die Dimension der Matrix verringert. Das oben definierte Vorgehen erleichtert jedoch im Folgenden die Argumentation.

Lemma 3.14: Sei σ ein Kompressionsschritt.

1. Sei $\sigma = (row, i, k)$. Jede optimale Lösung für σI ist auch eine optimale Lösung von I, also

$$OPT(\sigma I) = OPT(I).$$

2. Sei $\tau = (col, j, \ell)$. Jede optimale Lösung von τI ist auch eine optimale Lösung von I, falls $c_j \geq c_\ell$, also
$$OPT(\tau I) \subset OPT(I).$$

3. Für eine Instanz J, die aus einer Instanz I durch eine Folge von Anwendungen von (gewichteten) Kompressionsschritten hervorgegangen ist, gilt
$$\begin{aligned} \text{cost}(J) &= \text{cost}(I) \quad \text{und} \\ \text{OPT}(J) &\subset \text{OPT}(I) \end{aligned}$$

Beweis.

1. Jede Lösung von I ist offensichtlich auch Lösung von σI, also $OPT(I) \geq OPT(\sigma I)$. Aber jede Lösung von σI ist umgekehrt auch Lösung von I, da mit k stets auch i erschlossen ist, also gilt auch $OPT(I) \leq OPT(\sigma I)$.

2. Auch hier ist wieder jede Lösung von σI auch Lösung für I. Falls nun $OPT(\sigma I)$ keine optimale Lösung für I wäre, so müsste $OPT(I)$ den Kandidaten j enthalten. Dann käme man aber durch Austausch von j mit ℓ auf eine Lösung von σI mit kleineren Kosten als $OPT(\sigma I)$, was nicht sein kann.

3. Folgt sofort durch Induktion. □

Beispiel 3.2: Die folgende Matrix kann durch eine Folge von drei Kompressionsschritten, nämlich $(col, 1, 2), (col, 3, 2), (row, 2, 3)$ in eine Matrix umgewandelt werden, die die selbe Optimallösung hat:

$$\begin{pmatrix} \overset{\lesssim}{1} & 1 & \overset{\gtrsim}{0} & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \rightsquigarrow \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{matrix} \\ \wr\wedge \end{matrix} \rightsquigarrow \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Diese zerfällt in zwei triviale Komponenten, so dass wir die optimale Lösung unmittelbar als $OPT = \{2, 3\}$ ablesen können. □

Als nächstes wollen wir zeigen, dass die Form der komprimierten Matrix nicht von der Reihenfolge und Auswahl der Kompressionsschritte abhängt. Dazu müssen wir zunächst einige Vorarbeit leisten.

Bemerkung 3.15: Sei A eine Matrix und B eine Matrix, die durch eine Folge F von Kompressionsschritten aus A hervorgegangen ist.

1. Sei $z_i(B) = 0$. Dann gibt es einen Schritt der Form $(row, i, k) \in F$.

2. Sei $s_j(B) = 0$. Dann gibt es einen Schritt der Form $(col, j, \ell) \in F$ oder es gibt für jeden Eintrag von $s_j(A)$ einen Schritt $(row, i, k) \in F$, der diesen löscht.

3. Der letzte der im vorigen Punkt erwähnten Schritte sei $\sigma = (row, i_0, k_0)$. Es muss eine Spalte j' geben, so dass $A(i_0, j') \neq 0 \neq A(k_0, j')$. (Sonst wäre (row, i_0, k_0) kein gültiger Kompressionsschritt.) Fügen wir vor σ den Schritt (col, j, j') in F ein, so ändert sich B nicht.

Definition 3.16: Gibt es eine Spalte $s_j(B) = 0$, für die es keinen Schritt $(col, j, \ell) \in F$ gibt, so heißt F ***implizit***, sonst ***explizit***. Wir können jede implizite Folge zu einer expliziten erweitern, ohne am Ergebnis B etwas zu ändern.

Bei expliziten Kompressionsfolgen können Zeilen und Spalten also nicht „einfach so verschwinden".

Lemma 3.17: Sei A eine Matrix und $B = \sigma A$ für ein $\sigma = (M, \iota, \kappa)$ und $z_i(A)$ werde dominiert.

1. Falls $M = col$ gilt, so wird $z_i(B)$ ebenfalls dominiert.
2. Falls $M = row$ ist und $z_i(B)$ wird nicht dominiert so ist entweder $\iota = i$ oder es gilt $\kappa = i$ und $z_i(A) = z_\iota(A)$.

Die analoge Aussage gilt auch für Spalten, falls σ explizit ist.

Beweis. Im ersten Fall müsste σ eine Zeile (implizit) gelöscht haben, um das Dominieren aufzuheben. Dies ist aber laut voriger Bemerkung nicht möglich.

Im zweiten Fall muss offensichtlich entweder die Zeile i oder die letzte sie dominierende Zeile k gelöscht worden sein. Also gilt $\iota = k$ und da $\lesssim$ transitiv ist, folgt im zweiten Fall zuerst $\kappa = i$ und dann aus $i \lesssim k$ und $\iota \lesssim \kappa$ auch $z_i(A) = z_\iota(A)$. □

Lemma 3.18: Sei A eine Matrix und $\sigma = (M, i, k)$ und $\tau = (N, j, \ell)$ zwei für A mögliche Kompressionsschritte.

1. Falls $M = N$ und $k = j$ sei $\sigma' := (M, i, \ell)$. Dann gilt $\tau\sigma A = \sigma'\tau A$.
2. Falls $M = N$ und $i = \ell$ sei $\tau' := (N, j, k)$. Dann gilt $\tau'\sigma A = \sigma\tau A$.
3. Sonst gilt $\tau\sigma A = \sigma\tau A$, falls $M \neq N$ oder $i \neq j$, sowie falls $\tau\sigma$ und $\sigma\tau$ explizit sind.

Beweis. Im ersten Fall würde τ die Zeile oder Spalte j löschen und somit σ unmöglich machen. Da $\lesssim$ jedoch transitiv ist folgt die Behauptung mit dem korrigierten σ. Der zweite Fall funktioniert exakt analog.

Falls $M = N$ ist, folgt für $i = j$ natürlich nichts Brauchbares. Falls aber i, j, k und ℓ verschieden sind, ist die Richtigkeit der Behauptung offensichtlich.

Was schließlich den Fall $M \neq N$ betrifft, so ist er lediglich eine Präzisierung von Fall 1 von Lemma 3.17. Die Details überlassen wir dem Leser. □

Nun sind wir in der Lage, das gewünschte Resultat zu folgern.

Satz 3.19: Sei $\mathbb{I} = (\mathbb{A}, \gamma)$ eine Instanz und $I = (A, c)$ und $J = (B, d)$ zwei vollständig komprimierte Instanzen, die aus (I) durch zwei verschiedene Folgen von Kompressionsschritten hervorgegangen sind. Dann gilt

$$\mathrm{cost}(\mathbb{I}) = \mathrm{cost}(I) = \mathrm{cost}(J)$$

und bei geeigneter Sortierung der Siedlungsflächen und Kandidaten von I gilt

$$A = B \quad \text{und} \quad c = d.$$

Beweis. Die Gleichheit der Kosten folgt unmittelbar durch Anwenden von Lemma 3.14.

Œ dürfen wir annehmen, dass

$$F_I = ((M_1, i, k), (M_2, i_2, k_2), \ldots, (M_x, i_x, k_x)) \text{ und}$$

$$F_J = ((N_1, j_1, \ell_1), \ldots, (N_y, j_y, \ell_y))$$

explizit sind. Wir machen Induktion über die Länge x von F_I. Für $x = 0$ ist nichts weiter zu zeigen. $x > 0$ impliziert auch $y > 0$. Laut Lemma 3.17 gibt es entweder einen Schritt der Form $\sigma' = (M_1, i, \kappa)$ oder der Form $\tau = (M_1, \kappa, i)$ in F_J. Lemma 3.17 garantiert auch, dass im zweiten Fall τ durch $\sigma' := (M_1, i, \kappa)$ ersetzt werden kann und Lemma 3.18 sichert schließlich zu, dass σ' gestrichen und durch $\sigma = (M_1, i, \kappa')$ als erster Schritt in F_J ersetzt werden kann, jeweils ohne B zu verändern. Danach liefert die Induktionsvoraussetzung angewandt auf $(M_1, i, k)\mathbb{A} = \sigma\mathbb{A}$, $F_I \setminus (M_1, i, k)$ und $F_J \setminus (M_1, i, \kappa)$ die Behauptung. □

3.3.2 Kompressionsalgorithmen

Kompression ist ein wertvolles Mittel, um die Größe der zu lösenden Instanz zu verringern. In den später behandelten Lösungsalgorithmen wird sie ein wichtiger Bestandteil sein. Im Folgenden befassen wir uns deshalb ausführlich mit der algorithmischen Seite des Kompressionsproblems.

Lemma 3.20: Eine Matrix lässt sich in $O(n^2m)$ bzw. $O(nN) = O(N^2)$ nach Spalten und in $O(nm^2)$ bzw. $O(mN) = O(N^2)$ nach Zeilen komprimieren.

Algorithmus 2: Spaltenkompression

```
Start
  Fuer alle Spalten f wiederhole
    Falls f nicht gelöscht dann
      Falls cover(f) = ∅ dann
        Entferne f;
1     Fuer alle p ∈ cover(f) wiederhole
        Fuer alle f' ∈ precover(p) \ {f} wiederhole
          Falls cover(f') = cover(f) dann
            Entferne f;
          sonst, falls cover(f') ⊂ cover(f) dann
            Entferne f;
          sonst, falls cover(f) ⊂ precover(f') dann
            Entferne f';
Ende
```

Beweis. Algorithmus 2 leistet die vollständige Spaltenkompression. Jede Spalte wird mit jeder anderen verglichen, das sind $O(n^2)$ Vergleiche. Jeder Vergleich benötigt den Vergleich der

Indizes in den Adjazenzlisten. Seien die Spalten der Matrix nach ihrer Länge $m_1 \leq \cdots \leq m_n$ geordnet, so ergibt sich eine Gesamtzahl von

$$\sum_{i \neq j} \max(m_i, m_j) = \sum_{i=1}^{n-1} \sum_{j=2}^{n} m_j$$
$$= \sum_{i=2}^{n} (i-1) m_i = n \sum_{i=2}^{n} m_i$$

Vergleichen. Unter Benutzung von $m_i \leq m$ ergibt sich die erste und mit $\sum m_i = N$ die zweite behauptete Abschätzung.

Kompression nach Zeilen funktioniert genau analog. $\square$

Satz 3.21: Eine Matrix lässt sich in $O(m^2 n^2)$ bzw. $O(Nnm) = O(N^3)$ in eine komprimierte Form bringen.

Algorithmus 3: Kompression

Eingabe : Matrix $A \in \{0,1\}^{m \times n}$, Vektor $c \in \mathbb{R}^n$

Ausgabe : komprimierte Matrix B

Start

  Solange A *komprimierbar* **wiederhole**

    Komprimiere A nach Spalten;

    Komprimiere A nach Zeilen;

  return A;

Ende

Beweis. In jedem Schleifendurchlauf von Algorithmus 3 mit Ausnahme des letzten wird A um mindestens eine Zeile und eine Spalte kleiner. Daher terminiert er nach höchstens $\max(m, n)$ Schleifendurchläufen. Also ist die Laufzeit

$$\begin{aligned} T &\leq mn^2 + m^2(n-1) + (m-1)(n-1)^2 + \ldots \\ &\leq (mn^2 + m^2 n) \min(m, n) \\ &= mn(n+m) \min(m, n) \\ &\leq mn \cdot 2 \max(m, n) \min(m, n) &= 2m^2 n^2, \end{aligned}$$

bzw.

$$\begin{aligned} T &\leq nN + mN + (n-1)N + \ldots \\ &\leq \min(m, n)(m+n)N &= 2Nmn. \end{aligned}$$ $\square$

Der verhältnismäßig naive Ansatz in Algorithmus 2 lässt sich noch verbessern: Zum einen zeigte es sich in der Praxis, dass es nicht unbedingt nötig ist, alle Paare von Siedlungsflächen zu vergleichen.

Stattdessen wählten wir etwa, anstatt in Zeile 1 von Algorithmus 2 über alle $p \in \text{cover}(f)$ zu iterieren, jeweils nur eine zufällige Zeile aus. Das garantiert zwar nicht mehr unbedingt die vollständige Komprimierung, führte aber insgesamt zu einer merklichen Beschleunigung.

Zum anderen ist das Kompressionsproblem äquivalent zu dem Problem, minimale (Zeilenkompression) bzw. maximale (Spaltenkompression) Elemente in einer Menge von Teilmengen einer gegebenen Grundmenge zu finden. Diese Grundmenge ist im Falle der Spaltenkompression die Menge der Siedlungsflächen und jede Spalte entspricht der Menge der Siedlungsflächen, die mit den Einsen der Spalte korrespondieren.

Yellin und Jutla beschreiben in [YJ93] einen Algorithmus, der die Extrema im Durchschnitt in $O(N^2/\log(N))$ und im schlechtesten Fall in $O(N^2/\sqrt{logN})$ findet.

Pritchard beschreibt in [Pri97] einen Algorithmus mit Laufzeit in $O(N^2/logN)$ im schlechtesten Fall. Für den (realistischen) Fall, dass jede Spalte und Zeile höchstens eine (kleine) konstante Zahl von Einträgen ungleich Null hat, und somit $\text{mag}(A)$ in $O(|F|) = O(|P|)$ liegt, terminiert er sogar in $O(N \log N) = O(|P| \log |P|)$.

Damit ergibt sich

Korollar 3.22:

1. Eine Matrix lässt sich in $O(|P| \cdot \text{mag}(A)^2/\log(\text{mag}(A)))$ vollständig komprimieren.
2. Falls die Anzahl der Einträge ungleich Null in jeder Spalte und jeder Zeile kleiner einer von $|P|$ und $|F|$ unabhängigen Konstanten ist, lässt sich die Matrix in $O(|P|^2 \cdot \log(P))$ vollständig komprimieren.

Die Möglichkeiten sind damit aber noch nicht ausgeschöpft. So kann etwa auf dem Gebiete der „online"-Kompression noch viel getan werden. Wie wir später sehen werden (Abschnitt 3.4.4), kommen wir nämlich oft in die Situation, Spalten zur Matrix hinzuzufügen oder Spalten zu entfernen. Wenn es etwa gelingt, von diesen neuen Spalten dominierte oder die neue Spalte dominierende Spalten schnell zu erkennen, könnten beträchtliche Verbesserungen möglich sein. Die Arbeiten von Yellin und Jutla beschäftigen sich ebenfalls mit derartigen Problemen.

3.3.3 Kompression II

Sei $A = \begin{pmatrix} 1 & 1 & 0 & 0 & \\ 0 & 1 & 1 & 0 & \dots \\ 0 & 0 & 1 & 1 & \\ & & 0 & & \ddots \end{pmatrix}$ und $c = (1, 2, 4, 1, \dots)$. Die dadurch bestimmte Instanz ist nicht komprimierbar. Die zweite Spalte von A dominiert jedoch „beinahe" die dritte, es fehlt nur die Abdeckung der dritten Zeile. Betrachtet man jedoch die vierte Spalte, so bemerkt man, dass Spalten 2 und 4 zusammengenommen Spalte 3 dominieren, denn die Summe ihrer Kosten $c_2 + c_4$ ist geringer als die Kosten c_3 von Spalte 3.

Diese Erkenntnis lässt sich zur weiteren Kompression der Instanz nutzen. Wir realisieren dies, indem wir uns zu jeder Zeile p den abdeckenden Kandidaten mit dem geringsten Gewicht $c_{\min}(p)$ merken. Wir sagen, Spalte f' werde von Spalte f dominiert, wenn

$$c_f + \sum_{p \in \text{cover}(f') \setminus \text{cover}(f)} c_{\min}(p) < c_{f'}.$$

Algorithmus 2 wird entsprechend angepasst, wodurch sich die Laufzeit aber nicht verschlechtert. Wir benötigen lediglich ein Preprocessing, dass aber nicht mehr als $O(\text{mag}\, A)$ Zeit in Anspruch nimmt.

