Online-Synthese von Aufzugssteuerungen als Planungsproblem

Bernhard Seckinger **Jana Koehler**
Institut für Informatik
Albert Ludwigs Universität
Am Flughafen 17
79110 Freiburg
seckinge|koehler@informatik.uni-freiburg.de

Zusammenfassung

Bei MICONIC-10, einem neuartigen Aufzugssystem der Firma Schindler Aufzüge AG, geben die Passagiere schon vor Fahrtantritt das Zielstockwerk an. Dies ermöglicht es, Passagiere einzeln zu erfassen und somit auf ihre speziellen Bedürfnisse einzugehen. Gleichzeitig soll möglichst der Aufzugsbetrieb optimiert werden.

Eine komplexitätstheoretische Analyse des Problems zeigt jedoch sofort, daß das Minimierungsproblem bezüglich der Lösungslänge NP-vollständig ist. Eine direkte Berechnung eines minimalen Fahrtwegs des Aufzugs ist somit effizient nicht möglich. Da das Problem Planungscharakter hat – es soll der Fahrtweg des Aufzugs geplant werden – bietet sich eine entsprechende Modellierung an. Nachfolgend diskutieren wir zwei alternative Realisierungen des Planungsverfahrens. Die erste Realisierung basiert auf einem Constraint-Satisfaction Algorithmus, die zweite auf einer heuristisch angesteuerten Vorwärtssuche.

Einleitung

Bei MICONIC-10, einem neuartigen Aufzugssystem der Firma Schindler Aufzüge AG, geben die Passagiere schon vor Fahrtantritt das Zielstockwerk an. Die Eingabe wird bestätigt und der Passagier direkt einem der möglichen Aufzüge zugewiesen. Im Innern des Aufzugs müssen keine Knöpfe mehr gedrückt werden. Dieses innovative System ermöglicht es, die Passagiere einzeln zu erfassen und somit auf ihre speziellen Bedürfnisse einzugehen. Die aktuelle Steuerungssoftware faßt Passagiere mit gleicher Ein- und Ausstiegsposition zusammen und berechnet eine heuristisch gute Lösung. Dadurch kann eine Überfüllung von Aufzügen sowie das unnötige Halten vermieden werden. Das System ist seit mehreren Jahren erfolgreich auf dem Markt.

Der kommerzielle Erfolg des Produkts wirft die Frage nach einer weiteren Verbesserung der Steuerungssoftware auf. Zum Beispiel werden spezielle Dienste für behinderte Passagiere angeboten und der Zutritt auf bestimmte Stockwerke kann beschränkt werden. Aktuell werden diese Erweiterungen direkt im Code implementiert, was nicht nur aufwendig ist, sondern auch die Gefahr birgt, daß mittelfristig die Qualität der Software beeinträchtigt wird.

Die Steuerung basiert auf einem Angebotsmodell. Jeder Passagier löst bei der zentralen Prozeßsteuerung eine Anfrage bestehend aus Ein- und Ausstiegsposition aus. Die zentrale Steuerung leitet diese Anfrage an jeden Aufzug weiter und erfragt ein Angebot. Der Passagier wird dann auf den Aufzug mit dem besten Angebot gebucht. Die Aufgabe des Planungssystems ist es, die Angebotserstellung für einen Aufzug zu realisieren.

Definition 1 *Eine* Anfrage *ist ein Tupel* (e, a)*, wobei* e *die Ein- und* a *die Ausstiegsposition ist.*

Ein Auftrag *ist ein Tupel* (A, p) *bestehend aus einer Menge* A *von* n *Anfragen und der aktuellen Position des Aufzugs* p*.*

Wir betrachten dazu folgendes Problem:

Gegeben: Eine Menge von Anfragen $A = \{(e_1, a_1), \ldots, (e_n, a_n)\}$ und die aktuelle Position des Aufzugs p.

Gesucht: Finde eine Positionsfolge, die in p beginnt,

1. die Positionen e_i bzw. a_i enthält, wobei e_i mindestens einmal vor a_i vorkommt,
2. eine Menge zusätzlicher Erweiterungen erfüllt, und
3. optimal bezüglich eines gegebenen Kriteriums ist.

Gefragt ist ein Algorithmus, der

- effizient ist – die Planungszeit darf höchstens eine Sekunde betragen,
- korrekt und vollständig ist und somit die optimale Lösung garantiert, und
- flexibel erweitert werden kann.

Im nächsten Abschnitt werden wir das Problem formal definieren und auf dessen Komplexität eingehen. Wir werden insbesondere zeigen, daß die Minimierung der Lösungslänge ein NP-hartes Problem ist. Nach einer kurzen Diskussion der speziellen Eigenheiten des Problems und einer Abgrenzung zu Scheduling Verfahren untersuchen wir einen Constraintsolver und ein Verfahren mit spezieller Vorwärtssuche. Anhand ausgewählter Beispiele werden beide Verfahren miteinander verglichen und ihre Vor- und Nachteile analysiert.

