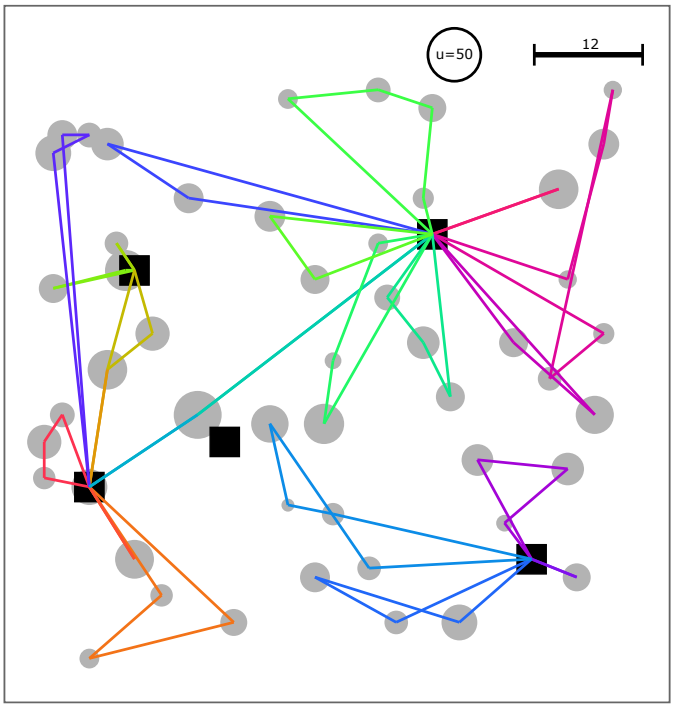
Universität Augsburg

Institut für Mathematik

Ausarbeitung

zum Programmierprojekt

Capacitated Location Routing with Hard Facility Capacities



von:
Lukas Graf

Betreut von:
Prof. Dr. Tobias Harks

Inhaltsverzeichnis

1 Capacitated Location Routing (CLR) 3
1.1 Problemdefinition 3
1.2 Ein Approximationsalgorithmus für CLR 4
1.3 Visualisierung des Algorithmus 6

2 CLRwith Hard Facility Capacities (CLRhFC) 7
2.1 Problemdefinition 7
2.2 Lösungsansätze 9
2.3 Der Algorithmus und seine Implementierung 9
2.4 Analyse der Algorithmen 14
2.5 Ausblick 16

Literatur 18

„Abstract“

Zusammenfassung/Überblick der Arbeit

1 Capacitated Location Routing (CLR)

1.1 Problemdefinition

Eine Instanz des **Capacitated Location Routing Problems (CLR)** ist gegeben durch

- einen ungerichteten, zusammenhängenden Graphen $G = (V,E)$,
- eine Partition der Knoten in Kunden $\mathscr{C}$ und Fabrikstandorte $\mathscr{F}$,
- eine metrischen Kostenfunktion auf den Kanten $c : E \to \mathbb{R}_{\geq 0}$,
- Eröffnungskosten für die Fabriken $\phi : \mathscr{F} \to \mathbb{R}_{\geq 0}$,
- Bedarfe der Kunden $d : \mathscr{C} \to \mathbb{R}_{\geq 0}$
- und eine einheitliche Kapazität $u > 0$ für die Fahrzeuge.

Zulässige Lösungen bestehen aus

- einer Teilmenge $F \subseteq \mathscr{F}$ von eröffneten Fabriken
- und einer Menge von Touren $\mathcal{T} = \{T_1, \ldots, T_k\}$,

sodass gilt:

- Zu jeder Tour gibt es eine geöffnete Fabrik $f \in F$, an der diese startet und endet.
- Alle Touren zusammen erfüllen alle Bedarfe der Kunden.
- Keine der Touren übersteigt die Kapazität u.

Das Optimierungsziel ist es die Gesamtkosten für das Eröffnen der Fabriken und die gefahrenen Touren zu minimieren, also die Minimierung der Kostenfunktion [1]

$$\sum_{T \in \mathcal{T}} c(T) + \sum_{f \in F} \phi(f)$$

Beobachtung 1.1. *CLR* ist NP-schwer, denn es beinhaltet beispielsweise metrisches TSP (betrachte Instanzen mit $|\mathscr{F}| = 1$, $d \equiv 1$ und $u = |\mathscr{C}|$).

[1] Wir verwenden hier, dass eine Funktion $\mathbb{R}$-wertige Funktion $c : M \to \mathbb{R}$ auf einer Menge M eine Funktion $\tilde{c} : \mathcal{P}(M) \to \mathbb{R} : M \supseteq N \mapsto \sum_{x \in N} c(x)$ auf der Potenzmenge $\mathcal{P}(M)$ induziert. Zur Vereinfachung der Notation bezeichnen wir diese Funktion dann ebenfalls mit c. Unter nochmaliger Anwendung dieser Konvention ließe sich die obige Kostenfunktion daher auch als $c(\mathcal{T}) + \phi(F)$ schreiben.

Bemerkung 1.2. Gilt $d \equiv 1$ und $u = 1$, so erhält man eine Instanz des (metrischen) *Uncapacitated Facility Location Problems* (ULF). Statt Touren von den geöffneten Fabriken zu den Kunden zu finden, genügt es hier offensichtlich eine Zuordnung von Kunden zu Fabriken zu bestimmen. Für dieses Problem sind eine ganze Reihe von Approximationsalgorithmen bekannt (vergleiche z.B. [Shm00]), unter anderem erweist sich ein einfacher Greedy-Ansatz bereits als 1,861-approximativ (siehe [Jai+02]).