3.4 Exakte Lösungsmethoden

3.4.1 Bekannte Verfahren

Allgemeine Lösungsmethoden für lineare Programme sind bereits hinreichend bekannt, vgl. [NW88, NRKT89]. Auch für das SET COVERING-Problem gibt es eine Reihe von exakten Lösungsverfahren, etwa [BH80], [Bea87], [Bea90a] oder [HP93].

Ruf beschreibt in [Ruf02] einen Branch & Bound Algorithmus für eine spezielle Klasse von SET COVERING-Problemen, der z.T. ebenfalls exakte Lösungen für praktische Probleme findet.

3.4.2 Instanzen mit Intervall-Eigenschaft

In diesem Abschnitt betrachten wir Instanzen, deren Überdeckungsmatrix eine besonders einfache Form hat.

Definition 3.23: Eine Matrix hat die ***Intervall-Eigenschaft*** (oder ***consecutive ones property***, kurz ***C1P***), falls sie durch Permutation der Spalten in eine Form gebracht werden kann, wo gilt:

$$a_{ik} = 1 \wedge a_{il} = 1 \wedge k < l \quad \Longrightarrow \quad \forall k \leq j \leq l \quad a_{ij} = 1 \tag{3.3}$$

Die Matrix hat die ***starke C1P,*** falls sie (3.3) erfüllt, ohne dass eine Permutation der Spalten notwendig ist.

Bemerkung 3.24: In der Literatur werden manchmal als Intervall-Matrizen diejenigen bezeichnet, die C1P für Spalten erfüllen. Wir werden diese Unterscheidung hier nicht machen und stattdessen von C1P für Zeilen bzw. für Spalten sprechen.

Beispiel 3.3: Die Matrix

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

hat C1P. Durch Vertauschen von Spalte 3 und 6 entsteht die Matrix

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 \end{pmatrix}$$

mit starker C1P.

Bemerkung 3.25: Hat A (starke) C1P, so hat die durch Entfernen einer Zeile oder einer Spalte entstehende Matrix wieder (starke) C1P.

Bemerkung 3.26: Hat eine Matrix die Intervalleigenschaft, so kann sie durch eine Folge von Kompressionsschritten und anschließende Permutation ihrer Zeilen in eine Einheitsmatrix überführt werden.

Beweis. Sei A die Matrix mit C1P. Ihre Spalten seien entsprechend sortiert. Kann sie bereits durch Zeilenpermutation in die Einheitsmatrix überführt werden, so ist weiter nichts zu zeigen.

Falls die erste Spalte gegen die zweite komprimierbar ist, hat $(col, 1, 2)A$ wieder C1P.

Sonst gibt es einen Index i so, dass $a_{i1} = 1$ ist und $a_{i2} = 0$, also auch $a_{ij} = 0$ für $j >= 2$. Enthält die erste Spalte einen weiteren Eintrag $a_{i'1} = 1$ mit $i' \neq i$, so ist Zeile i' gegen i komprimierbar und $(row, i', i)A$ hat wieder C1P.

Existiert kein solcher Eintrag, so vertauscht man die Zeilen i und 1 und die kleinere Matrix $(a_{ij})_{2 \leq i, 2 \leq j}$ hat wieder C1P.

In den letzten drei Fällen liefert jeweils Induktion die Behauptung. □

Für gewichtete Probleme ist eine solche vollständige Lösung nur über Kompression i.A. jedoch nicht möglich, wie etwa das Beispiel

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \text{ mit Spaltengewichten } c = (1, 2, 1)^T$$

zeigt. (A ist gewichtet nicht komprimierbar.)

Deshalb müssen wir uns um eine andere Lösung unseres Problems für Matrizen, die C1P besitzen, bemühen. Aus der Literatur ist folgendes Ergebnis bekannt:

Satz 3.27: Hat die Matrix einer Instanz die Intervall-Eigenschaft, so ist eine exakte Lösung in polynomialer Zeit berechenbar.

Beweis. In diesem Fall ist die Matrix total unimodular und damit liefert die IP-Relaxation von (3.1) eine exakte Lösung (vgl. etwa [NW88, Kap. 3]). □

Wie wollen uns jedoch um einen direkteren Lösungungsweg bemühen.

In [BL76] ist noch folgendes Resultat zu finden:

Satz 3.28: In $O(\text{mag}(A))$ lässt sich überprüfen, ob eine Matrix C1P hat und die Spalten entsprechend sortieren.

Der Implementationsaufwand der dort beschriebenen Verfahren, die auf PQ-Bäumen basieren, ist jedoch beträchtlich. Da wir es in der Regel ohnehin nicht mit solch einfach strukturierten Problemen zu tun haben, haben wir diesen Weg nicht weiter beschritten.

Wir geben stattdessen zunächst einen Algorithmus an, der für Matrizen, die C1P für Spalten haben, eine Lösung in Linearzeit liefert:

Satz 3.29: Hat die Matrix starke C1P für Spalten, so ist die Lösung in $O(\text{mag}\, A)$ berechenbar.

Beweis. Algorithmus 4 findet die Kosten der Optimallösung. Er kann leicht modifiziert werden, damit er auch die Optimallösung selber zurückgibt.

Da $\{f\} \cup S(I(\{1, \ldots, p'\))$ für jedes f eine Lösung für I ist, folgt $\mathsf{Optimum} \geq \text{cost}(S)$. Sei nun S eine optimale Lösung für I. Diese muss die erste Siedlungsfläche p abdecken, enthält also einen Kandidaten $f \in \text{precover}(p)$. Weiter ist $S \setminus \{f\}$ eine optimale Lösung für $I(P \setminus \text{cover}(f))$ (da S optimal für I ist). Also gilt $\mathsf{Optimum} \leq \text{cost}(S)$ und damit folgt die Korrektheit des Algorithmus.

Algorithmus 4: SCP-C1P-Solver

Eingabe : Instanz I
Vorbedingung : A hat starke C1P für Spalten
Ausgabe : Kosten der Optimallösung

```
Start
    Falls I = ∅ dann
        return 0;
    Optimum := ∞;
    p := 1 ; // erste Siedlungsfläche
    Fuer alle f ∈ precover(p) wiederhole
        p' := erste nicht von f abgedeckte Siedlungsfläche;
1       Kosten := Löse I(P \ {1, ..., p'});
        Falls cost(f) + Kosten < Optimum dann
            Optimum := cost(f) + Kosten;
    return Optimum ;
Ende
```

Um die lineare Laufzeit zu erreichen, muss man lediglich eine Tabelle mit der Optimallösung S_i für $P(P \setminus \{1, \ldots, i\}$ anlegen. Da aufgrund der C1P in Zeile 1 für Spalten nur solche Instanzen auftreten und diese ebenfalls C1P für Spalten besitzen, wird diese Spalte insgesamt nur $O(\text{mag}(A))$ mal erreicht. □

Eine Matrix mit C1P für Zeilen hat nicht unbedingt C1P für Spalten. Ein Gegenbeispiel ist

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Ist die Matrix jedoch zeilenkomprimiert, so garantiert folgender Satz die benötigte Eigenschaft:

Satz 3.30:

1. Eine vollständig zeilenkomprimierte Matrix mit C1P für Zeilen hat C1P für Spalten.
2. Die Matrix A habe starke C1P für Zeilen und sei zeilenkomprimiert. Für jede Zeile i sei l_i die Spalte der am weitesten links gelegenen Eins und r_i die am weitesten rechts gelegene Eins. Seien ferner die Zeilen so sortiert, dass gilt

$$\forall i, j \in \{1, \ldots, |P|\} : i < j \Rightarrow l_i \leq l_j$$

Dann hat A bereits starke C1P für Spalten.

Beweis. Teil 1 folgt aus Teil 2.

Erfüllt A die Voraussetzungen von Teil 2, dann gilt auch

$$\forall i, j \in \{1, \ldots, |P|\} : i < j \Rightarrow r_i \leq r_j,$$

denn sonst wäre Zeile i von j dominiert. Daraus folgt bereits, dass A starke C1P für Spalten hat:

Für jede Spalte k sind Zeilen mit Einträgen ungleich 0 in k genau die, für welche $l_i \leq k$ und $r_i \geq k$ gilt. Diese Menge dieser Zeilen nennen wir M. Die übrigen Zeilen haben entweder $r_i < k$ und liegen somit oberhalb von M oder $l_i > k$ und liegen unterhalb von M. □

Die in Satz 3.30, Teil 2 beschriebene, besonders schöne Form wird im Folgenden kurz ***Normalform*** genannt. Sie ist charakterisiert durch „überlappende“ Zeilen, wie zum Beispiel in

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

Es bleibt also das Problem, auf C1P zu testen und die Matrix in die von Algorithmus 4 benötigte Form zu bringen. Satz 3.28 und Satz 3.30 liefern bereits alles, um dies in $O(\operatorname{mag} A)$ zu leisten.

Als Alternative formulieren wir

Satz 3.31: Algorithmus 5 findet die gesuchte Sortierung in $O(\operatorname{mag} A \cdot m)$. Er findet sogar die Normalform.

Wir wollen hier keinen vollständigen Korrektheitsbeweis führen, die folgenden Bemerkungen sollten es dem Leser jedoch ermöglichen, diesen ggf. selbst nachzuvollziehen.

Zeile 1 Invariante 1: Wenn A Normalform hat, steht in Spalte 1 genau eine Eins, und zwar in Zeile 1. (Sonst wäre Zeile 2 von Zeile 1 dominiert.)

Zeile 2 Eine einzelne Eins kann aber auch in anderen Spalten stehen. Auch dann muss jedoch die nächste (und die vorherige) Spalte der Normalform genau zwei Einträge haben. Wir arbeiten uns von dort erst nach unten („vorwärts“) und dann nach oben („rückwärts“) vor:

$$\begin{matrix} & \longleftarrow & & & \\ \end{matrix}$$
$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & \boxed{1} & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$
$$\begin{matrix} & & & \longrightarrow & \\ \end{matrix}$$

Zeile 3 Wenn die Schleifenabbruchbedingung zutrifft, ist eine „Richtung“ komplett.

Zeile 4 Invariante 2: Wenn A C1P hat, müssen alle diese Spalten in der sortierten Matrix nebeneinander stehen. Aufgrund der Normalform in nicht absteigender Kardinalität.

Zeile 5 Invariante 3: Die Bedingung kann nicht zutreffen, wenn A unzerlegbar ist.

Zeile 7 Initialisierung, nur beim allerersten Durchlauf der „Vorwärtsrichtung“.

Zeile 8 Invariante 4: Die erste Spalte, die für eine neuen Reihe betrachtet wird, darf genau eine Eins weniger haben als die vorhergehende:

Algorithmus 5: C1P-Sorter

Eingabe : Matrix A
Vorbedingung : A zeilenkomprimiert, (gewichtet) spaltenkomprimiert, unzerlegbar
Ausgabe : A in Normalform, falls A C1P hat, sonst „A hat nicht C1P“

```
Start
1      Suche eine Spalte c, in der nur eine 1 steht;
       (r_1, c_1) := Zeile und Spalte dieser 1;
2      Fuer i := vorwärts, rückwärts wiederhole
           r_akt := r_1; c_akt := c_1;
3          Wiederhole
4              Sortiere nicht erledigte Spalten, die bei r_akt eine 1 haben nach der Anzahl der
               Einträge: c_1, ..., c_ν;
5              Falls ν == 0 und nicht fertig dann return „nicht C1P“ ;
6              Fuer j := 1 bis ν wiederhole
                   r_neu := 0;
7                  Falls cperm == ∅ dann
                       r_neu := r_akt;
                   sonst
8                      Falls j == 1 dann
9                          Falls |c_j| == |c_akt| − 1 dann
10                             diff (c_akt, c_j);
11                         sonst, falls |c_j| == |c_akt| dann
12                             r_neu:=diff'(c_akt, c_j);
                           sonst return „nicht C1P“ ;
13                     sonst, falls |c_akt| ≠ |c_j| − 1 dann
14                         Falls r_akt == r_1 und “vorwärts“ dann
                               Beginne nächsten Durchlauf der j-Schleife;
                           sonst return „nicht C1P“ ;
15                     sonst
16                         r_neu:= diff''(c_akt, c_j);
17                 cperm.push_back(c_j), c_akt:= c_j, c_j erledigt;
                   Falls r_neu ≠ 0 dann
18                     rperm.push_back(r_neu);
19             Falls r_akt ≠ rperm.last() dann
                   r_akt :=rperm.succ(r_akt);
               sonst „vorwärts fertig“;
           bis “vorwärts fertig“;
20         rperm.reverse(); cperm.reverse();
21     Wende rperm bzw. cperm auf Zeilen bzw. Spalten von A an und gib das Ergebnis zurück.;
Ende
```

Zeile 9 Invariante 3: In diesem Fall muss c_j genau eine Zeile später anfangen als c_{akt} und in der gleichen Zeile enden:

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Zeile 10 `diff` überprüft, ob $\mathrm{Tr}(s_{\mathsf{c}_j}, A) \dot\cup \mathsf{r}_{akt} = \mathrm{Tr}(s_{\mathsf{c}_{akt}}, A)$.

Zeile 11 Invariante 3′: Im anderen Fall muss c_j genau eine Zeile später anfangen und eine Zeile später aufhören (bezüglich der Normalform) als c_{akt}:

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Zeile 12 `diff′` überprüft, ob $\mathrm{Tr}(s_{\mathsf{c}_j}, A) \setminus \mathrm{Tr}(s_{\mathsf{c}_{akt}} A)$ genau eine neue Zeile enthält, diese wird in r_{neu} gespeichert.

Zeile 13 Invariante 3″: In diesem Fall passt die aktuelle Spalte nicht in die Normalform. Dies folgt daraus, dass keine zwei l_i oder r_i (vgl. Satz 3.30) gleich sind, wenn A in Normalform ist und zeilenreduziert. (Letztere Voraussetzung ist sogar zu stark. Es genügt, dass A keine zwei gleichen Zeilen hat.

Zeile 14 Falls wir in „Vorwärtsrichtung“ sind, überspringen wir c_j und beginnen mit dem nächsten Durchlauf der j-Schleife.

Zeile 15 Invariante 3‴: Hier muss c_j in der gleichen Zeile anfangen und eine Zeile später aufhören (bezüglich der Normalform) als c_{akt}:

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Zeile 16 entspricht Zeile 12

Zeile 17 c_j steht als nächste Spalte fest.

Zeile 18 Falls die vorherige Zeile beendet wurde, wird r_{neu} übernommen.

Zeile 19 Alles wird für die „Rückwärtsrichtung“ vorbereitet: rperm und cperm werden umgedreht und es geht wieder bei der ersten Zeile und Spalte los.

Zeile 20 rperm ist eine Liste von Zeilennummern. Die Ergebnismatrix enthält die Zeilen von A in der Reihenfolge, die durch rperm gegeben ist. Entsprechendes gilt für cperm.

Zeile 21 Nachdem der Algorithmus terminiert, geben cperm und rperm die Permutation der Zeilen bzw. Spalten an, die A in Normalform bringen oder es wurde „nicht C1P“ zurückgegeben.

Zur Laufzeit: In einem Preprocessing-Schritt sortieren wir die Spalten von A nach ihrer Kardinalität. Dies erfordert einen Aufwand von $O(\text{mag}(A) + n\log(n))$.

Zeile 1 hat dann Laufzeit $O(1)$.

Zeile 3 Diese Schleife wird pro Zeile höchstens einmal durchlaufen.

Zeile 4 Dieser Schritt bedeutet keinen weiteren Aufwand, wenn die Spalten schon nach Kardinalität sortiert sind.

Zeile 6 Diese Schleife wird insgesamt höchstens $2\,\text{mag}(A)$ mal durchlaufen: Jeder Durchlauf entspricht einer Eins in Zeile r_{akt}, die höchstens einmal für jede Richtung berührt wird.

Zeile 17 benötigt höchstens $O(\text{mag}(A))$.

Auch die `diff`-Aufrufe benötigen zusammen nur $O(\text{mag}(A))$, da insgesamt jede Spalte hier nur zweimal durchlaufen wird: Einmal als c_j, bevor sie in cperm eingefügt wird und einmal als c_{akt}.

Alle anderen Schritte haben Aufwand $O(1)$.

Wir notieren noch folgende

Bemerkung 3.32: Für den Fall, dass der Graph G aus einer einzigen Kante besteht, hat die zugehörige Matrix die Intervall-Eigenschaft.

Beweis. Es genügt hier, die Kandidaten der weiter oben eingeführten Ordnung $<_e$ entsprechend zu sortieren. Die Details werden in [SHLW02] beschrieben. □

und kommen dann zu dem Schluss:

Korollar 3.33: Für eine Instanz, deren Matrix C1P für Zeilen hat, kann man stets in $O(\text{mag}\,A + n\log n)$ eine optimale Lösung finden.

3.4.3 Real-World-Instanzen

Algorithmus 4 lässt sich auch auf beliebige Instanzen verallgemeinern (Algorithmus 6), allerdings unter Verlust der Laufzeitgarantie. Hier kann nämlich nicht garantiert werden, dass in Zeile 1 nur wenige Teilmengen für die dynamische Programmierung berücksichtigt werden müssen. Für Matrizen, die nur geringfügig von der C1P-Form abweichen, liefert er jedoch gute Ergebnisse, vorausgesetzt, eine geeignete Sortierung der Zeilen kann gefunden werden.

Die in Zeile 1 betrachteten Instanzen I haben stets Siedlungsflächenmengen

$$P_I = P_I^* \cup \{p_{\nu_I}, \dots, p_m\}$$

mit

$$P_I^* \subset \{p_2, \dots, p_{\nu_I - 2}\} \quad \text{und} \quad \nu_I \in \{2, \dots, m+1\}.$$

Weiterhin ist

$$|P_I^*| \le k^* := \max_j (\max\{i : a_{ij} = 1\} - \min\{i : a_{ij} = 1\}).$$

Algorithmus 6: SCP-Solver

Eingabe : Instanz I
Ausgabe : Kosten der Optimallösung
Start
  Falls $I = \emptyset$ **dann**
    return *0*;
  $p := 1$; // erste Siedlungsfläche
  Fuer alle $f \in \text{precover}(p)$ **wiederhole**
1   Kosten := SCP-Solver(I(P \ cover(f)));
      Falls cost(f) + *Kosten* < *Optimum* **dann**
        Optimum := cost(f) + Kosten;
  return *Optimum* ;
Ende

k^* ist die „maximale Spaltenlänge" der Überdeckungsmatrix. Eine erste Abschätzung liefert damit

Lemma 3.34: In Zeile 1 von Algorithmus 6 kommen höchstens $2^{k^*} \cdot |P|$ verschiedene Instanzen vor.

Implementiert man den Algorithmus unter Benutzung einer Dictionary-Datenstruktur, deren Schlüssel Teilmengen von P und deren Werte dem Wert der dazu gehörenden Optimallösung entsprechen, so sind höchstens $\ell \cdot 2^{k^*}$ lesende und $2^{k^*} \cdot |P|$ schreibende Dictionary-Zugriffe nötig.

Beweis. Die Zahl der schreibenden Zugriffe ist gleich der Zahl der verschiedenen Instanzen und ergibt sich unmittelbar aus der eben erwähnten Gestalt der Siedlungsflächenmengen.

Betrachten wir den Rekursionsbaum eines Laufs des Algorithmus. Dieser besteht aus einem Knoten pro Aufruf. Die Wurzel ist der Aufruf mit der vollständigen Instanz. Jeder Knoten hat höchstens ℓ direkte Nachfolger, da die Schleife höchstens so viele Durchläufe hat. Ein Knoten im Baum ist ein Blatt, falls der entsprechende Aufruf entweder die leere Instanz als Parameter hatte, oder falls sich die entsprechende Instanz mit ihrem Wert schon im Dictionary befindet. Eine innerer Knoten entspricht dagegen einem erfolglosen Dictionary-Zugriff. Demnach gibt es höchstens 2^{k^*} innere Knoten und demnach höchstens $\ell \cdot 2^{k^*}$ Blätter. □

Lemma 3.34 gibt zwar nur eine grobe Abschätzung der tatsächlichen Zahl an Teil-Instanzen, trotzdem scheint es aber erstrebenswert, eine Sortierung der Zeilen der Überdeckungsmatrix zu finden, die k^* möglichst klein lässt.

Leider ist das Problem, eine solche Sortierung mit minimaler Spaltenlänge zu finden, **NP**-schwer [Pap76]. Ein Überblick über die Problematik ist in [CCDG82] zu finden.

Der Algorithmus von Cuthill-McKee ist hierfür die gängigste Methode. Er basiert auf Breitensuche. Wir können Algorithmus 1 leicht in diesem Sinne anpassen: Hierfür werden in Zeile 1 bzw. 2 von Algorithmus 1 die Spalten- bzw. Zeilen-Indizes nach steigendem Grad (d.h. nach Anzahl der Einsen der Spalte bzw. Zeile) eingefügt. (Für eine genauere Beschreibung des Algorithmus von Cuthill-McKee siehe [CM69] oder [LS76].)

Beispiel 3.4: Im Vorgriff auf den nächsten Abschnitt und auf den Ergebnisteil wollen wir hier an einem Beispiel belegen, welche Rolle die Reihenfolge der Zeilen spielt. Die Abbildungen 3.2 und 3.3 zeigen beide die selbe (vollständig komprimierte) Instanz, die als Zufallsmatrix erzeugt wurde.

Die beiden Matrizen unterscheiden sich nur durch die unterschiedlicher Zeilen- und Spaltensortierung. In den ersten vier Zeilen sind jeweils die Spaltengewichte aufgelistet. Beide Sortierungen wurden mittels Cuthill-McKee gefunden, jedoch mit unterschiedlichen Startzeilen. Als Laufzeiten ergaben sich ca. 2s im ersten und 615s im zweiten Fall.

Für weitere Beispiele verweisen wir auf Kapitel 4.

3.4.4 Die Kombinationsmethode

Eine Implementation von Algorithmus 6.

Algorithmus 7: Kombi-Solver 1

Eingabe : Instanz **I**
Ausgabe : Optimallösung
Start
  Falls $I = \emptyset$ **dann**
    return *0*;
  Solange $A \neq (1)$ **wiederhole**
1   Wähle ein Paar (r_1, r_2) von Zeilen zum Reduzieren;
    Fuer alle *Spalten* f *von* A *mit einer Eins in Zeile* r_2 *und einer Null in Zeile* r_1 **wiederhole**
      Fuer alle *Spalten* f_1 *mit 1 in Zeile* r_1 **wiederhole**
        Füge Spalte $A_f + A_{f_1}$ mit Gewicht $c_f + c_{f_1}$ zu A hinzu;
      Lösche Spalte f;
      Komprimiere;
    Komprimiere A; // Dabei wird Zeile r_1 gelöscht
  return c_1;
Ende

Lemma 3.35: Sei $I = (A, c)$ eine Instanz und f eine Spalte mit $A_{1f} = 0$. Wir definieren eine neue Instanz J, die aus I entsteht durch Entfernen und Hinzufügen neuer Spalten: Wir entfernen die Spalte f aus A und für jedes i mit $A_{1i} = 1$ fügen wir je eine neuen Spalte $s_i + s_f$ mit Gewicht $c_i + c_f$ zu A hinzu. Dann gilt

$$\text{cost}(I) = \text{cost}(J).$$

Beweis. Der Beweis liefert auch ein Verfahren, eine optimale Lösung für I aus einer optimalen Lösung von J zu gewinnen.

Wir bezeichnen mit φ_i den Kandidaten von J, der aus der Kombination von i und f entstand.

Sei $S = \{s_i |\ i = 1, \ldots, N\}$ eine Lösung von I. Falls $f \notin S$, so ist S eine Lösung von J. Sonst gibt es ein $i \in S$ mit $1 \in \text{cover}(i)$. Dann ist $S \setminus \{i, f\} \cup \{\varphi_i\}$ eine Lösung von J mit gleichen Kosten wir S.

```
911:891:830:620:238:880:684:226:924:931:284:589:687:859:641:932:
307:894:786:229:447:657:907:971:216:640:458:921:187:903:663:20:
276:147:440:668:881:721:432:169:761:513:398:64:482:557:688:951:
266:957:375:815:438:350:439:349:330:532:851:192:739:86:
00: 111...........................................................
01: 1..11.........................................................
02: 1....1111.....................................................
03: 1......1.11111111.............................................
04: .1....1........1.11...........................................
05: .1..1.............1111........................................
06: .1................11..111.....................................
07: .1....1..................11111................................
08: ..1..............1............11..............................
09: ..1.......................1....111...........................
10: ..1.........1.....1................111........................
11: ..1...1.................1.......1.....111.....................
12: ..1.......1.........11...........1.......11...................
13: ..1...........1.....1..........1........1..111................
14: ..1.1........................1...1...11...1...11..............
15: ...1..........1.1..............1................1.............
16: ...1..11.1.................1......1..............1............
17: ...1......1............1..............1....1......11..........
18: ...1................1...1.....1........1....1.....1.111.......
19: ....1.....................................1......1............
20: ....1..........1...1....................1.......1.............
21: .....1.......1..............1.....1...........................
22: .....1.....1....1..............1...............1..............
23: .....1..11...11.1.........1......................1.1..1.......
24: ......1......................11..............1...1....1.......
25: .......1............................11...11......1............
26: .......11...1........1.......1....1..............1............
27: .......1........1.1..1..1......................1...1...1......
28: ........1.......1..........1..........................1.1.....
29: ........1......1.1................1...........1...............
30: ........1...............................1....1.......1...1....
31: ........1................1.............1.....11......1........
32: ...........1.1.1......1.......1.........................1.....
33: ............1.......................................1.....11..
34: .............1..1......1..........................1...........
35: ..............1..1..1..1...............1.........1....1....1..
36: ...............1.....1........1..................1.........1..
37: ..................1....1....1...............1............1....
38: ..................1..1.........1..........1........1........1.
39: ....................1........................1.........1......
40: ......................1..........1...1......................11
41: ........................1....................1..1.............
42: .........................1...............1......1.........1...
43: .............................1...............................1
```

Abbildung 3.2: Überdeckungsmatrix, Sortierung A

```
64:482:761:641:20:447:557:830:284:432:375:815:620:971:532:187:786:
438:350:169:216:663:439:229:266:238:932:924:276:684:307:192:894:
957:931:589:911:687:859:226:458:880:951:739:398:657:349:891:513:
147:903:640:688:440:668:881:721:921:907:851:330:86:
13: 11111111......................................................
17: 1.......111111................................................
37: .1...........1111.............................................
18: .1...1....1.1....111111.......................................
20: ..1....................1111...................................
30: ..1...1.......1...1........1..................................
11: ..1....1.1.........11.......11................................
15: ...11.......1...........1.....1...............................
35: ...1.1.......1.....1..1........111............................
03: ...1....1.................1...1...111111......................
23: ...1.......1..........1....1..1..11...1.11....................
08: ....1..1.............1..........1.............................
22: ....1.........................1....1.....11...................
38: ....1......1....1..........................111................
39: .....11.......................................1...............
05: .....1..........1......1.1...................1.1..............
12: .....1.11...................................11..11............
41: ......1.............1...1.....................................
24: ......1..............11......1...1................1...........
31: ......1...........11.......1.......................11.........
00: .......1............................1..........1..............
09: .......1....................1...........1........1...1........
10: .......1........1....................1................111.....
14: .......1.1...............1................1.1....11.1...1.....
34: ..........1..1................1.......1.......................
27: ...........1....1...1.........1........1..1..11...............
01: ............1............1..........1.........................
16: ............1................1...11....1.............1...1....
06: .............1..1...1..1.......................1..........1...
21: ...............1......................1..1...........1........
07: ...............1.............1..........1......1..11.....1....
04: ................1.........1..1..1..............1..............
33: .................1.............1.....1.....................1..
36: .....................1....1....1.1...........1................
32: .....................1....1........1..1...................1.1.
28: ......................1....1..1..........................1..1.
42: ........................1.......................1..1.......1..
19: .........................1.......1..........1.................
29: ..........................11....1...................11........
02: ...........................1.1......1..1.1....................
26: ...........................1.....1...1.1.....1....1..1........
25: .................................1.....1....1...1......11.....
40: ...........................................1.....1......1.1..1
43: ..................................................1..........1
```

Abbildung 3.3: Überdeckungsmatrix, Sortierung B

Sei $T = \{t_i |\ i = 1, \ldots, M\}$ eine Lösung von J. Falls kein φ_i in T ist, so ist T eine Lösung von I. Sonst gibt es ein $\varphi_i \in T$ und $T \setminus \{\varphi_i \in T\} \cup \{i |\ \varphi_i \in T\} \cup \{f\}$ ist eine Lösung von I mit gleichen oder geringeren Kosten wie T.

Also entspricht jede Lösung von I einer Lösung von J mit gleichen oder geringeren Kosten und umgekehrt. Also ist $\text{cost}(I) = \text{cost}(J)$. □

Beispiel 3.5: Die folgende Instanz ist nicht komprimierbar:

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \qquad c = (1, 3, 1) \qquad F = (1, 2, 3)$$

Wir wählen $r_1 = 1$ und $r_2 = 2$. Der Vektor F gibt die Namen der Spalten wieder.

$$\begin{pmatrix} 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{pmatrix} \qquad c = (1, 3, 1, 1 + 1, 1 + 3) \qquad F = (1, 2, 3, 3 + 1, 3 + 2)$$

Komprimieren nach Spalten:

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \qquad c = (1, 2) \qquad F = (1, 3 + 1)$$

Komprimieren nach Zeilen und dann wieder nach Spalten:

$$\begin{pmatrix} 1 \end{pmatrix} \qquad c = (2) \qquad F = (3 + 1)$$

Daraus liest man die Optimallösung $S = \{1, 3\}$ mit Kosten 2 sofort ab.

Lemma 3.35 gibt also eine Methode an, Spalten zu eliminieren, die eine bestimmte Siedlungsfläche nicht abdecken. Diese Technik nutzen wir in Algorithmus 7. Wir benutzen Lemma 3.35, um nacheinander alle Spalten loszuwerden, die eine Komprimierung von Zeile r_1 gegenüber r_2 verhindern.

Direkter geht Algorithmus 8 vor. Wie in Algorithmus 6 vorgegeben, werden für jede Siedlungsfläche r alle Kandidaten durchprobiert. Die Matrix B enthält dabei am Ende der äußeren Schleife stets alle Teillösungen, die alle bisher gewählten Zeilen abdecken. Der Schlüssel zum Korrektheitsbeweis ist

Satz 3.36: Am Ende eines jeden Durchlaufs der äußersten Schleife von Algorithmus 8 hat $\left(\begin{smallmatrix} A & B \\ 0 & 1 \end{smallmatrix}\right)$ die selbe Optimallösung wie A am Anfang.

Beweis. Bereits vor dem ersten Durchlauf hat $\left(\begin{smallmatrix} A & B \\ 0 & 1 \end{smallmatrix}\right)$ die selbe Optimallösung wie A: Sie zerfällt in die Komponenten A und (1) und der letzte Kandidat hat Kosten 0.

Die innerste Schleife ist nichts weiter als eine Anwendung von Lemma 3.35 auf die Matrix $\left(\begin{smallmatrix} A & B \\ 0 & 1 \end{smallmatrix}\right)$. Das Komprimieren von B ändert ebenfalls nichts an der Lösung.

Das Löschen von Zeile r würde beim anschließenden Komprimieren nach Zeilen ohnehin passieren, da Zeile r von der letzten Zeile von $\left(\begin{smallmatrix} A & B \\ 0 & 1 \end{smallmatrix}\right)$ dominiert wird. □

Anmerkung zum letzten Punkt: Man könnte auch $\left(\begin{smallmatrix} A & B \\ 0 & 1 \end{smallmatrix}\right)$ nach Zeilen und Spalten komprimieren, aber es ist ohnehin unwahrscheinlich das Spalten von A von Spalten von B dominiert werden.

Algorithmus 8 zeigte sich in der Praxis überlegen (vgl. Abschnitt 4.6).