1.2 Ein Approximationsalgorithmus für CLR

Der in [HKM13] beschriebene 4,38-approximative[2] Algorithmus für CLRbasiert im Wesentlichen auf den folgenden Schritten (schematisch dargestellt in Abb. 1):

ULF-Phase: Erstelle eine ULF-Instanz mit den gleichen Fabriken und Kunden, aber mit um $2/u$ skalierten Kosten. Löse diese Instanz approximativ und eröffne alle hier geöffneten Fabriken auch in der CLR-Instanz. Es zeigt sich, dass die Kosten einer optimalen Lösung der ULF-Instanz eine untere Schranke für die optimale Lösung der CLR-Instanz bildet.

MST-Phase: Ergänze den Graphen der CLR-Instanz um einen zusätzlichen Knoten r mit kostenlosen Kanten zu allen Fabriken. Erhöhe ferne alle Kanten zwischen Fabriken und Kunden um die halben Eröffnungskosten der jeweiligen Fabrik. Dann finde einen minimalen Spannbaum B in diesem Graphen und öffne in der CLR-Instanz zusätzlich alle Fabriken, die in B wenigstens einen Kunden als Nachbarn haben. Die Kosten einen solchen Spannbaumes sind erneut eine untere Schranke für die Kosten einer optimalen Lösung der CLR-Instanz.

Large Demand-Phase: Alle Kunden mit Bedarf von mindestens u werden durch $\lceil d/u \rceil$ Touren mit der jeweils nächsten Fabrik verbunden. Die hierfür anfallenden Anbindungskosten sind höchstens zweimal die entsprechenden Kosten aus der ULF-Phase.

Merge-Phase Betrachte B aus der MST-Phase als Baum mit Wurzel r. Solange dies möglich ist, finde darin einen Teilbaum (mit Wurzel v), der einen Gesamtbedarf von mehr als u hat, aber dessen nächstkleinere Teilbäume (beginnend bei den Kindern von v) einen Bedarf von höchstens u haben. Ein solcher Baum wird als *Relieve-Tree* bezeichnet.

Fasse diese kleineren Teilbäume und den „Baum“ $\{v\}$ so zusammen, dass jede Menge einen Bedarf zwischen $\frac{u}{2}$ und u hat (die letzte ggfls. weniger). Wandle jede der großen Mengen in eine Tour um (durch Verdoppeln der Kanten im entsprechenden Teilbaum und nachfolgendem Abkürzen bei doppelt besuchten Kunden) und verbinde sie mit der nächstliegenden offenen Fabrik. Die Verbindungskosten hierfür sind beschränkt durch das Doppelte der Kosten im Spannbaum B und

[2] Die diesem Programmierprojekt zugrunde liegende Implementierung weist allerdings nur eine Approximationsgarantie von 5,722 auf, da zur approximativen Lösung der ULF-Instanz ein leichter zu implementierender Greedy-Algorithmus verwendet wurde, anstatt eines anderen Verfahrens mit besserer Approximationsgarantie.

den entsprechenden Verbindungskosten aus der ULF-Phase. Die Kunden aus der kleinen Menge bleiben vorerst unversorgt und werden beim nächsten Relieve-Tree neu berücksichtigt.

Ist der verbleibende Gesamtbedarf unter einer Fabrik schließlich kleiner oder gleich u, so werden alle im entsprechenden Teilbaum verbliebenen Kunden zu einer Tour (*Remaining-Tour*) zusammengefasst und mit der nächstliegenden offenen Fabrik verbunden. Die Verbindungskosten hierfür sind erneut durch das Doppelte der entsprechenden Kosten aus dem Spannbaum begrenzt.