Algorithmus 8: Kombi-Solver 2

Eingabe : Instanz $\mathsf{I} = (P, F, c, \mathsf{A})$
Ausgabe : Optimallösung
Start
  Falls $I = \emptyset$ **dann**
    return *0*;
  Setze Matrix $\mathsf{B} := 0 \in \{0,1\}^{|P| \times 1}$, Gewichtsvektor $d := (0)$;
  Solange $B \neq (1)$ **wiederhole**
1   $r := 1$;
    Fuer alle *Spalten f von B mit 0 in Zeile r* **wiederhole**
      Fuer alle *Spalten g von A mit einer Eins in Zeile r* **wiederhole**
        Füge neue Spalte $\mathsf{A}_g + \mathsf{B}_f$ mit Gewicht $c_g + d_f$ zu B hinzu;
      Lösche Spalte f;
      Komprimiere B nach Spalten;
    Lösche Zeile r von A und B ;
2   Komprimiere A und B nach Spalten und $\left(\begin{array}{c|c} \mathsf{A} & \mathsf{B} \\ \hline 0 & 1 \end{array}\right)$ nach Zeilen;
  return d_1;
Ende

3.4.5 Heuristiken bei der Auswahl der Kombinationskandidaten

Die im vorigen Abschnitt vorgestellten Algorithmen arbeiten die Zeilen der Matrix in einer bestimmten Reihenfolge ab. Wie bereits erwähnt, wirkt sich diese Reihenfolge entscheidend auf die Laufzeit aus. Wir wollen uns hier damit beschäftigen, wie eine gute Reihenfolge gefunden werden kann.

Wie in Abschnitt 3.4.3 erwähnt ist Breitensuche (bzw. Cuthill-McKee) geeignet, die Bandbreite der Matrix gering zu halten. Dieser Methode wohnen jedoch einige schwerwiegende Nachteile inne:

Zum einen hängt die Güte der Lösung stark von der Wahl des Startknoten ab. Kriterien für gute Startknoten zu finden ist Gegenstand aktueller Forschung.

Zum anderen liegt bei Verwendung dieser Methode die Reihenfolge vor Ablauf des eigentlichen Algorithmus fest und wird später nicht mehr angepasst. Zu einem späteren Zeitpunkt möglicherweise vorhandene Informationen können also nicht mehr für die Auswahl der nächsten Zeile genutzt werden.

Wir suchten deshalb nach einer Methode, einen geeigneten Kandidaten für den jeweils nächsten Schritt aus der gegenwärtigen Gestalt der Matrix zu gewinnen. Mit jeder Runde von Algorithmus 8 (d.h. mit jeder bearbeiteten Zeile) werden einige Zeilen „aktiv“ (d.h. sie enthalten eine Eins in B) und einige „verschwinden“ (durch Komprimierung). Es ist offensichtlich wünschenswert, die Zahl der „aktiven“ Zeilen möglichst gering zu halten. Dazu ermitteln wir vor der Auswahl von r die Menge $\mathcal{A}$ der „aktiven“ Zeilen und bewerten jede Zeile z von A nach ihrer Übereinstimmung. Sei dazu $\mathcal{A}_z := \text{cover}(\text{precover}(z))$. Dann ist

$$\mathcal{G}(z) := f(|\mathcal{A}_z \setminus \mathcal{A}|, |\mathcal{A}_z \cap \mathcal{A}|)$$

ein Maß für die Güte von z und die Zeile mit dem höchsten Wert für $\mathcal{G}(z)$ wird als nächste gewählt. Dabei sollte f eine im ersten Argument monoton fallende und im zweiten monoton wachsende Funktion sein. $\mathcal{G}(z) := -2|\mathcal{A}_z \setminus \mathcal{A}| + |\mathcal{A}_z \cap \mathcal{A}|$ lieferte in der Praxis gute Ergebnisse.

Obwohl unsere Implementation nicht sehr trickreich vorgeht (Sie benötigt für jede Runde $O(\text{mag}(A) \log m + m \,\text{mag}(A) \log m)$ Operationen), erzielten wir mit der Heuristik in der Praxis sehr gute Ergebnisse.

Alle zuletzt geschilderten Verfahren haben gemeinsam, dass eine lineare Anordnung der Knoten des Graphen vorgenommen wird. Dabei vernachlässigt man aber unausweichlich gewisse Informationen über die Struktur des Graphen. Wie man diese Information nutzen und trotzdem von Mechanismen wie Kompression profitieren kann ist bisher noch unerforscht.

3.5 Näherungsverfahren

Da im Allgemeinen nicht zu erwarten ist, dass unser Problem einfacher ist als Set Covering, ist der Versuch, eine exakte Lösung zu finden, unter Umständen nicht sehr vielversprechend. Im Rest des Kapitels wollen wir einige Heuristiken zur näherungsweisen Lösung vorstellen.

3.5.1 Heuristische Zerlegungsmethoden

Wie wir in Abschnitt 3.2 gesehen haben, kann das Problem verlustfrei zerlegt werden, wenn die Matrix in Blockdiagonalgestalt gebracht werden kann. Nun werden wir versuchen, das Problem auch dann zu zerlegen, wenn die Matrix keine solche Form hat. Und wir werden sehen, dass der Fehler, den wir dabei machen, durch die Kosten des „Schnitts“ der Teilprobleme abschätzbar ist.

Lemma 3.37: Sei $I = (P, F, c, A)$ eine Set Covering-Instanz, $F = F_1 \dot\cup F_2$, $I_1 := I(F_1)$, sowie $I_2 := (P \setminus \text{cover}(F_1), F_2, c_\restriction, A_\restriction)$. Sei weiterhin $I_0 := I(\text{cover}(F_1) \cap \text{cover}(F_2), F_1)$ und schließlich S_i eine optimale Lösung von I_i $(i = 0, 1, 2)$. Dann ist $S_1 \cup S_2$ Lösung für I und es gilt

$$\text{cost}(S_1) + \text{cost}(S_2) \leq \text{cost}(I) + \text{cost}(S_0)$$

Beweis. Sei S^* Optimallösung von I und $S_i^* := S^* \cap F_i$ $(i = 1, 2)$. Dann ist S_2^* eine Lösung für I_2. S_1^* ist nicht notwendig Lösung von I_1, sicher aber $S_1^* \cup S_0$. Da S_1 und S_2 optimal sind, gilt

$$\text{cost}(S_1) \leq \text{cost}(S_1^* \cup S_0)$$

und

$$\text{cost}(S_2) \leq \text{cost}(S_2^*)$$

und damit, wegen $F_1 \cap F_2 = \emptyset$,

$$\text{cost}(S_1) + \text{cost}(S_2) \leq \text{cost}(S^*) + \text{cost}(S_0).$$

□

Bemerkung 3.38: Dies liefert eine Faktor-2-Approximation, denn $\text{cost}(S_0) \leq \text{cost}(I)$.

Bemerkung 3.39: In Abbildung 3.4 ist der Überdeckungsgraph eines worst-case-Beispiels dargestellt: Die weißen Quadrate seien Siedlungsflächen, die schwarzen Punkte seien Kandidaten mit ihrem jeweiligen Gewicht. Wird nun die Instanz wie durch die großen Kreise angedeutet aufgeteilt, ergibt die Kombination der Lösungen der Teilprobleme Kosten von 2. Die Optimallösung wäre jedoch $1+\varepsilon$. Damit ergibt sich ein Approximationsfaktor, der beliebig nahe bei 2 liegt.

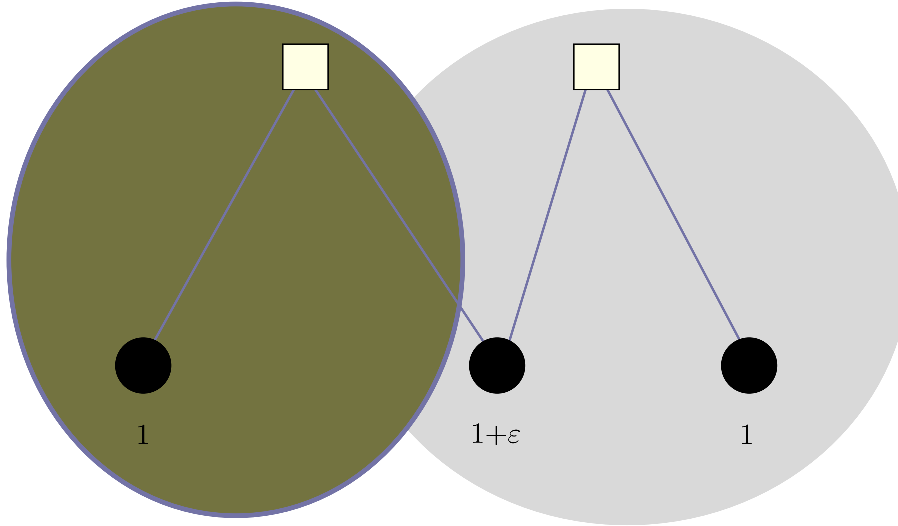


Abbildung 3.4: Beispiel

Bei praktischen Problemen besteht jedoch die berechtigte Hoffnung, dass nur sehr wenige Siedlungsflächen in $\text{cover}(F_1) \cap \text{cover}(F_2)$ liegen und mithin der Fehler sehr klein ist.

Eine andere Methode liefert

Lemma 3.40: Seien $F_1, F_2 \subset F$ mit $F_1 \dot\cup F_2 = F$, $P_1 := P \setminus \text{cover}(F_2)$, $P_2 := P \setminus \text{cover}(F_1)$ und $I_i := I(P_i, F_i)$ für $i = 1, 2$. Weiter sei S_i optimale Lösung von I_i und X irgendeine Lösung für $I_0 := I(\text{cover}(F_1) \cap \text{cover}(F_2), F)$. Dann ist $S := S_1 \cup S_2 \cup X$ Lösung von I und es gilt

$$\text{cost}(S) \leq (S_1) + \text{cost}(S_2) + \text{cost}(X) \leq \text{cost}(I) + \text{cost}(X).$$

Beweis. Sei S^* optimale Lösung von I. Dann ist $S^* \cap F_i$ Lösung von I_i und damit gilt $\text{cost}(S_i) \leq \text{cost}(S^* \cap F_i)$ für $i = 1, 2$. Da $F_1 \cap F_2 = \emptyset$ folgt die Behauptung durch Addition dieser Ungleichungen. □

Bemerkung 3.41: Der Nachteil an dieser zweiten Methode ist, dass I_0 sehr groß werden. Es kann sogar $I_0 = I$ eintreten, wonach nichts gewonnen wäre. Wiederum besteht die praktische Hoffnung, dass I_0 sehr klein wird.

Wir können noch bessere Resultate erzielen indem wir statt I_0 die eventuell kleinere Instanz

$$I(\text{cover}(F_1) \cap \text{cover}(F_2) \setminus \text{cover}(S_1) \setminus \text{cover}(S_2))$$

lösen. Damit erhalten wir trotzdem eine zulässige Lösung für I.

Bei beiden Methoden muss außerdem die Vereinigung der Teillösungen S_1, S_2 und ggf. X auch dann noch nicht „prim“ sein, d.h. wir können eventuell Kandidaten aus der Lösung entfernen, und trotzdem eine zulässige Lösung erhalten.

Wir wollen nun Lemma 3.37 noch auf mehrere Teilinstanzen erweitern.

Lemma 3.42: Sei nun $F = \dot{\bigcup}_{1\leq i\leq k} F_i$, $P_1 := \operatorname{cover}(F_1)$, $P_i := \operatorname{cover}(F_i) \setminus \overline{P}_{i-1}$, wobei $\overline{P}_j := \bigcup_{1\leq i\leq j} P_i$, $I_i := (P_i, F_i, c_{\restriction}, A_{\restriction})$ und S_i optimale Lösung von I_i. $\overline{S} := \bigcup S_i$ ist dann Lösung für I.

X_i sei (optimale) Lösung von $I(\overline{P}_{i-1} \cap \operatorname{cover}(F_i), F_i)$.

Dann gilt

$$\operatorname{cost}(\overline{S}) \leq \operatorname{cost}(I) + \sum \operatorname{cost}(X_i).$$

Beweis. Wir zeigen durch Induktion:

$$\operatorname{cost}(\overline{S}_i) \leq \operatorname{cost}(\overline{I}_i) + \operatorname{cost}(\overline{X}_i) \text{ für } i = 2, \ldots, k,$$

wobei $\overline{S}_j := \dot{\bigcup}_{1\leq i\leq j} S_i$, $\overline{X}_j := \dot{\bigcup}_{2\leq i\leq j} X_i$ und $\overline{I}_i = I(\overline{P}_i, \overline{F}_i)$ ist. Für $i = k$ ergibt sich genau die Behauptung des Satzes.

Für $i = 2$ ist dies gerade Lemma 3.37. Für den Schritt von i auf $i + 1$ bemerken wir zuerst, dass, ebenfalls wegen 3.37,

$$\operatorname{cost}(\overline{I}_i) + S_{i+1} \leq \operatorname{cost}(\overline{I}_{i+1}) + X_{i+1}$$

gilt. Aus der Induktionsvoraussetzung folgt zunächst

$$\operatorname{cost}(\overline{S}_i) - \operatorname{cost}(\overline{X}_i) + S_{i+1} \leq \operatorname{cost}(\overline{I}_{i+1}) + X_{i+1}$$

und daraus direkt die Behauptung. □

Bemerkung 3.43: Für den Fall das die Schnitt-Instanzen $I(\overline{P}_{i-1} \cap \operatorname{cover}(F_i), F_i)$ paarweise disjunkt sind, ergibt sich auch hier eine Faktor-2-Approximation.

Wir können nun unsere bisherigen Algorithmen dahingehend erweitern, dass wir entweder a priori geeignete Schnitte für die Aufteilung suchen und dann die Teilprobleme exakt lösen oder indem wir das Problem erst dann aufteilen, wenn im Verlaufe von Algorithmus 7 oder 4 die auftretenden Matrizen zu groß für eine exakte Lösung werden. Im letzten Kapitel sind einige Ergebnisse aufgeführt. Da jedoch die meisten Probleme bereits exakt leicht lösbar waren, haben wir diese Ideen nicht weiter vertieft.

Kapitel 4

Implementation und Ergebnisse

4.1 Hilfsmittel

Der Großteil der eingeführten Algorithmen wurde in C++ implementiert. Für die Visualisierung wurde die LEDA-Bibliothek [LED] eingesetzt.

Die Anfange des Projektes entstanden im Rahmen einer Zusammenarbeit der Universität Konstanz mit der Deutschen Bahn AG, der an dieser Stelle für die zur Verfügung gestellten Daten gedankt sei.

Die für die Testläufe verwendete Hardware war ein Linux PC mit einem AMD Athlon XP 1500+ Prozessor und 512MB RAM.

4.2 Daten und Parameter

Die Testdaten beschreiben das aktuelle Netz der Deutschen Bahn AG, sowie die Siedlungsflächen Deutschlands.

4.2.1 Reale Daten

Das Netz besteht aus den Bahnhöfen und einigen zusätzlichen Punkten, sowie den Verbindungen dazwischen. Es umfasst ca. 8200 Knoten (davon 6800 Bahnhöfe) und 8700 Kanten. Desweiteren sind für jede Strecke die Querschnittsbelastung, d.h. die Zahl der Passagiere pro Tag im Nahverkehr gegeben. Die Querschnittsbelastung ist sowohl ein Maß für die Verzögerung, die Passagieren durch das Anfahren eines neuen Haltes entsteht, als auch für die Betriebskosten einer Station. Da von diesen beiden Punkten die Kosten eines Haltes wesentlich bestimmt werden, eignet sich diese Größe als Kostenmaß für Strecken. Die Kosten neuer Halte auf „optischen Punkten" werden über die Querschnittsbelastung der adjazenten Strecken berechnet.

Aufgrund einiger Lücken in den Daten hat ein nicht unbedeutender Teil der Strecken die Querschnittsbelastung 0. Da dies die Ergebnisse weitgehend unbrauchbar machen kann (was außerdem einen Indikator für Schwächen des Modells darstellt), wurden drei verschiedene Graphen betrachtet:

1. G ist der unveränderte Graph, einschließlich aller Strecken mit Kosten 0.

2. G' ist der Graph bei dem alle Strecken mit Querschnittsbelastung 0 entfernt wurden. Damit verblieben ca. 6900 Kanten.

3. G'' ist der Graph, wobei jedoch alle Querschnittsbelastungen durch gleichverteilte, zufällige Werte im Intervall $[1, 100]$ ersetzt wurden.

Die ca. 30000 Siedlungsflächen waren auf Gemeindeebene gegeben. Jede Gemeinde war dabei durch ihren Umrisse als Polygone gegeben, deren Schwerpunkt anschließend berechnet wurde.

Da einige Gemeinden sehr groß oder sehr unregelmäßig sind, wurden sie durch Schneiden mit einem rechteckigen Gitter verfeinert, wie im Folgenden skizziert:

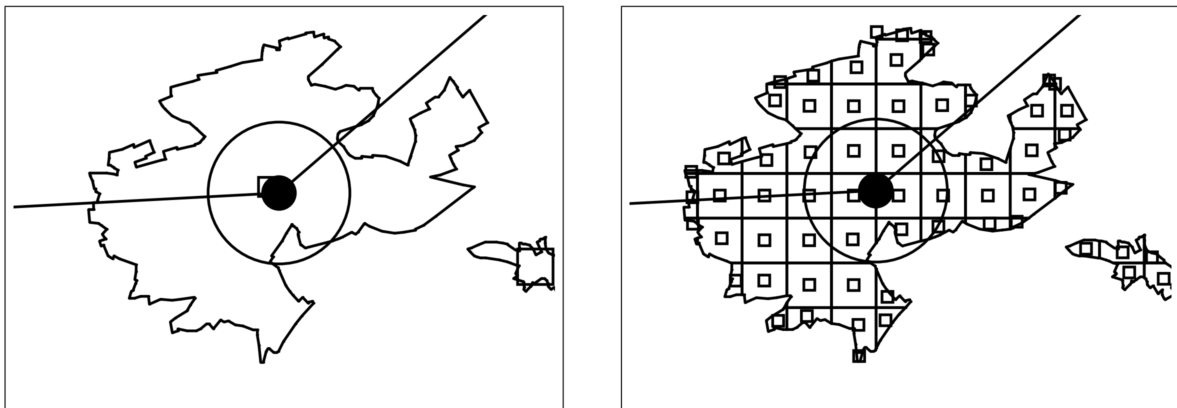


Abbildung 4.1: Gitterverfeinerung
links: Ursprüngliche Siedlungsfläche mit einem Schwerpunkt, der überdeckt wird
rechts: Unterteilung in mehrere Siedlungsflächen mit jeweiligem Schwerpunkt, die nur teilweise überdeckt werden

Dadurch ergeben sich drei Mengen von Siedlungsflächen:

1. P, die Schwerpunkte der Polygone der Gemeinden, ca 20000 Punkte.

2. P', die Schwerpunkte der mit einem Gitter der Maschenweite 2km geschnittenen Polygone, ca. 69000 Punkte.

3. P'', das Selbe mit einem Gitter der Maschenweite 0.5km, ca 265000 Punkte.

4.2.2 Parameter

Aus den Grunddaten lassen sich durch Variation von Parametern sehr unterschiedlich komplexe Instanzen erzeugen. Diese Parameter sind im einzelnen:

- x: Die Auswahl der Strecken (siehe vorheriger Abschnitt).

0	alle Strecken
1	alle Strecken, aber Strecken mit Querschnittsbelastung $<$ 10 bekommen zufällige Querschnittsbelastung
+	nur Strecken mit positiver Querschnittsbelastung
r	zufällige Querschnittsbelastung

- y: Die Menge der gegebenen Bahnhöfe.

a	Alte Bahnhöfen gegeben (CSLP′)
n	Neuplanung aller Halte (CSLP)

- d: Die Maschenweite des Gitters, etwa $d = 2$ und $d = 0.5$ (km). $d = 0$ steht für kein Gitter, d.h. die Instanz mit Siedlungsflächen auf Gemeindeebene.
- r: Der Überdeckungsradius. Wir betrachten die Werte $r = 0.5$km, 1km, 2km, 5km und 10km.

Außerdem unterscheiden wir noch zwei Gruppen von unterschiedlich großen Instanzen:

„Kleine“ Instanzen, „K“ Hier wurde nur ein Ausschnitt der aus dem Südwesten der Deutschland-Instanz verwendet, ca. 3000 Siedlungsflächen und 900 Knoten und Kanten des Bahnnetzes.

„Große“ Instanzen, „D“ Diese enthalten alle Siedlungsflächen Deutschlands und alle Knoten des Bahngraphen.

Wir bezeichnen im folgenden eine Instanz mit $Pxyz(d, r)$, also etwa:

K0a$(\infty, 5)$: Die „kleine“ Instanz mit allen Strecken, vorgegebenen alten Bahnhöfen, ohne Verfeinerung der Siedlungsflächen und mit Radius 5km.

Drn$(0.5, 2)$: Die vollständige Deutschland-Instanz mit zufälligen Streckengewichten, Neuplanung aller Bahnhöfe, Gitter mit Maschenweite 0.5km und Radius 2km.

4.2.3 Sonstige Daten

Zufällige Instanzen, „R“, „S“, „T“ Dies sind Instanzen, die zufällig erzeugt wurden. Hierbei wurden die Knoten des Bahnnetzes und die Siedlungsflächen zufällig, gleichverteilt über eine vorgegeben Fläche verteilt. Anschließend wurde jeder Bahnknoten mit seinen zwei nächsten Nachbarn verbunden und zusätzlich einige wenige zufällige Kanten eingefügt um den Zusammenhang zu erhöhen.

Die entstehenden Bahngraphen sind i.A. nicht zusammenhängend, die entstehenden Überdeckungsmatrizen ähneln jedoch denen von „natürlichen“ Instanzen (optisch und was die Performance unserer Algorithmen betrifft) weit mehr als reine Zufallsmatrizen.

Literatur-Beispiele Als Referenz zum Vergleich mit anderen Algorithmen sollten noch einige Daten aus der Literatur betrachtet werden. Sie sind alle der „OR-Library“ von J.E. Beasley entnommen ([Bea90b]). Es handelt sich dabei um zwei Gruppen von Daten:

Zufällige, „dünne“ Matrizen, „SCP“ Diese wurden auch von Balas und Ho in [BH80] verwendet (dort: Probleme 4.1 bis 4.5). Sie haben die Dimensionen 200×1000 und eine Dichte von 2%, d.h. $\text{mag}(A) = 4000$.

Steiner Triple Systems, „STS“ Diese wurden von Fulkerson, Nemhauser und Trotter in [FNT74] eingeführt und erwiesen sich in der Folge als besonders schwer. Unseres Wissens konnten bisher lediglich die Probleme für $|F| = 9, 15, 27, 45$ und 81 exakt gelöst werden ([FNT74, MS95]).

4.3 Vergleich der Instanzen

Die folgenden Tabellen enthält die Eckgrößen aller Instanzen. Dabei ist

- $|V|$: Anzahl der Knoten des Bahngraphen
- $|E|$: Anzahl der Kanten des Bahngraphen
- $|H|$: Anzahl der aktuell vorhandenen Bahnhöfe
- $|P|$: Anzahl der abdeckbaren Siedlungsflächen
- $|F|$: Anzahl der abdeckenden Kandidaten
- AD_0: ***Erschließungsgrad*** der alten Halte H, definiert durch

$$\frac{|\{p \in P\colon \text{dist}(P,H) \le r\}|}{|P|}.$$

- AD'_0: Erschließungsgrad der alten Halte, jedoch berechnet über die Schnittfläche der Siedlungsflächenpolygone mit den Kreisen mit Radius r um die alten Halte
- AD_1: Erschließungsgrad der gefundenen Optimallösung S = maximaler Erschließungsgrad für Radius r:

$$\frac{|\{p \in P\colon \text{dist}(P,S) \le r\}|}{|P|} = \frac{|\{p \in P\colon \text{dist}(P,F) \le r\}|}{|P|}$$

- AD'_1: Erschließungsgrad der Optimallösung, jedoch berechnet über Polygone
- $|K|$: Anzahl der Komponenten der Überdeckungsmatrix
- $\text{mag}(A)$ Größe der Überdeckungsmatrix
- $|S|$: Kardinalität der gefundenen Optimallösung
- $\text{cost}(S)$: Kosten der Optimallösung
- T: CPU-Sekunden für die Lösung des vollständig komprimierten Problems

Name	r	$\|V\|$	$\|E\|$	$\|P\|$	AD_0	AD'_0	AD_1	$\|S\|$	cost(S)
K	3	917	964	1400	.740	.736	.793	818	4.343e6[1]
K	10	917	964	2943	.984	.984	.988	818	4.343e6
D	3	8178	8735	11200	.710	.673	.771	6829	3.656e7
D	10	8178	8735	27108	.960	.960	.988	6829	3.656e7

Tabelle 4.1: Aktuelle Situation

Status quo

Tabelle 4.1 zeigt die Größe der „realen“ Instanzen, sowie Erschließungsgrad und Kosten der Optimallösung.

Bahnhöfe

Bei gegebenen Bahnhöfen erwies sich das Problem als wesentlich weniger komplex, insbesondere bei großen Radien, da die gegebenen Bahnhöfe das Problem hier typischerweise in viele, sehr kleine Teilprobleme aufspalten bzw. bereits alle Siedlungsflächen überdecken.

Die Neuplanung aller Bahnhöfe erwies sich hingegen als deutlich schwieriger, v.a. bei größeren Radien. Tabelle 4.2 belegt diesen Sachverhalt.

Name	$\|P\|$	$\|F\|$	$\|K\|$	mag(A)	AD_0	AD_1	$\|S\|$	cost(S)
K0a$(0, 2)$	307	703	157	1291	.624	.702	1003	596730
K0a$(0, 5)$	130	314	37	1296	.892	.911	46	106940
K0a$(0, 10)$	47	81	9	1026	.984	.988	9	18790
K0a$(0, 20)$	0	0	0	0	≈ 1	≈ 1	0	0
K0n$(0, 2)$	1305	3881	128	9666	0	.702	526	1.6164e6
K0n$(0, 5)$	2328	7499	2	81279	0	.911	297	617872
K0n$(0, 10)$	2987	13237	2	501989	0	.988	126	139041
K0n$(0, 20)$	3127	23018	1	3126082	0	≈ 1	43	6994

Tabelle 4.2: Alte Bahnhöfe vs. Neuplanung

Strecken

Die Wahl der Streckengewichte hat einen deutlichen Einfluss auf die Schwere des Problems. Tabelle 4.3 fasst die Ergebnisse für die erwähnten Varianten zusammen. Und selbst bei den nur leicht korrigierten Versionen mit $x = 1$ und $x = +$ ist eine deutliche Erhöhung der Kosten der Optimallösung zu beobachten.

Gitter

Bei der Originalaufteilung und Berechnung der Überdeckung mit Hilfe der Schwerpunkte der Siedlungsflächen werden Lösungen generiert, die bei Berechnung der Überdeckung mit

[1] $4.343e6 = 4.343 \cdot 10^6$

Name	$\|P\|$	$\|F\|$	$\|K\|$	mag(A)	AD$_1$	$\|S\|$	cost(S)	T
K0n(0, 2)	1305	3881	128	9666	.702	526	1.6164e6	0.05s
K1n(0, 2)	1305	3881	128	9666	.702	530	2.0135e6	0.06s
K1r(0, 2)	1305	3881	128	9666	.702	548	1.98678e6	0.07
K+n(0, 2)	1256	3339	170	8281	.689	525	1.74041e6	0.05s
K0n(0, 5)	2328	7499	2	81279	.911	297	617872	0.01s
K1n(0, 5)	2328	7499	2	81279	.911	301	796337	0.19s
Krn(0, 5)	2328	7499	2	81279	.911	310	902170	0.1s
K+n(0, 5)	2307	6346	5	69192	.910	302	693861	0.1s
K0n(0, 10)	2987	13237	2	501989	.988	126	139041	0.58s
K1n(0, 10)	2987	13237	2	501989	.988	123	207846	2.78s
Krn(0, 10)	2987	13237	2	501989	.988	130	268324	2.68s
K+n(0, 10)	2987	11027	2	420028	.988	131	155137	0.6s

Tabelle 4.3: Streckengewichte

Hilfe der Polygone einen wesentlich geringeren Erschließungsgrad haben. Dies wird durch den Vergleich der Spalten AD$_1$ und AD$'_1$ von Tabelle 4.4 bei den Instanzen K0n(0,r) belegt.

Die Verfeinerung der Daten führt zu einer deutlichen Milderung dieses Effektes, wie Tabelle 4.4 ebenfalls zeigt. Es ist nicht überraschend, dass diese Instanzen deutlich höhere Kosten aufweisen.

Bei sehr kleiner Maschenweite d wächst die Anzahl der Siedlungsflächen wie $1/d^2$. Bei den hier von uns gewählten Maschenweiten ist jedoch ein weit geringeres Wachstum zu beobachten.

Weiter fällt auf, dass die Größe der Instanzen nach der Kompression höchstens linear mit dem Kehrwert der Maschenweite zu wachsen scheint (und nicht mindestens quadratisch, wie man vermuten könnte).

Name	$\|P\|$	$\|F\|$	$\|K\|$	mag(A)	mag(A'')	AD$_1$	AD$'_1$	$\|S\|$	cost(S)
K1n(0, 1)	885	2616	643	3280	692	.573	.288	690	2.040e6
K1n(2, 1)	2535	5958	442	15107	940	.525	.400	904	3.420e6
K1n(0.5, 1)	10659	24692	282	250196	1299	.511	.484	1274	5.717e6
K1n(0, 2)	1305	3881	128	9666	569	.702	.517	525	1.619e6
K1n(2, 2)	3448	9015	39	54589	664	.693	.629	630	2.190e6
K1n(0.5, 2)	15147	38604	21	1151398	821	.688	.677	796	3.077e6
K1n(0, 5)	2328	7499	1	81279	452	.911	.856	296	619116
K1n(2, 5)	5600	17661	1	481982	430	.909	.889	323	747631
K1n(0.5, 5)	22567	72591	1	9559962	1576	.909	.906	376	993654

Tabelle 4.4: Gittermaschenweite

Radius

Wie schon erwähnt nimmt die Komplexität (gemessen etwa über $|F|$ und mag(A)) bei Instanzen mit gegebenen alten Halten mit dem Radius ab, da bei großen Radien viele Siedlungsflächen schon abgedeckt sind.

Bei der Neuplanung aller Halte hingegen nimmt die Zahl der Einträge pro Spalte etwa proportional zu r^2 zu. Dadurch nimmt die Komplexität der Instanzen mit wachsendem r ebenfalls zu.

4.4 Kompression

Name	$\|P_A\|$	$\|F_A\|$	$\|K_A\|$	mag(A)	$\|P_{A'}\|$	$\|F_{A'}\|$	$\|K_{A'}\|$	mag(A')
K1n(0,2)	1305	3881	127	9666	538	544	507	590
K1n(0,5)	2328	7499	1	81279	385	435	241	1031
K1n(0,10)	2987	13237	1	508919	725	1039	36	21975
K1n(0,20)	3127	23018	1	3126082	1646	2904	3	318969
K1n(2, 5)	5600	17661	1	481982	436	467	258	1336
K1n(0.5, 5)	22567	72578	1	9559962	798	833	313	16047
D1n(0,2)	10785	31898	2366	69681	5275	5378	4969	5697
D1n(0,5)	20071	56737	20	519219	3682	3933	2698	6745
D1n(0,10)	27617	109521	2	3544904	5755	7595	812	136519
Sn(0,2)	1697	5690	406	12790	794	825	720	927
Sn(0,5)	3230	12916	3	119360	642	779	295	1453
Sn(0,10)	3500	23707	1	801781	2215	3625	2	94047
Sn(0,20)	3500	43037	1	5512192	2698	7478	1	900771
SCP41	200	1000	1	4009	200	940	1	3930
SCP42	200	1000	1	3982	200	934	1	3894
SCP43	200	1000	1	3982	200	947	1	3912
SCP44	200	1000	1	3984	200	933	1	3914
SCP45	200	1000	1	4009	200	928	1	3843
STS15	35	15	1	105	35	15	1	105
STS27	117	27	1	351	117	27	1	351
STS45	330	45	1	990	330	45	1	990
STS81	1080	81	1	3240	1080	81	1	3240

Tabelle 4.5: Kompression

Die Kompressionsmethoden lieferten ausgezeichnete Ergebnisse, die in Tabelle 4.5 und Tabelle 4.6 zusammengefasst sind.