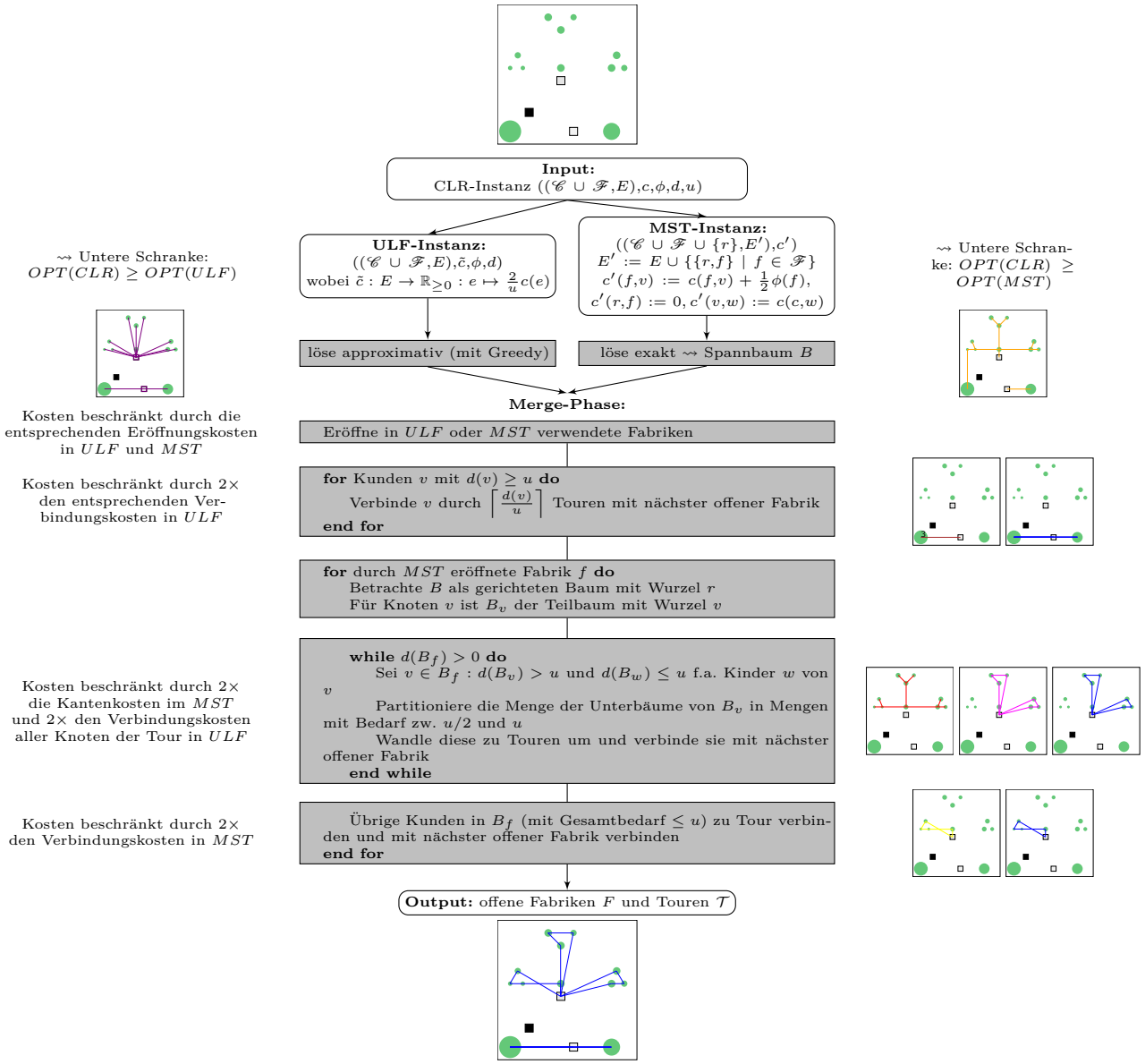


Abbildung 1: Schematische Darstellung des Algorithmus für CLR

1.3 Visualisierung des Algorithmus

Im ersten Teil des Programmierprojektes ging es darum den Ablauf sowie das Ergebnis des oben beschriebenen Algorithmus zu visualisieren. Dazu wurde die existierende Implementierung des Algorithmus um eine Klasse `CLR\_Drawing` erweitert. Diese wird zu Beginn des Algorithmus mit der zu lösenden Instanz initialisiert und kann dann an

verschiedenen Stellen des Algorithmus aufgerufen werden, um einen Schnappschuss mit dem momentanen Stand zu erstellen.

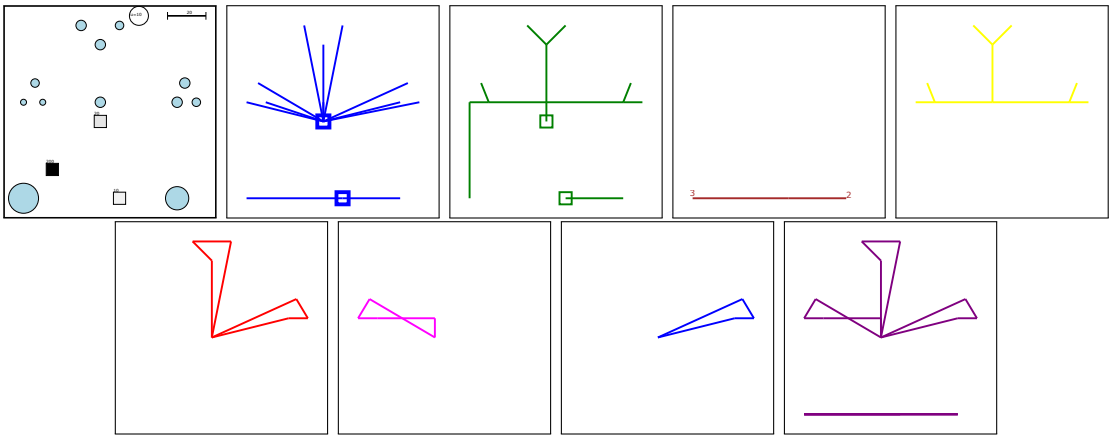


Abbildung 2: Ausgabe der Klasse `CLR_Drawing`: Die Instanz selbst, die Lösung der ULF-Instanz, die Lösung der MST-Instanz, Touren aus der Large-Demand-Phase, ein Relieve-Tree, eine daraus entstandene Tour, eine Remaining-Tour, eine einzelne Tour, alle Touren

Die Ausgabe besteht aus SVG-Dateien, die mit Hilfe der C++-Bibliothek simple-svg ([Tur10]) erstellt werden. Diese können dann beispielsweise von Hand übereinander gelegt werden, um ein bestimmtes Zwischenstadium des Algorithmus darzustellen, oder zu einer Animation zusammengesetzt werden, um den gesamten Ablauf des Algorithmus abzubilden.

Ein Beispiel für ersteres sind die Bilder in dieser Arbeit, als Beispiel für Letztes ...

Klasse detaillierter beschreiben?

Verweis auf eigenständiges Programm zum Einlesen von Touren

Verweis auf Webseite/Anhang/...?

2 CLRwith Hard Facility Capacities (CLRhFC)

Eine Verallgemeinerung von CLRerhält man, indem man die Kapazitäten der Fabriken beschränkt. In diesem Kapitel geht es darum, wie der Approximationsalgorithmus für CLRso angepasst werden kann, dass er auch für das neue Problem zulässige Lösungen findet.

2.1 Problemdefinition

Eine Instanz von **Capacitated Location Routing with Hard Facility Capacities (CLRhFC)** ist gegeben durch:

- eine Instanz $(G = (\mathscr{C} \cup \mathscr{F}, E), c, \phi, d, u)$ von CLR
- und zusätzlich Kapazitäten der Fabriken $l : \mathscr{F} \to \mathbb{R}_{\geq 0}$.