A sei die Matrix, die aus den Daten und den vorher beschriebenen Kandidaten gewonnen wird. A' sei die nach Abschnitt 3.3.1 und A'' die nach Abschnitt 3.3.3 komprimierte Matrix. $|F_A|$, $|P_{A''}|$ seien auf die natürliche Art definiert:

- F_A Anzahl der Spalten der Matrix A.
- $F_{A'}$ Anzahl der Spalten der Matrix nach Kompression I
- $F_{A''}$ Anzahl der Spalten der Matrix nach Kompression II

Besonders die realitätsnahen Instanzen, aber auch in geringerem Maße die zufälligen, konnten um mehrere Größenordnungen verkleinert werden. Lediglich die Steiner-Tripel-Instanzen zeigten sich völlig resistent gegen Kompression.

Name	$\|P_{A'}\|$	$\|F_{A'}\|$	$\|K_{A'}\|$	mag(A')	$\|P_{A''}\|$	$\|F_{A''}\|$	$\|K_{A''}\|$	mag(A'')
K1n(0,2)	538	544	507	590	531	537	513	564
K1n(0,5)	385	435	241	1031	328	337	266	463
K1n(0,10)	725	1039	36	21975	193	204	83	444
K1n(0,20)	1646	2904	3	318969	114	114	19	880
K1n(2, 5)	436	467	258	1336	349	363	300	430
K1n(0.5, 5)	798	833	313	16047	480	481	338	1576
D1n(0,2)	5275	5378	4969	5697	5198	5246	5029	5421
D1n(0,5)	3682	3933	2698	6745	3349	3457	2881	4150
D1n(0,10)	5755	7595	812	136519	2176	2237	1189	7060
S1n(0,2)	794	825	720	927	768	794	740	841
S1n(0,5)	642	779	295	1453	532	588	387	764
S1n(0,10)	2215	3625	2	94047	279	295	98	1019
S1n(0,20)	2698	7478	1	900771	89	85	30	273
SCP41	200	940	1	3930	141	136	22	398
SCP42	200	934	1	3894	192	205	4	943
SCP43	200	947	1	3912	190	223	4	1004
SCP44	200	933	1	3914	191	197	4	886
SCP45	200	928	1	3843	190	211	3	948
STS15	35	15	1	105	35	15	1	105
STS27	117	27	1	351	117	27	1	351
STS45	330	45	1	990	330	45	1	990
STS81	1080	81	1	3240	1080	81	1	3240

Tabelle 4.6: Kompression II

Was die Performance anbelangt so deuten die experimentellen Daten auf eine Komplexität von $O(|F|^2)$ für die vollständige Kompression der auftretenden Matrizen hin.

Abbildung 4.2 zeigt für alle Zwischenergebnisse von Algorithmus 8 die Anzahl der Spalten und die Laufzeit bis zur vollständigen Kompression für alle betrachteten R-, S- ,T- und U-Instanzen. In Abbildung 4.3 ist das selbe für die SCP-Instanzen zu sehen.

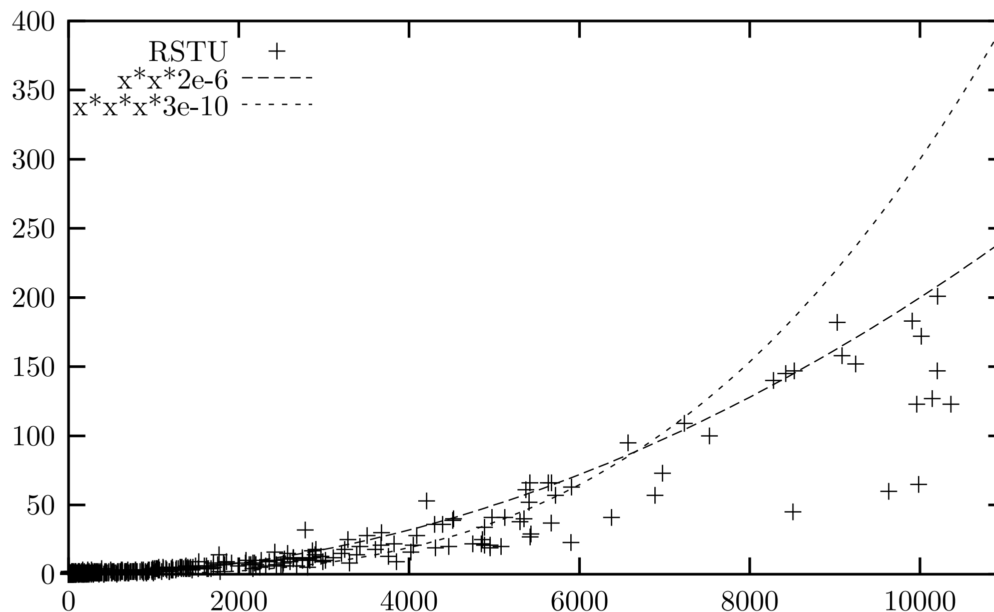


Abbildung 4.2: $|F|$ vs. Kompressionszeit

4.5 Komponenten und Sortierung

Da der Algorithmus zur Komponentenzerlegung in Linearzeit läuft, hat er praktisch keinen Einfluss auf die Gesamtlaufzeit. Die Zerlegung in Komponenten ist bei sorgfältiger Implementation der Lösungsalgorithmen auch nicht entscheidend für deren Performance.

Algorithmus 7 und 8 reagieren, was ihre Laufzeit betrifft, jedoch sehr sensibel auf die Sortierung der Siedlungsflächen, wie schon bei Beispiel 3.4 erwähnt. Abbildung 4.4 und 4.5 zeigen die selbe Instanz (K1n(0, 9), vollständig komprimiert II). Die Abbildungen 4.8 und 4.9 zeigen den jeweiligen Überdeckungsgraphen (dieser ist bis auf Isomorphie unabhängig von der Sortierung der Knoten). Die schwarzen Knoten entsprechen den Siedlungsflächen, die grauen Knoten sind die Kandidaten. Die Siedlungsflächenknoten sind jeweils in der Reihenfolge nummeriert, in der sie ausgewählt wurden. (Deshalb deckt sich die Nummerierung in Abb. 4.9 nicht mit der in Abb. 4.5.)

Beide Ordnungen wurden nach Cuthill-McKee erzeugt (vgl. Abschnitt 3.4.3), in Fall A mit dem Startknoten mit dem Label „0“ in Abb. 4.4, in Fall B mit Startknoten „95“.

Sortierung A ergab eine Laufzeit von unter 2 Sekunden, bei Sortierung B waren es ca. 150 Sekunden.

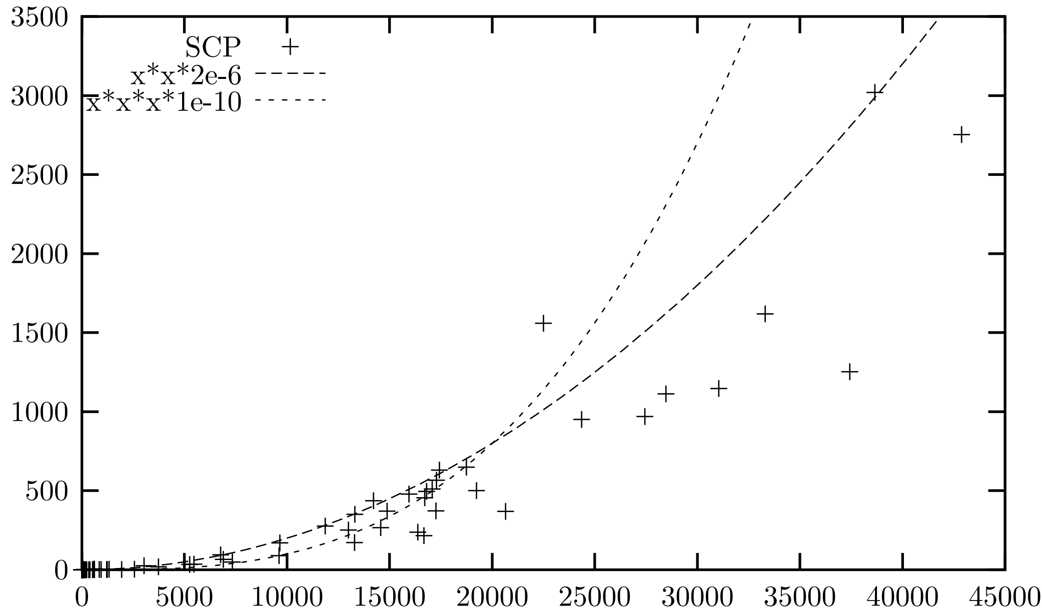


Abbildung 4.3: $|F|$ vs. Kompressionszeit

Die Abbildungen 4.7 und 4.10 zeigen die von der Heuristik in Abschnitt 3.4.5 gewählte Reihenfolge. Die untere Hälfte der Matrix enthält dabei die Zeilen die zwischendurch komprimiert wurden und deshalb nie explizit ausgewählt wurden. Sie tragen auch in Abb. 4.10 keine Nummern. Abbildung 4.6 zeigt das Selbe bei Sortierung A, also ebenfalls in der unteren Hälfte die „unterwegs“ komprimierten Zeilen.

Name	$t_{\min}$(CMK)	$t_{\max}$(CMK)	t_{avg}(CMK)	$t_{\min}$(H)	$t_{\max}$(H)	t_{avg}(H)
r7050	2s	608s	63.9s	4s	57s	18.8s
K1n(0,9)	1s	137s	10.4s	0.2s	0.5s	0.3s

Tabelle 4.7: Cuthill-McKee vs. Heuristik

In Tabelle 4.7 werden die Unterschiede nochmals deutlich: Sie vergleicht die Performance der Lösungsalgorithmen unter Benutzung von Cuthill-McKee-Sortierung (Spalten 2 bis 4) und der Heuristik (Spalten 5 bis 7). Es ist jeweils die minimale, maximale und durchschnittliche Laufzeit über alle möglichen Startknoten aufgezeigt. Die erste Zeile enthält die Ergebnisse einer zufällig erzeugten Matrix (vgl. Beispiel 3.4), die zweite Zeile ist die eben erwähnte Instanz.

Die CMK-Sortierung zeigt sich deutlich sensibler auf die Wahl des Startknotens und insgesamt (wenn auch nicht immer) schlechter als die Heuristik.

```
202:650:462:2114:5764:354:5764:3656:5764:1236:356:1946:356:1946:4806:4806:4564:4564:3656:4564:356:1236:330:356:356:4710:4710:4710:4710:4710:4710:4710:4710:
4710:4710:1452:4710:1452:1628:4385:1628:1628:298:298:1344:1628:522:630:630:330:1114:1114:70:560:70:1344:1628:410:410:1056:412:412:412:412:412:560:560:560:
560:70:70:560:70:410:410:410:410:412:412:410:1342:412:564:632:340:408:70:70:1114:668:668:668:35:70:70:1342:340:788:408:408:408:632:632:632:5308:564:564:
564:1572:564:632:632:206:334:344:528:2544:394:408:408:13500:2004:4491:8982:1572:564:3884:2004:13500:244:344:2544:2894:13500:2920:244:2004:2920:766:214:
3546:642:3546:188:472:456:6452:10750:244:3226:6452:3902:
```

Abbildung 4.4: K1n(0,9), Sortierung A

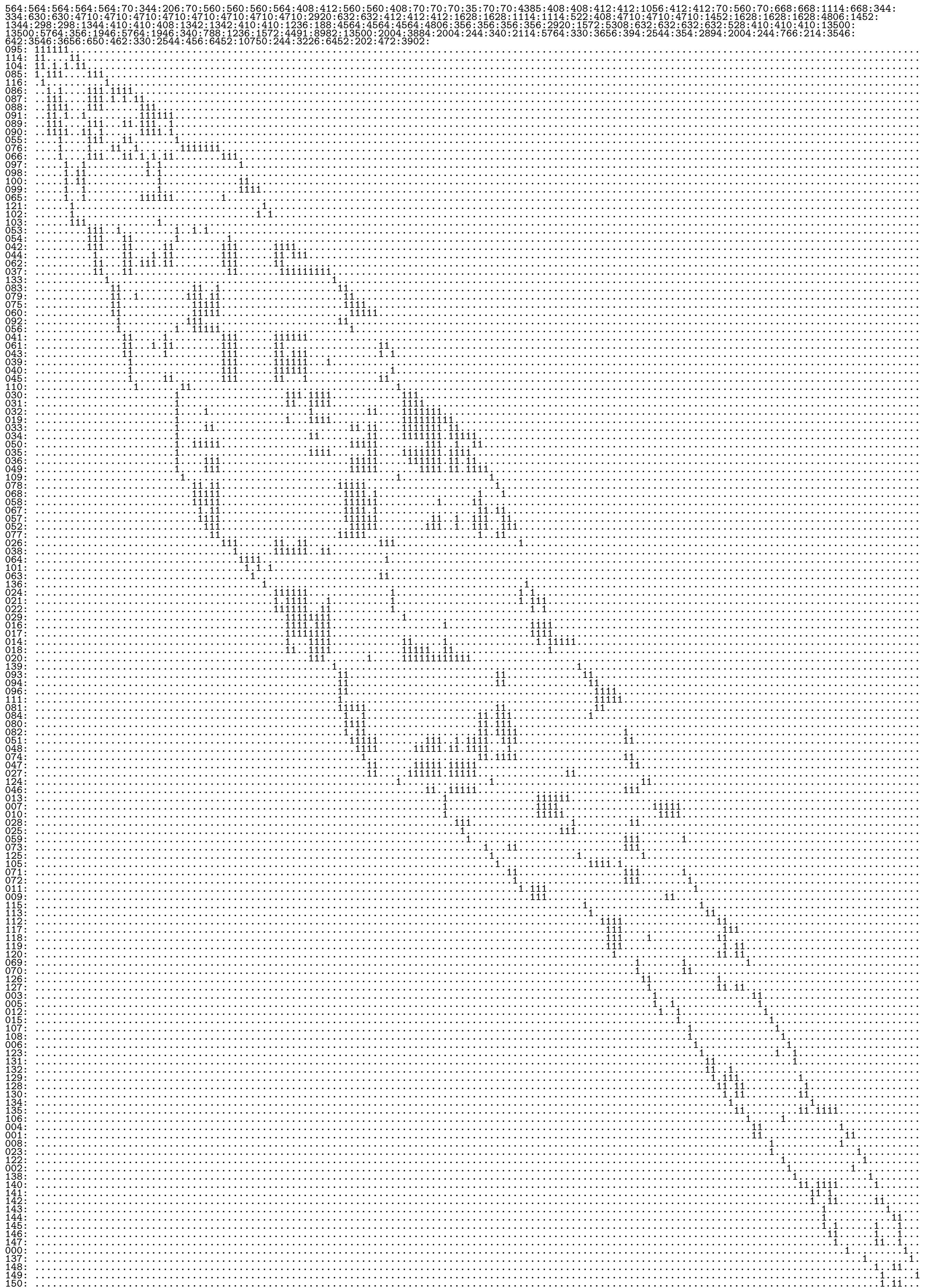


Abbildung 4.5: K1n(0,9), Sortierung B

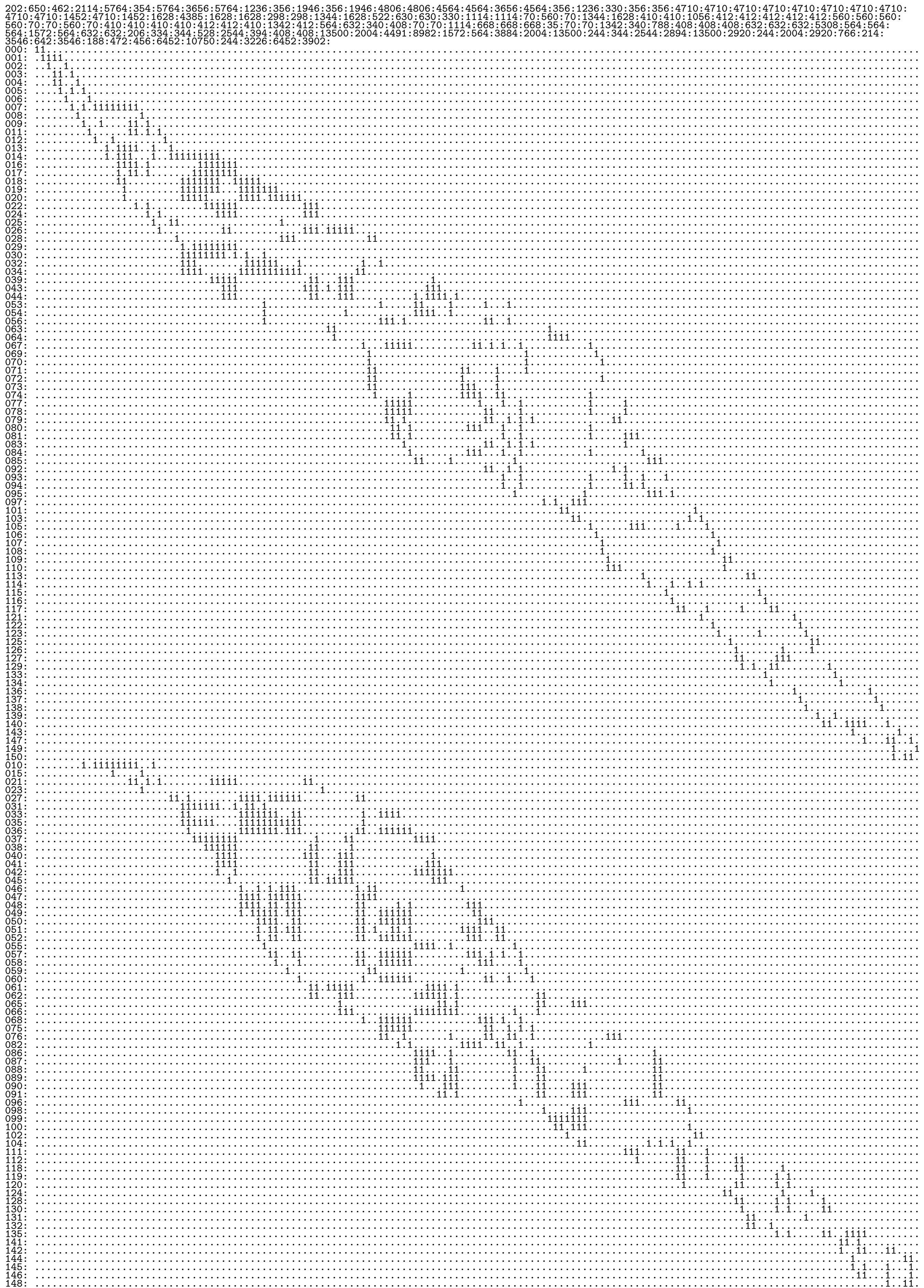


Abbildung 4.6: K1n(0,9), Sortierung A'