Zulässige Lösungen sind Lösungen der zugrunde liegenden CLR-Instanz, die zudem die Kapazitätsschranken der Fabriken einhalten.

Das Optimierungsziel ist weiterhin die Minimierung der unveränderten Kostenfunktion der CLR-Instanz.

Bemerkung 2.1. Im Gegensatz zu den hier verwendeten *harten Fabrikkapazitäten* gibt es auch Varianten von CLRmit *weichen Fabrikkapazitäten*. Dies bedeutet, dass die Fabriken zwar beschränkte Kapazität haben, jedoch mehrfach eröffnet werden dürfen, wodurch die Kapazität (aber natürlich auch die Eröffnungskosten) entsprechend vervielfacht werden (siehe z.B. [CC09]).

CLRhFCkann auch als Mixed Integer Program (MIP) beschrieben werden. Dabei gibt es

- für jede Fabrik eine Variable o_f, welche bestimmt, ob die entsprechende Fabrik geöffnet (1) ist oder nicht (0),
- sowie für jede mögliche Tour T eine Variabel y_T, welche bestimmt, wie oft die entsprechende Tour genutzt wird,
- und für jeden auf T liegenden Kunden v eine Variable x_{vT}, welche besagt, wie viele Einheiten durch Tour T insgesamt an den Kunden v geliefert werden.

Das MIP sieht dann wie folgt aus:

minimiere

$$\sum_{f \in \mathscr{F}} o_f \phi(f) + \sum_{T \in \mathscr{T}} y_T c(T)$$

unter den Nebenbedingungen:

$$(1) \quad \sum_{v \in T \setminus \mathscr{F}} x_{vT} \leq u y_T, \quad T \in \mathscr{T}$$

$$(2) \sum_{T \in \mathscr{T}_f} \sum_{v \in T \setminus \mathscr{F}} x_{vT} \leq o_f l(f), \quad f \in \mathscr{F}$$

$$(3) \quad \sum_{T \in \mathscr{T}, v \in T} x_{vT} \geq d(v), \quad v \in \mathscr{C}$$

wobei

$$o_f \in \{0,1\}, \quad y_T \in \mathbb{N}_0, \quad x_{vT} \geq 0$$

Die (zu minimierenden) Gesamtkosten ergeben sich als Summe der Eröffnungskosten $\phi(f)$ und der Tourkosten $c(T)$.

(1) Die Fahrzeugkapazität u wird eingehalten. D.h. wird eine Tour y_T-mal genutzt, so können durch sie höchstens $y_T u$ Einheiten an die auf ihr liegenden Kunden geliefert werden.

(2) Die Fabrikkapazitäten l werden eingehalten. Alle bei eine Fabrik f beginnenden Touren $(\mathscr{T}_f)$ können zusammen höchstens so viele Einheiten ausliefern, wie die Fabrikkapazität $l(f)$ zulässt.

(3) Die Bedarfe der Kunden d werden erfüllt. Dies ist der Fall, wenn alle Touren, auf denen ein Kunde v liegt, zusammen mindestens so viel an ihn liefern, wie sein Bedarf $d(v)$ ist.

Beobachtung 2.2. Die Menge aller denkbaren Touren $\mathscr{T}$ wächst exponentiell in der Zahl der Kunden der CLRhFC-Instanz. Dementsprechend schnell steigt auch die Zahl

der Variablen und der Nebenbedingungen an, sodass das Problem nur für sehr kleine Eingabeinstanzen exakt gelöst werden kann.

2.2 Lösungsansätze

Um überhaupt zulässige Lösungen für CLRhFCzu finden, muss der bestehende Algorithmus an wenigstens zwei Stellen angepasst werden: Beim Erstellen der Touren muss nun zusätzlich darauf geachtet werden, dass die Kapazität der ausgewählten Fabrik nicht überschritten wird. Und allgemein muss immer gewährleistet sein, dass überhaupt offene Fabriken mit freier Kapazität verfügbar sind.

Ersteres lässt sich sicherstellen, indem Touren gegebenenfalls noch weiter in Teiltouren aufgespalten werden, die jeweils klein genug sind, um von einer einzelnen offenen Fabrik vollständig beliefert zu werden.

Für Letzteres gibt es zwei naheliegende Ansätze: Entweder man passt die ULF- und/oder MST-Phase so an, dass bereits hier die Fabrikkapazitäten berücksichtigt und dementsprechend viele Fabriken geöffnet werden. Oder man erlaubt dem Algorithmus zusätzlich noch in der Large-Demand- bzw. Merge-Phase bei Bedarf weiter Fabriken zu eröffnen.

Der erste Ansatz ließe sich beispielsweise dadurch verwirklichen, dass statt einer ULF-Instanz eine Instanz des *nonuniform Capacitated Facility Location Problems* erstellt und (approximativ) gelöst wird. Ein entsprechender auf lokaler Suche basierender Approximationsalgorithmus wird in [PTW01] beschrieben. Alternativ (oder auch zusätzlich) könnte man versuchen beim Erstellen des Spannbaumes die Fabrikkapazitäten zu berücksichtigen. Allerdings ist nicht klar, wie eine entsprechende Anpassung dieser Phase aussehen könnte. Allgemein hätte dieser Ansatz möglicherweise den Vorteil .