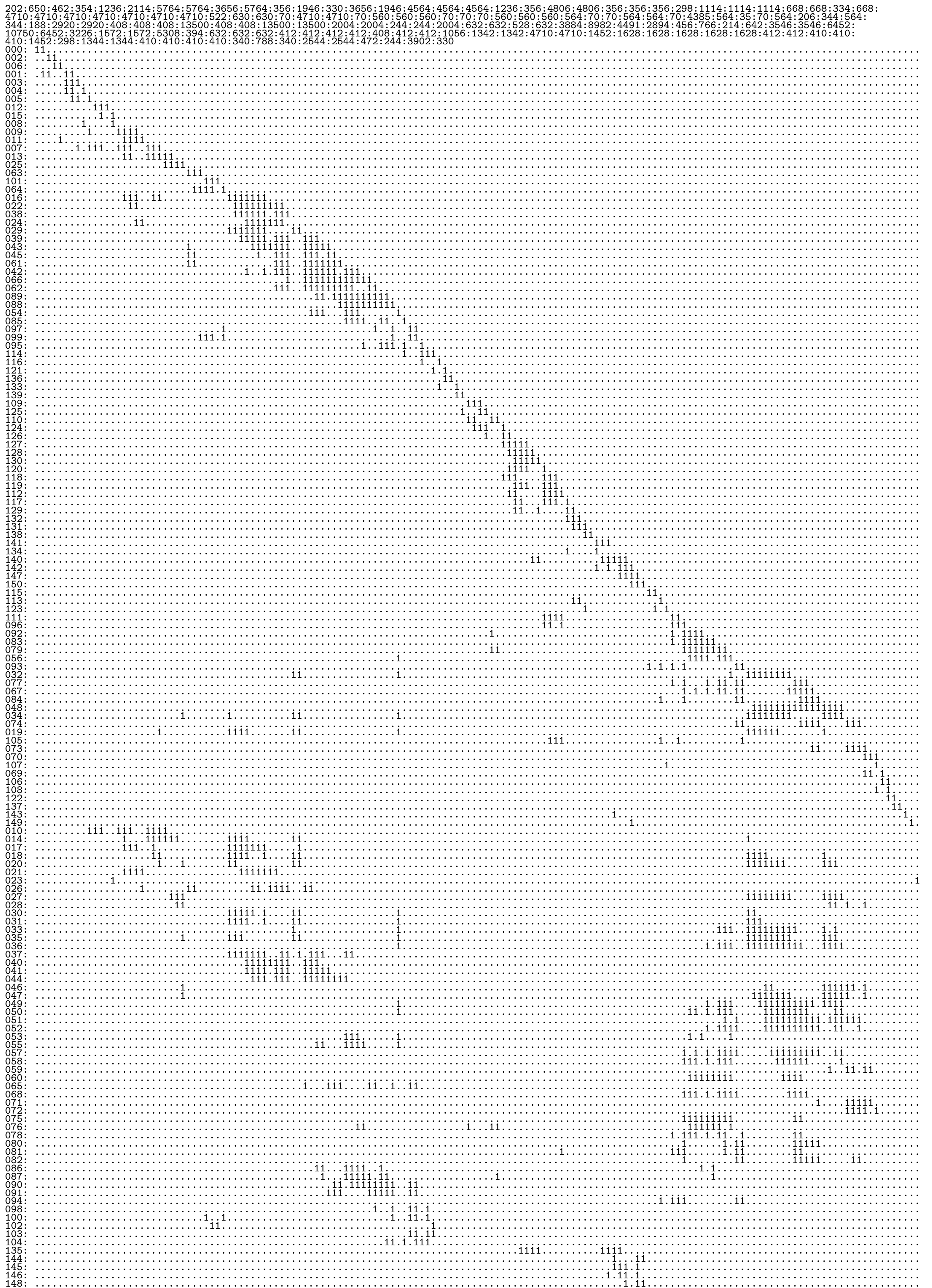


Abbildung 4.7: K1n(0,9), Sortierung C

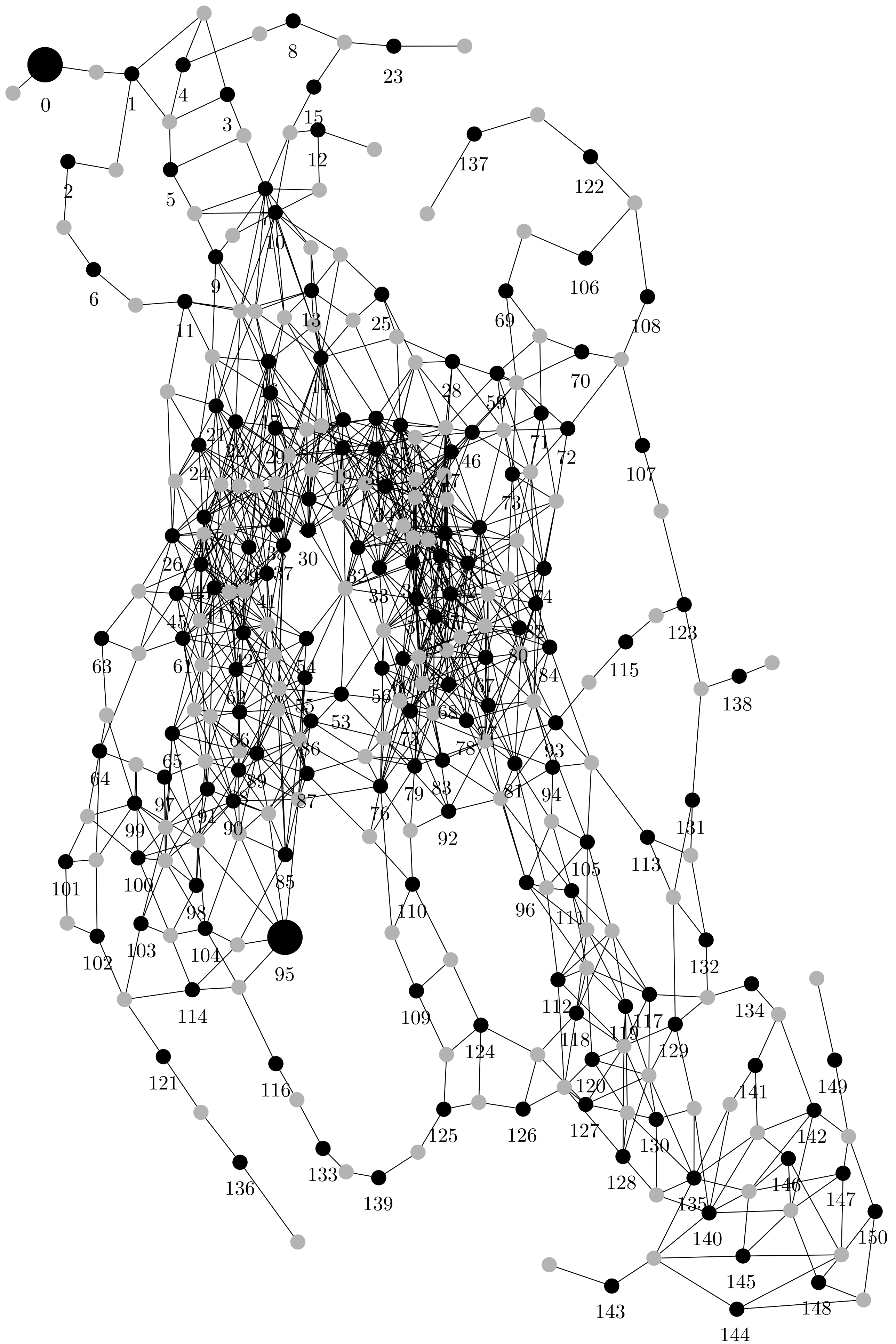


Abbildung 4.8: Überdeckungsgraph, Sortierung A

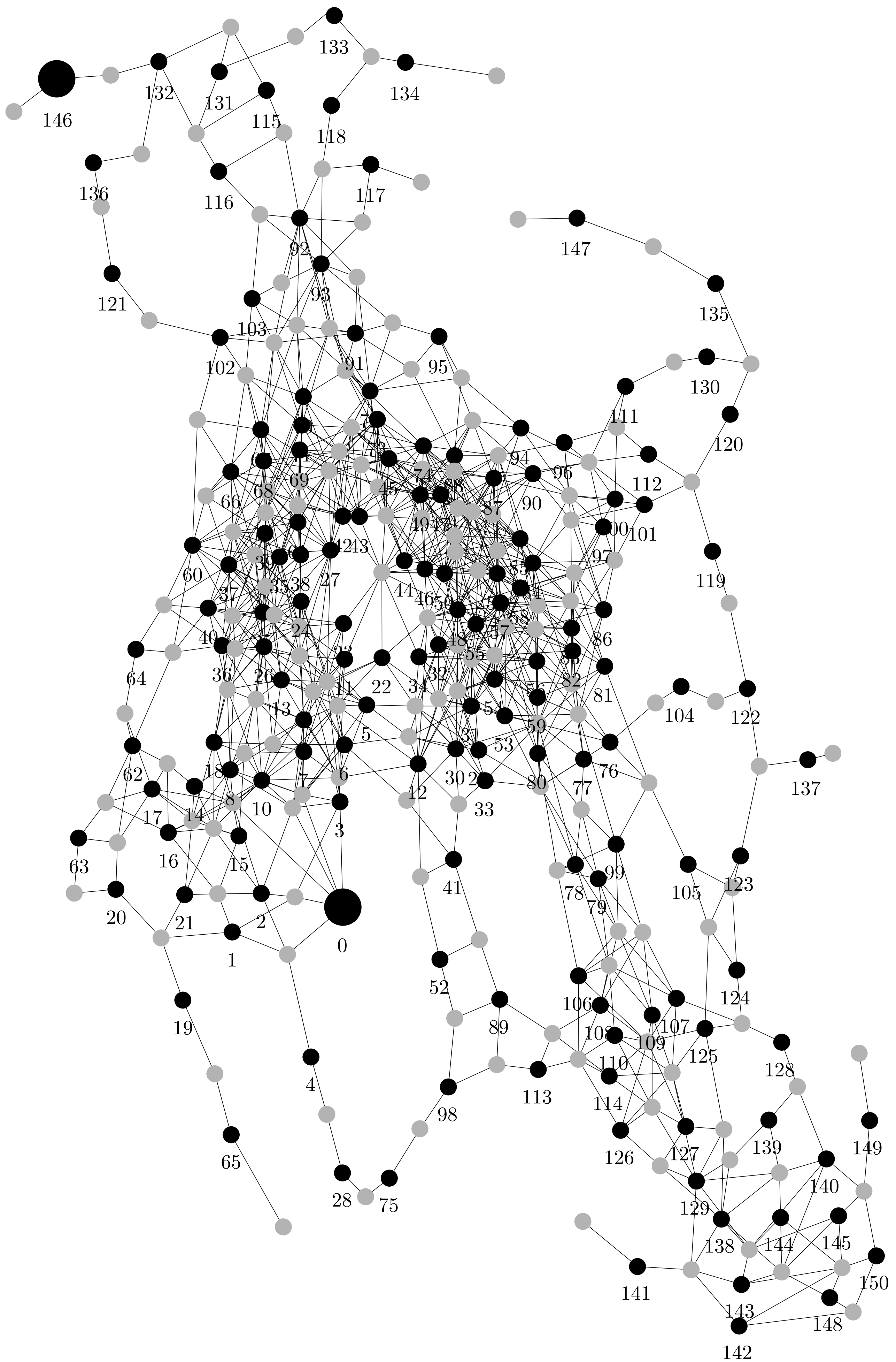


Abbildung 4.9: Überdeckungsgraph, Sortierung B

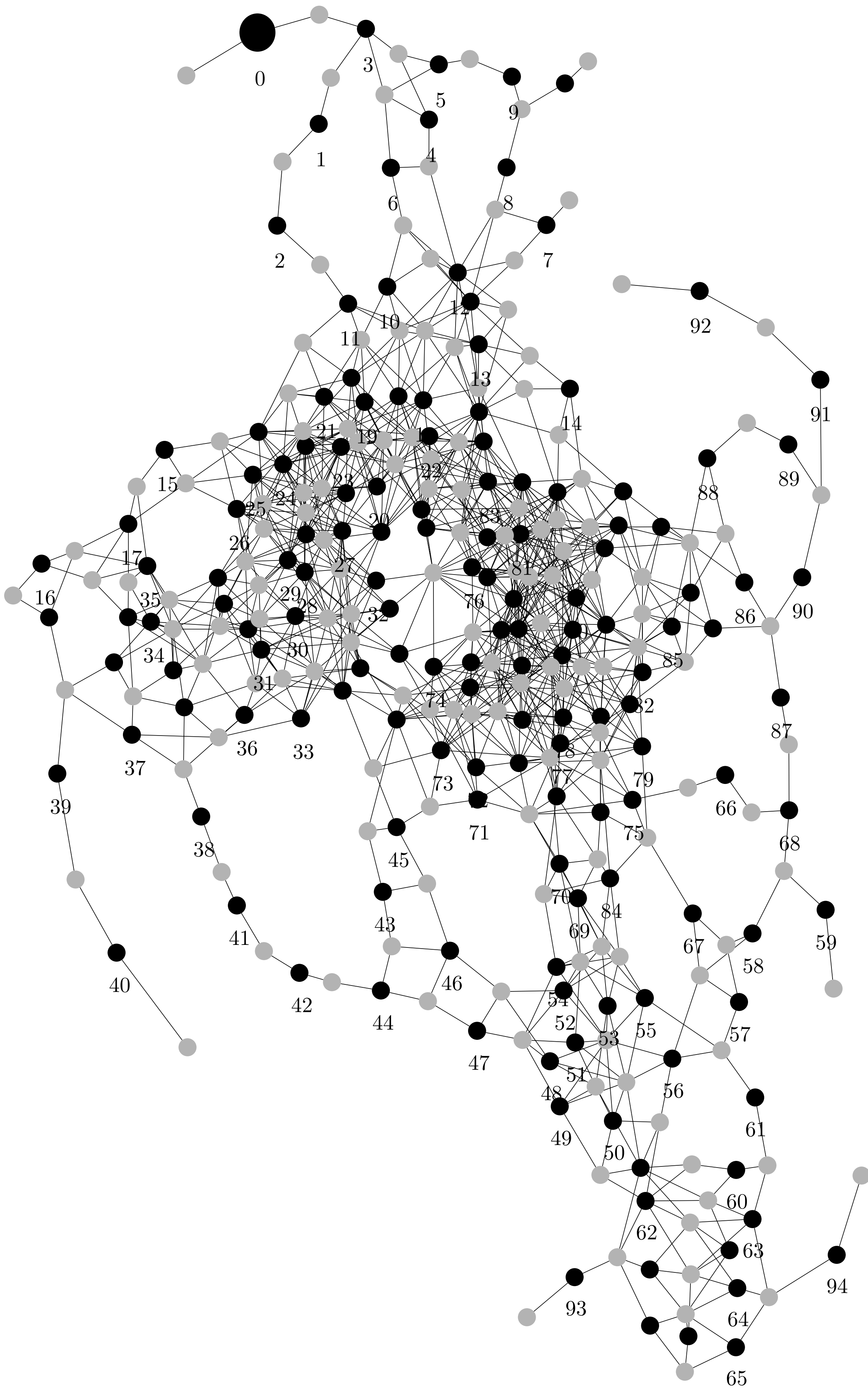


Abbildung 4.10: Überdeckungsgraph, Sortierung C

4.6 Performance der Kombinationsalgorithmen

Die in Abschnitt 3.4.4 beschriebenen Algorithmen leisteten gute Arbeit bei den realen und realitätsnahen Problemen. Tabelle 4.8 fasst die Ergebnisse zusammen.

- $|P_{A''}|$ und $\mathrm{mag}(A'')$ sind Anzahl der Zeilen und Größe der komprimierten Matrix.
- t_1 ist die Laufzeit (CPU-Sekunden) von Algorithmus 7.
- t_2 ist die Laufzeit von Algorithmus 8.
- t_3 ist die Laufzeit von Algorithmus 7 unter Verwendung der in Abschnitt 3.5.1 erklärten Heuristik. Dabei wurde die Instanz immer dann aufgeteilt, wenn die als Zwischenergebnis verwendete Matrix mehr als 1000 Spalten hatte.
- c_{best} sind die Kosten der damit gefundenen Lösung.
- c_{opt} sind die Kosten der optimalen Lösung.
- Die letzte Spalte gibt den relativen Fehler an.

Die exakten Algorithmen waren in der Lage für alle realitätsnahen Problemen eine Lösung zu finden. Das ursprüngliche Problem, nämlich die Instanzen D1a war durch die Reduktion schon nahezu vollständig gelöst.

Die übrigen K-, D-, R-, S-, T- und U-Instanzen konnten ebenfalls alle exakt gelöst werden. Nur eine Instanz benötigte eine Laufzeit von über einer Minute. Algorithmus 7 zeigte sich deutlich unterlegen, einige Instanzen konnten damit nicht in befriedigender Zeit gelöst werden. Die Heuristik zur näherungsweisen Lösung baut auf Algorithmus 7 auf und zeigte sich im Vergleich mit diesem sehr erfolgreich. Sie erzielte dort gute Näherungen in akzeptabler Zeit, wo der exakte Algorithmus versagte.

Ganz anders sieht die Situation bei den Literaturbeispielen aus. Nur eine der SCP-Instanzen konnte exakt gelöst werden, auch das nicht exakte Verfahren zeigte sehr mäßige Ergebnisse. Von den Steiner-Tripel-Instanzen konnten nur die beiden kleinsten exakt gelöst werden.

Name	$\lvert P_{A''}\rvert$	mag(A'')	t_1	t_2	t_3	c_{best}	c_{opt}	%
D1a(0,2)	2154	2156	0.2s	0.3s	–	–	–	–
D1a(0,5)	889	889	0.1s	0.1s	–	–	–	–
D1a(0,10)	226	226	0.0s	0.0s	–	–	–	–
D1a(0,20)	15	15	0.0s	0.0s	–	–	–	–
K1n(0,2)	534	564	0.0s	0.1s	0.0s	1.6186e6	1.6186e6	0
K1n(0,5)	329	452	0.1s	0.1s	0.1s	619116	619116	0
K1n(0,10)	193	444	0.2s	0.1s	0.3s	139843	139843	0
K1n(2,2)	639	646	0.0s	0.1s	0.0s	2.18959e6	2.18959e6	0
K1n(2,5)	349	436	0.0s	0.1s	0.0s	747631	747631	0
K1n(2,10)	232	611	–	0.5s	25.7s	177354	158717	11.7
K1n(.5,2)	803	821	0.0s	0.1s	0.0s	3.07679e6	3.07679e6	0
K1n(.5,5)	488	1576	0.6s	0.7s	0.8s	993654	993654	0
D1n(0,2)	5184	5225	0.0s	0.8s	0.0s	1.58639e7	1.58639e7	0
D1n(0,5)	3372	4201	1715s	0.7s	6.7s	8.32134e6	8.31251e6	0.1
D1n(0,10)	2432	9081	–	46.5s	196s	2.52509e6	2.52415e6	0.0
R1n(0,2)	268	314	–	0.01s	–	–	18884	
R1n(0,5)	149	196	–	0.03s	–	–	7668	
R1n(0,10)	153	1163	–	0.3s	–	–	1266	
S1n(0,2)	777	841	–	0.1s	–	–	55966	
S1n(0,5)	528	747	–	0.1s	–	–	26930	
S1n(0,10)	278	1010	–	0.5s	–	–	3778	
T1n(0,2)	1268	1351	–	0.2s	–	–	44252	
T1n(0,5)	940	1279	–	0.2s	–	–	89582	
T1n(0,10)	745	3711	–	17.5s	–	–	6706	
U1n(0,2)	3115	3483	–	0.6s	–	–	218878	
U1n(0,5)	2112	3096	–	0.7s	–	–	98872	
U1n(0,10)	2554	24225	–	1.75h	–	–	14454	
SCP41	141	398	> 8h	7.51h	160s	517	429	20.5
SCP42	192	943	> 8h	> 8h	713s	794	512	55.1
SCP43	190	1004	> 8h	> 8h	290s	802	516	55.4
SCP44	191	886	> 8h	> 8h	183s	1008	494	104
SCP45	190	948	> 8h	> 8h	488s	780	512	52.3
STS9	12	36	–	0.0s	–	–	5	
STS15	35	105	–	4.6s	–	–	9	
STS27	117	351	–	> 8h	–	–	18	
STS45	330	990	–	> 8h	–	–	30	
STS81	1080	3240	–	> 8h	–	–	61	

Tabelle 4.8: Performance der Lösungsalgorithmen

Omega

Wir haben gesehen, was für Möglichkeiten es gibt, die Wirkung neuer Halte in einem bestehenden Bahnnetz zu modellieren und wie man das zunächst kontinuierliche Problem diskretisieren kann. Desweiteren haben wir einige Möglichkeiten aufgezeigt, die Größe des ursprünglichen, **NP**-schweren CSLP-Problems auf unseren Daten mit geringem Aufwand substantiell zu verringern. Dies funktioniert in der Regel so gut, dass das verbleibende Problem sehr einfach gelöst werden kann. Unserer Meinung nach sind die vorgestellten Methoden durchaus noch erweiterbar, sowohl was eine effizientere Implementierung betrifft, als auch in Richtung neuer, komplexerer Kompressionskriterien.

Die von uns entwickelten Algorithmen zur Lösung beliebiger Probleme zeigten sich ebenfalls (wiederum v.a. auf realen Daten) sehr leistungsfähig. Auch hier ist noch genug Luft für weitere Verbesserungen vorhanden bei der Beschneidung der im Verlaufe der Kombinationsalgorithmen betrachteten Teilloßungen, bei der Auswahl der jeweils nächsten Siedlungsfläche zur Reduktion und auch bei der Weiterentwicklung der nicht exakten Methoden. So liegen etwa noch keine Erfahrungen über deren Anwendung auf Algorithmus 8 vor.

Auch der Komplexitätsstatus von CSLP und anderen Problemen ist noch weitgehend ungeklärt. Zwar sind die meisten als **NP**-schwer bekannt, jedoch liegen noch keine Ergebnisse über Approximierbarkeit vor. Weiter könnte es interessant sein, Kriterien zu finden, welche Instanzen (wie in der Realität beobachtet) einfach und welche wirklich schwer sind.

Ein weiterer vielversprechender Schritt ist die Anwendung der gewonnenen Erkenntnisse auf andere Probleme, wie zum Beispiel die Mehr-Radien-Version oder die MaxGain- und Facility Location-Probleme.

Literaturverzeichnis

[ALMS92] S. Arora, C. Lund, R. Motwani, and M. Szegedy. Proof verification and intractibility of approximation problems. *Proc. 33rd IEEE Symposium on Foundations of Computer Science*, pages 14–23, 1992.

[Bea87] J. E. Beasley. An algorithm for the set covering problem. *European Journal of Operational Research*, 31:85–93, 1987.

[Bea90a] J. E. Beasley. A lagrangian heuristic for the set-covering problems. *Naval Research Logistics*, 37:151–164, 1990.

[Bea90b] J.E. Beasley. OR-Library. http://www.ms.ic.ac.uk/jeb/jeb.html, http://mscmga.ms.ic.ac.uk/info.html, 1990.

[BH80] E. Balas and A. Ho. Set covering algorithms using cutting planes, heuristics and subgradient optimization: A computational study. *Mathematical Programming*, 12:37–60, 1980.

[BL76] K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *Journal of Computer and System Science*, 13(3):335–379, 1976.

[CCDG82] P. Chinn, J. Chvtalov, A. Dewdney, and N. Gibbs. The bandwidth problem for graphs and matrices – a survey. *J. Graph Theory*, 6:223–254, 1982.

[CGTS99] Moses Charikar, Sudipto Guha, Eva Tardos, and David B. Shmoys. A constant-factor approximation algorithm for the k -median problem (extended abstract). In *ACM Symposium on Theory of Computing*, pages 1–10, 1999.

[CLR97] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT press, Cambridge, Massachusetts, 1997.

[CM69] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proc. ACM National Conference*. Association of Computing Machinery, 1969.

[FNT74] D. R. Fulkerson, G. L. Nemhauser, and L. E. Trotter. Two computational difficult set covering problems that arise in computing the l-width of incidence matrices of steiner triple systems. *Mathematical Programming Study*, 2:72–81, 1974.

[GK99] S. Guha and S. Khuller. Greedy strikes back: Improved facility location algorithms. *Journal of Algorithms*, 31:228–248, 1999.

[HLS+01] Horst W. Hamacher, Annegret Liebers, Anita Schöbel, Dorothea Wagner, and Frank Wagner. Locating new stops in a railway network. *Electronic Notes in Theoretical Computer Science, Proc. ATMOS 2001*, 50(1), 2001.

[HP93] Karla A. Hoffman and Manfred Padberg. Solving air crew scheduling problems by branch and cut. *Management Science*, 39(6):657–682, 1993.

[JV99] Kamal Jain and Vijay V. Vazirani. Primal-dual approximation algorithms for metric facility location and k-median problems. In *IEEE Symposium on Foundations of Computer Science*, pages 2–13, 1999.

[KPS+02] Evangelos Kranakis, Paolo Penna, Konrad Schlude, David Scot Taylor, and Peter Widmayer. Improving customer proximity to railway stations. In *ATMOS 2002*. ATMOS, 2002.

[LED] LEDA. Library of Efficient Data types and Algorithms. http://www.mpi-sb.mpg.de/LEDA; http://www.algorithmic-solutions.com.

[LS76] W. H. Liu and H. Sherman. Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices. *SIAM J. Numer. Anal.*, 13:198–213, 1976.

[LY94] Carsten Lund and Mihalis Yannakakis. On the hardness of approximating minimization problems. *Journal of the ACM*, 41(5):960–981, September 1994.

[MS95] C. Mannino and A. Sassano. Solving hard set covering problems, 1995.

[NRKT89] G.L. Nemhauser, A.H.G.Rinnooy Kan, and M.J. Todd. *Optimization*. Elsevier Science Publishers B.V., Amsterdam, 1989.

[NW88] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. Wiley, 1988.

[Pap76] C. H. Papadimitriou. The NP-completeness of the bandwidth minimization problem. *Computing*, 16:263–270, 1976.

[Pri97] Paul Pritchard. An old sub-quadratic algorithm for finding extremal sets. *Information Processing Letters*, 62:329–334, 1997.

[Ruf02] Nikolaus Ruf. Locating train stations: Set covering problems with almost consecutive ones property. Master's thesis, Deparment of Mathematics, University of Kaiserslautern, September 2002. Supervised by Prof. Dr. Horst W. Hamacher and Dr. Anita Schöbel.

[SHLW02] Anita Schöbel, Horst W. Hamacher, Annegret Liebers, and Dorothea Wagner. The continuous stop location problem in public transportation networks. *Report in Wirtschaftsmathematik*, 81, 2002.

[Shm95] D. B. Shmoys. Computing near-optimal solutions to combinatorial optimization problems. Technical report, Ithaca, NY 14853, 1995.

[STA97] David B. Shmoys, Eva Tardos, and Karen Aardal. Approximation algorithms for facility location problems (extended abstract). In *ACM Symposium on Theory of Computing*, pages 265–274, 1997.

[Tho01] Mikkel Thorup. Quick k-median, k-center, and facility location for sparse graphs. In Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Crete, Greece, July 8-12, 2001, Proceedings*, volume 2076 of *Lecture Notes in Computer Science*, pages 249–260. Springer, 2001.

[YJ93] D. M. Yellin and C. S. Jutla. Finding extremal sets in less than quadratic time. *Inform. Process. Lett.*, 48:29–34, 1993.

Index

A

Abstand 7

B

B-median 16

C

C1P 30–36
C1P
  stark 30
consecutive ones property . . . . . *siehe* C1P
CSLP 8–11, 13–16, 19, 21, 49, 67
  Definition 8

D

dominiert 23, 24

E

Einnahmen 8
endliche dominierende Menge 14
entartet 19
Erschließungsgrad 9, 50
erschlossen 7

F

Facility Location 10, 16, 67

I

Instanz 19
Intervall-Eigenschaft 30

K

k-center 10
k-median 10, 16
Komponente 20
Kompressionsschritt 24
komprimiert 24
Kosten 7

L

Lösung 19

M

MaxGain 9–11, 16, 67
MinStation 8

O

optische Punkte 5

P

p-cover 9, 10

S

Saved Travel Time 10, 11, 16
SetCover 11, 13, 15–17, 19, 24, 30, 43
Siedlungsflächen 5
spaltenkomprimierbar 24

T

Teil-Instanz 19
  F-induziert 20
  P-induziert 20
  induziert 20
  vollständig 20
Teil-Instanzen
  unabhängig 20

U

Überdeckungsgraph 21
Überdeckungsmatrix 19
unzerlegbar 21

Z

zeilenkomprimierbar 24
zerlegbar 21
Zerlegung 20