Add Vorteil

Der zweite Ansatz - neue Fabriken bei Bedarf eröffnen - ist dagegen deutlich leichter zu implementieren und ist daher auch der, der in diesem Programmierprojekt weiterverfolgt wurde. Er hat allerdings den Nachteil, dass dadurch der Zusammenhang zwischen den Kosten der in ULF- und MST-Phase gefundenen Lösungen und denen der letztendlich bestimmten Lösung der CLRhFC-Instanz verloren gehen.

siehe Abschnitt 2.4

2.3 Der Algorithmus und seine Implementierung

Die Implementierung des angepassten Algorithmus erfolgt im Wesentlichen durch eine von der originalen `Solver`-Klasse abgeleiteten Klasse `SolverCap` (bzw. eine wiederum davon abgeleitete Klasse `SolverCapIt`). Dadurch kann die `main`-Funktion weitgehend gleich bleiben und es muss lediglich zu Beginn entschieden werden, welche Variante der Solver-Klasse (und damit welche Variante des Algorithmus) verwendet werden soll. Innerhalb der abgeleiteten `Solver`-Klassen werden dann die anzupassenden Funktionen überschrieben, während die restlichen Funktionen unverändert bleiben.

```
Solver* s;
```

```
if(inst.is_depot_cap_relevant()) {
    if(itCapAlgorithm)  s = new SolverCapIt(inst, tspPostOpt);
    else                s = new SolverCap(inst, tspPostOpt);
} else                  s = new Solver(inst, tspPostOpt);
...
Solution sol = s->solve();
```

Listing 1: Der weitere Ablauf des Programms kann unabhängig von der verwendeten Version des Solvers beschrieben werden

Zusätzlich zur neuen `Solver`-Klasse musste außerdem in der Klasse `Solution` eine neue Funktion `isFeasable2()` zur Zulässigkeitsprüfung erstellt werden. Denn um die Zulässigkeit von Lösungen der CLR-Instanz schnell prüfen zu können, wurden einige Eigenschaften dieser Lösungen genutzt, die bei Lösungen einer CLRhFCnicht mehr garantiert werden können - nämlich die *single assignment*-Eigenschaft (jeder Kunde wird von genau einer Fabrik beliefert) und die *single tour*-Eigenschaft für Kunden mit kleinem Bedarf. Außerdem sollte bei CLRhFC-Lösungen noch geprüft werden, ob auch die Fabrikkapazitäten eingehalten werden.

Hier bereits auf weitere Ergänzungen hinweisen? generateCap und writeMIP

In der folgenden Beschreibung bezeichnen $\tilde{d} : \mathscr{C} \to \mathbb{R}_{\geq 0}$ und $\tilde{l} : \mathscr{F} \to \mathbb{R}_{\geq 0}$ den zum jeweiligen Zeitpunkt verbleibenden Bedarf der Kunden bzw. die verbleibende Kapazität der Fabriken.

2.3.1 Toursplitting

Wann immer im ursprünglichen Algorithmus während der Merge-Phase eine neue Tour erstellt wird (aus einem Relieve-Tree in der Funktion `relieveSubtree` oder als Remaining-Tour in der Funktion `mergePhase`), so wird jetzt zusätzlich noch eine neue Funktion `splitTour` aufgerufen. Diese zerteilt die übergebene Tour in kleinere Teiltouren (entsprechend den verfügbaren Fabrikkapazitäten) und verbindet sie mit passenden Fabriken. Im Falle der Relieve-Trees wird dafür die eigene Partitionierung in Touren mit Bedarf zwischen $u/2$ und u weggelassen und dafür die Fahrzeugkapazitäten in der Zerteilung in Teiltouren mitberücksichtigt.

Sei dazu T die zu zerlegende Tour (oBdA bestehe diese nur aus Kunden mit verbleibendem Bedarf > 0). Unter den offenen Fabriken mit noch verfügbarer Kapazität wählen wir nun die „am besten“ zu den Kunden in T passende Fabrik f aus. Diese Fabrik wird dann mit dem ihr am nächsten liegenden Kunden aus T verbunden und von diesem aus T soweit durchlaufen wie die Kapazität von f (und die Fahrzeugkapazität) reicht. Die dadurch vollständig versorgten Kunden werden nun aus T entfernt, beim letzten Kunden der Tour wird der Bedarf entsprechend der gelieferten Menge reduziert und die Kapazität von f wird verringert. Dann suchen wir erneut eine „beste Fabrik“ für die in T verbliebenen Kunden (sofern T noch nicht leer ist).

Zu klären bleibt damit noch, wie die „beste Fabrik“ zu einer gegebenen Menge M an Kunden bestimmt wird. Dies geschieht in der Funktion `findBestFacility`. Dazu wird zunächst der momentane Gesamtbedarf der übergebenen Menge $\tilde{d}(M)$ bestimmt und dann jeder offenen Fabrik f wie folgt ein Wert zugeordnet:

$$\text{Wert}(f) := \frac{\min\{\tilde{l}(f), \tilde{d}(M)\}}{2c(f,M)}$$

Dabei bezeichnet $c(f,M)$ den Abstand von f zu dem nächstliegenden Kunden aus M. Der Wert einer Fabrik ist also umso höher je mehr nutzbare Kapazität sie hat und je näher an den zu versorgenden Kunden sie liegt (bzw. zumindest einem von diesen). Die Fabrik mit dem höchsten Wert wird schlussendlich ausgewählt.

Hat keine der offenen Fabriken mehr übrige Kapazität, so entscheidet die Funktion `findBestFacility` darüber, welche Fabrik nun zusätzlich eröffnet werden soll. Für diese Entscheidung wurden zwei verschiedene Ansätze implementiert, die in den folgenden beiden Abschnitten beschrieben werden.

2.3.2 Greedy Fabrikeröffnung

In der Klasse `SolverCap` wird bei der Auswahl der neu zu eröffnenden Fabrik eine Art Greedy-Ansatz verfolgt: Es wird jeder geschlossenen Fabrik f ein Wert

$$\text{Wert}(f) := \frac{\min\{\tilde{l}(f), \tilde{d}(M)\}}{2c(f,M) + \phi(f)}$$

zu gewiesen und dann die Fabrik mit dem höchsten Wert zusätzlich eröffnet.

```
class SolverCap: public Solver {
public:
    SolverCap(const Instance& in, bool tsp) : Solver(in, false) {
        demandsRemaining = vector<double> (n);
        for(int i=0; i<n; i++)
            demandsRemaining.at(i) = inst.demand(i);
        facCapacitiesRemaining = vector<double> (m);
        for(int i=0; i<m; i++)
            facCapacitiesRemaining.at(i) = inst.facCapacity(i+n);
    };

protected:
    vector<double> demandsRemaining;
    vector<double> facCapacitiesRemaining;

    virtual bool mergePhase(const Tree& tree);
    virtual double relieveSubtree(const Tree& t, int root,
        const vector<int>& children ,
        vector<pair<int, double> >& stDemands);
```

```
    virtual bool largeDemandPhase();
    virtual double handleSubtree(const Tree& t, int root);

    list<list<int> > splitTour(list<int> tour);
    int closestOpenFacility(int client);
    virtual int findBestFacility(list<int> subset);
    int findClosestClient(int fac, list<int> subset);

    double demandRemaining(int client) {
        return demandsRemaining.at(client);};
    bool reduceRemainingDemand(int client, double reduction);

    double facCapacityRemaining(int facility) {
        return facCapacitiesRemaining.at(facility-n);};
    bool reduceRemainingFacCapacity (int facility, double reduction);
};
```

Listing 2: Die Klasse SolverCap

2.3.3 Fabrikeröffnung durch wiederholte ULF/MST-Phasen

In der Klasse `SolverCapIt` werden die neu zu eröffnenden Fabriken dagegen mit Hilfe einer neuen ULF- und MST-Phase bestimmt. Dazu wird eine temporäre CLR-Instanz erstellt, welche nur die Kunden mit noch zu erfüllendem Bedarf und die (derzeit geschlossenen) Fabriken mit übriger Kapazität enthält. Auf dieser Instanz wird dann die übliche ULF- und MST-Phase durchgeführt und alle dabei geöffneten Fabriken werden nun auch in der ursprünglichen CLRhFC-Instanz zusätzlich geöffnet.

Nachdem dies geschehen ist, werden alle bisher gefundenen Touren gelöscht und die Tourenfindungsphase (Large Demand und Merge) neu gestartet. Dies wird so oft wiederholt bis ausreichend Fabriken geöffnet sind, um alle Kunden vollständig zu versorgen.

Implementiert ist dieses Vorgehen in der Klasse `SolverCapIt`, welche von der Klasse `SolverCap` abgeleitet ist und ausschließlich deren Funktion zum Finden der „besten“ Fabrik `findBestFacility` sowie die Funktion `solve` überschreibt. Zur Durchführung der zusätzlichen ULF- und MST-Phasen auf den temporären Instanzen gibt es zudem eine von der Klasse `Solver` abgeleitete Klasse `SolverRed`, die im Vergleich zu dieser die Möglichkeit bietet ULF- und MST-Phase eigenständig durchzuführen und anschließend die Menge der geöffneten Fabriken auszulesen.

```
class SolverCapIt: public SolverCap {
public:
    SolverCapIt(const Instance& in, bool tsp) :
        SolverCap(in, false) {};
    virtual Solution solve();

private:
```

```
    virtual int findBestFacility (list<int> subset);
};
```

Listing 3: Die Klasse SolverCapIt

```
class SolverRed: Solver {
public:
    SolverRed(const Instance& in, bool tsp) :
        Solver(in, false) {create_drawings = false;};
    vector<bool> getOpenFacs()  {return facOpen;};
    double uflPhase()       {return Solver::uflPhase();};
    const Tree treePhase()  {return Solver::treePhase();};
};
```

Listing 4: Die Klasse SolverRed

2.3.4 Zulässigkeitsprüfung

Vor dem Ausgeben der Lösung bietet die Klasse `Solution` die Möglichkeit die gefundene Lösung auf Zulässigkeit zu prüfen, d.h. ob durch die gefundenen Touren die Bedarfe aller Kunden erfüllt werden können ohne die Fahrzeug- oder Fabrikkapazitäten zu überschreiten. Hierzu wird ein maximaler Fluss auf folgendem gerichteten Graphen H (siehe auch Abb. 3) bestimmt:

- H besitzt je einen Knoten für jede Fabrik, jede Tour und jeden Kunden, sowie zusätzlich eine Quelle und eine Senke.
- Von der Quelle gibt es jeweils eine Kante zu jeder offenen Fabrik mit der Kapazität der Fabrik als Kantenkapazität.
- Jede Tour hat eine eingehende Kante mit Kapazität u von ihrer Startfabrik und ausgehende (unbeschränkte) Kanten zu jedem Kunden, der auf dieser Tour liegt.
- Von jedem Kunden aus führt eine Kante zur Senke mit dem Bedarf des Kunden als Kantenkapazität.

Lemma 2.3. *Der Graph H besitzt genau dann einen (maximalen) Fluss mit Flusswert gleich der Summe aller Bedarfe, wenn die gefundene Lösung zulässig ist.*

Beweis. Ein Fluss in diesem Graphen entspricht gerade einer Zuteilung von Produktion der Fabriken über die bestehenden Touren zu den Kunden. Die Einhaltung der Kantenkapazitäten zwischen Quelle und Fabriken bzw. Fabriken und Touren stellt die Einhaltung der Fabrik- bzw. Fahrzeugkapazitäten sicher und umgekehrt. Und schließlich schließlich sorgen die Kapazitäten der Kanten von den Kunden zur Senke dafür, dass kein Kunde mehr bekommt als seinem Bedarf ist. Somit entspricht ein Flusswert gleich der Summe aller Bedarfe gerade einer Verteilung bei der alle Bedarfe genau erfüllt werden. □

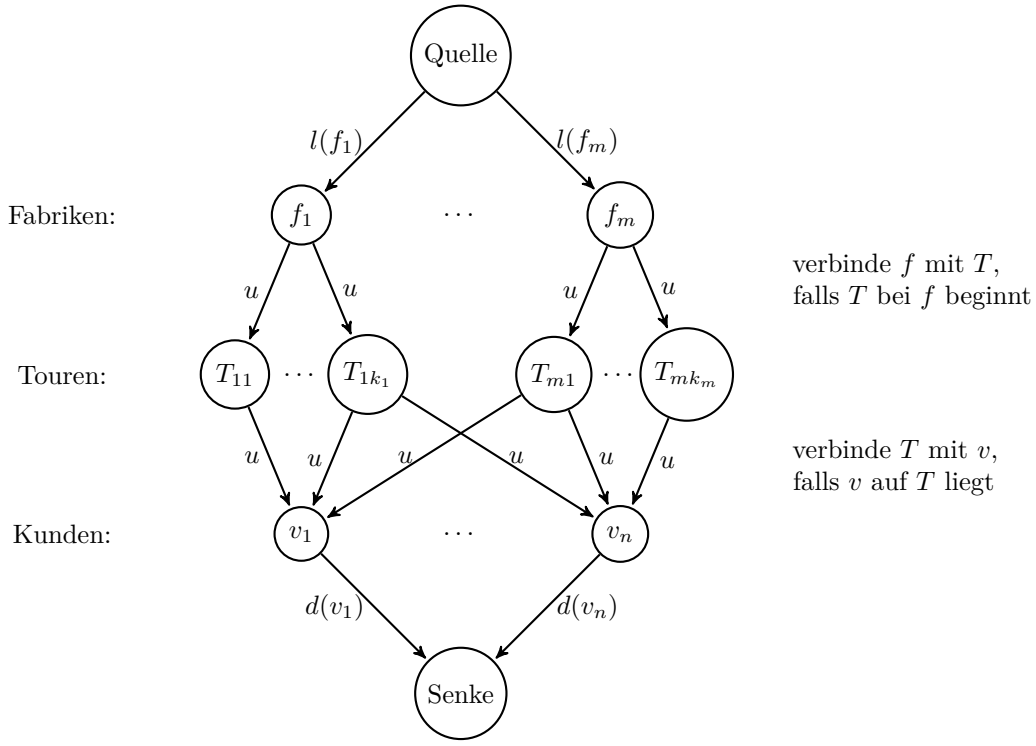


Abbildung 3: Die Max-Fluss-Instanz zur Überprüfung der Zulässigkeit einer gefundenen Lösung

Um also die Zulässigkeit einer gefundenen Lösung zu überprüfen erstellt die Funktion `isFeasable2` zunächst wie oben beschrieben eine Instanz für ein Max-Fluss-Problem. Dann wird mit Hilfe einer leicht vereinfachten Implementierung des Edmonds-Karp-Algorithmus (in der in [Dis15] beschriebenen Form) ein maximaler Fluss in diesem Graphen bestimmt und schließlich dessen Flusswert mit der Summe aller Bedarfe verglichen.

Dazu gibt es neben der Funktion `EdmondsKarp` selbst eine Hilfsfunktion `shortestSTPath`, welche mit Hilfe einer Breitensuche einen kürzesten augmentierenden Pfad von der Quelle zur Senke bestimmt.

2.4 Analyse der Algorithmen

2.4.1 Theoretische Betrachtungen

In [HKM13] werden zwei untere Schranken für die Kosten der optimalen CLR-Lösung gezeigt: Die Kosten der optimalen Lösungen der im Algorithmus verwendeten ULF-Instanz sowie die der MST-Instanz. Da jede Lösung einer CLRhFC-Instanz insbesondere auch

eine zulässige Lösung der zugrunde liegenden CLR-Instanz ist, gelten diese Schranken weiterhin.

Während jedoch im Fall der CLR-Problem aus den beiden Werten zusammen auch eine obere Schranke für die Kosten der optimalen CLR-Lösung bestimmt werden kann (auf diese Weise wird ja gerade die Approximationsgüte des Algorithmus bewiesen), ist dies für das CLRhFC-Problem nicht mehr der Fall. Insbesondere können diese beiden unteren Schranken hier also beliebige weit von den Kosten der optimalen CLRhFC-Lösung entfernt liegen.

Untere Schranken

Schlechte Beispiele

2.4.2 Heuristische Beurteilung

Um trotz der fehlenden bewiesen Approximationsgüte eine gewisse Einschätzung für die Qualität der gefundenen Lösungen zu bekommen sowie die beiden Varianten des Algorithmus miteinander zu vergleichen, habe ich die beiden Algorithmen auf einigen zufällig generierten Instanzen getestet.

Zu diesem Zweck habe ich das Programm um eine Funktion `writeCLRhFCMIP` ergänzt, welche zu einer Instanz von CLRhFCdas passende MIP im LP-Dateiformat von CPLEX (siehe [Lpf]) erzeugt. Das so erzeugte MIP kann dann beispielsweise mit dem Solver SCIP ([Gam+16]) gelöst werden bzw. zumindest untere und obere Schranken für die optimale Lösung bestimmt werden.

Um das MIP zumindest ein wenig kleiner zu halten, wird dabei nicht wie in Abschnitt 2.1 beschrieben zu *jeder* möglichen Tour eine eigene Variable eingeführt, sondern zu jeder Teilmenge der Kunden und jeder Fabrik nur eine optimale Tour berücksichtigt. Dadurch reduziert sich die Zahl der Variablen in etwa um einen Faktor $n!$. Zwar muss dafür für jede solche Teilmenge ein TSP gelöst werden (dies geschieht hier in der Funktion `solveTSP` mit Hilfe einer sehr einfachen Branch-and-Bound-Heuristik), da das MIP jedoch sowieso nur für sehr kleine Instanzgrößen gelöst werden kann (bereits bei 10 Kunden und 5 Fabriken besteht das MIP aus ~ 31.000 Variablen und ~ 36.000 Bedingungen), stellt dies hier kein Hindernis dar.

Ergebnisse beschreiben und einfügen...

#	ULF	MST	Greedy				Iterativ				optimal
0	341,22	123,02	3	7	657,00	1,34	3	7	602,08	1,23	491,19
1	350,67	126,97	4	8	1146,82	1,84	4	6	745,58	1,19	624,43
2	379,63	123,00	2	6	505,00	1,00	2	6	505,00	1,00	505,00
3	407,61	113,65	3	10	832,94	1,35	3	9	701,21	1,14	617,45
4	396,20	129,58	3	5	1077,55	1,41	3	5	1015,18	1,33	761,80
5	482,74	111,03	4	8	1050,78	1,31	4	8	1064,82	1,33	802,49
6	421,26	198,18	3	8	913,68	1,57	3	6	588,60	1,01	581,07
7	573,51	147,91	2	8	922,11	1,29	2	8	716,82	1,01	712,55
8	415,82	115,07	3	8	562,61	1,08	3	8	562,61	1,08	522,19
9	206,94	113,13	3	6	772,37	1,57	3	6	698,41	1,42	493,11

#	ULF	MST	Greedy			Iterativ			It/Greedy
10	687,96	157,28	3	15	1096,58	3	7	602,08	0,55
11	687,92	190,59	3	14	1248,96	4	6	745,58	0,60
12	480,02	180,59	2	13	806,66	2	6	505,00	0,63
13	1043,82	151,78	4	21	2311,70	3	9	701,21	0,30
14	500,34	181,41	2	9	1043,67	3	5	1015,18	0,97
15	806,04	223,23	3	16	1502,07	4	8	1064,82	0,71
16	504,80	140,26	3	10	1184,91	3	6	588,60	0,50
17	966,57	191,92	3	13	1847,71	2	8	716,82	0,39
18	584,23	228,74	3	11	1374,03	3	8	562,61	0,41
19	520,96	197,01	3	13	1252,55	3	6	698,41	0,56

2.5 Ausblick

Was könnte man verbessern? Welche Probleme gibt es dabei?

Liste der noch zu erledigenden Punkte

Zusammenfassung/Überblick der Arbeit 3
Verweis auf Webseite/Anhang/...? 7
Klasse detaillierter beschreiben? 7
Verweis auf eigenständiges Programm zum Einlesen von Touren 7
Add Vorteil 9
siehe Abschnitt 2.4 9
Hier bereits auf weitere Ergänzungen hinweisen? generateCap und writeMIP . . . 10
Untere Schranken 15
Schlechte Beispiele 15
Ergebnisse beschreiben und einfügen... 15
Was könnte man verbessern? Welche Probleme gibt es dabei? 16

Literatur

[CC09] Xujin Chen und Bo Chen. „Approximation Algorithms for Soft-Capacitated Facility Location in Capacitated Network Design“. In: *Algorithmica* 53.3 (2009), S. 263–297. ISSN: 1432-0541. DOI: 10.1007/s00453-007-9032-7. URL: http://dx.doi.org/10.1007/s00453-007-9032-7.

[Dis15] Yann Disser. „Vorlesungsskript zu Optimierung III“. 2015.

[Gam+16] Gerald Gamrath u. a. *The SCIP Optimization Suite 3.2*. eng. Techn. Ber. 15-60. Takustr.7, 14195 Berlin: ZIB, 2016.

[HKM13] Tobias Harks, Felix G. König und Jannik Matuschke. „Approximation Algorithms for Capacitated Location Routing“. In: *Transportation Science* 47.1 (2013), S. 3–22. DOI: http://dx.doi.org/10.1287/trsc.1120.0423. URL: http://researchers-sbe.unimaas.nl/tobiasharks/wp-content/uploads/sites/29/2014/02/HKM-TS-2013.pdf.

[Jai+02] Kamal Jain u. a. „Greedy Facility Location Algorithms Analyzed using Dual Fitting with Factor-Revealing LP“. In: *CoRR* cs.DS/0207028 (2002). URL: http://arxiv.org/abs/cs.DS/0207028.

[Lpf] *LP file format: algebraic representation*. IBM Knowledge Center. URL: https://www.ibm.com/support/knowledgecenter/SS9UKU_12.5.0/com.ibm.cplex.zos.help/FileFormats/topics/LP.html (besucht am 03. 12. 2016).

[PTW01] Martin Pal, Eva Tardos und Tom Wexler. „Facility Location with Nonuniform Hard Capacities“. In: *Proceedings of the 42nd IEEE Symposium on the Foundations of Computer Science*. 2001, S. 329–338. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.16.1003.

[Shm00] David B. Shmoys. „Approximation Algorithms for Facility Location Problems“. In: *Approximation Algorithms for Combinatorial Optimization: Third International Workshop, APPROX 2000 Saarbrücken, Germany, September 5–8, 2000 Proceedings*. Hrsg. von Klaus Jansen und Samir Khuller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, S. 27–32. ISBN: 978-3-540-44436-7. DOI: 10.1007/3-540-44436-X_4. URL: http://dx.doi.org/10.1007/3-540-44436-X_4.

[Tur10] Mark Turney. *simple-svg*. Google Code Archive. simple-svg ist eine header-only C++ Library, mit deren Hilfe einfache svg-Graphicen erstellt werden können. 2010. URL: https://code.google.com/archive/p/simple-svg/.