Zulässige und minimale Lösungen

Zu einer gegebenen Menge an Anfragen läßt sich sehr leicht eine Lösung finden, indem man einfach alle Einzelanfragen nacheinander ausführt. Dabei entstehen jedoch Irrfahrten, d. h. der Aufzug hält an Positionen, an denen weder Personen ein- noch aussteigen wollen. Eine Lösung, die keine solche Irrfahrten enthält, werden wir als *zulässige* Lösung bezeichnen. Formal läßt sich dies wie folgt definieren:

Definition 2 *Eine (triviale)* Lösung *eines Auftrags* (A, p) *ist eine Positionsfolge* $p_1, \ldots, p_l$ *der Länge* l *mit folgenden beiden Eigenschaften:*

1. $p_1 = p$
2. $\forall (e, a) \in A \; \exists i, j$ *mit* $i < j : p_i = e \wedge p_j = a$

Definition 3 *Eine Lösung heißt* zulässig, *wenn ausserdem noch gilt:*

3. *Es gibt keine* p_i, $1 < i \leq l$, *bei denen weder Personen ein- noch aussteigen.*

Weiter benötigen wir noch den Begriff der minimalen Lösung:

Definition 4 *Eine zulässige Lösung heißt* minimal *genau dann, wenn ihre Länge minimal unter allen zulässigen Lösungen ist.*

Beispiel 1 Sei folgender Auftrag gegeben: ({(4,5), (2,3), (1,4), (1,2), (5,1), (3,1)}, 6). Der Aufzug steht im sechsten Stock, ein Passagier will von der vierten in die fünfte Etage, der nächste von 2 nach 3, etc. Die Repräsentation des Auftrags durch einen Graphen zeigt Abb. 1.

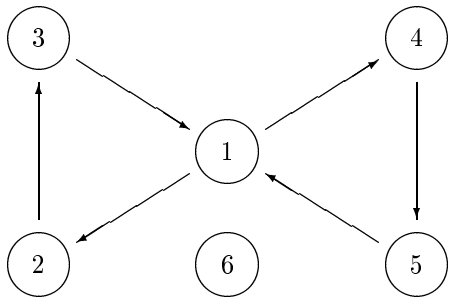

Abbildung 1: Graph-Repräsentation des Auftrags zu Beispiel 1.

Eine triviale Lösung läßt sich nun einfach durch Aneinanderreihen der Anfragen berechnen: $6, 4, 5, 2, 3, 1, 4, 1, 2, 5, 1, 3, 1$.

Wie man aus einer (trivialen) Lösung eine zulässige Lösung berechnen kann, zeigt folgender Satz:

Satz 1 *Zu jeder Lösung eines Auftrags* (A, p) *läßt sich in* $O(l \cdot n)$ *eine zulässige Lösung berechnen, die kürzer oder gleichlang ist.*

Beweis: Folgender Algorithmus leistet das gewünschte: Mittels Scanline-Prinzip durchläuft man die Lösung und verfolgt dabei zu jedem Passagier, ob er wartet, sich in der Kabine befindet oder schon am Ziel ist. Treten dabei an einer Position keine Änderungen auf, so kann diese Position gestrichen werden. □

Beispiel 1 (1. Fortsetzung) Wendet man den Algorithmus aus Satz 1 auf die Lösung aus Beispiel 1 an, so erhält man: $6, 4, 5, 2, 3, 1, 4, 2$. Dies ist eine von 2530 zulässigen Lösungen.

Im folgenden betrachten wir das Optimierungskriterium „minimale Länge“. Mit anderen Worten, wir suchen eine Lösung $p_1, \ldots, p_l$ mit l minimal.

Beispiel 1 (2. Fortsetzung) Daß obige Lösung nicht minimal ist, zeigt die Lösung $6, 1, 2, 3, 4, 5, 1$, welche um eine Position kürzer und tatsächlich minimal ist.

Mit dem nächsten Satz zeigen wir, daß dieses Minimierungsproblem NP-hart ist, indem wir die NP-Vollständigkeit des zugehörigen Entscheidungsproblems beweisen. Zuvor benötigen wir aber noch einige Definitionen:

Definition 5 *Sei* (A, p) *ein Auftrag. Eine Position* x *heißt* relevant, *wenn gilt:* $x = p \vee \exists (a, b) \in A : x = a \vee x = b$.

Definition 6 (AUFZUG)

Gegeben: *Ein Auftrag* (A, p) *und eine natürliche Zahl* $k \leq 2|n| + 1$.

Gefragt: *Gibt es eine zulässige Lösung mit Länge* $\leq k$*?*

Wir benötigen für den Beweis der NP-Vollständigkeit von AUFZUG das NP-vollständige Problem FEEDBACK VERTEX SET (FVS) (Garey & Johnson 1979):

Definition 7 (FEEDBACK VERTEX SET)

Gegeben: *Ein gerichteter Graph* $G = (V, E)$ *und eine natürliche Zahl* $k \leq |V|$.

Gefragt: *Gibt es eine Teilmenge* $V' \subseteq V$ *mit* $|V'| \leq k$ *derart, daß* V' *mindestens einen Knoten aus jedem Zyklus in* G *enthält?*

Satz 2 *Das Problem, ob zu einem Auftrag eine zulässige Lösung mit Länge* $\leq k$ *existiert, ist NP-vollständig.*

Beweis von Satz 2:
Abb. 1 hat bereits die Graph-Repräsentation von Aufträgen exemplarisch eingeführt. Dabei wird jeder relevanten Position ein Knoten und jeder Anfrage eine Kante im Graphen zugeordnet.

Mitgliedschaft in NP: Eine zulässige Lösung wird nichtdeterministisch geraten und kann zum Beispiel mit dem oben skizzierten Scanline-Verfahren in polynomialer Zeit verifiziert werden.

NP-Härte: Beweis durch Reduktion von FVS auf AUFZUG.

$G = (V, E)$ hat ein FVS der Größe $\leq k' = k - |V|$ genau dann, wenn der durch G repräsentierte Auftrag (A, p) eine zulässige Lösung der Länge $\leq k$ hat.

In Abb. 1 bildet die Menge $\{1\}$ ein minimales FVS. Es gibt also eine zulässige Lösung der Länge ≤ 7 wegen $k = k' + |V| = 1 + 6$. Eine solche Lösung haben wir in Beispiel 1, 2. Fortsetzung schon kennengelernt.

$\Longrightarrow$: Sei zu einem Graphen $G = (V, E)$ ein FVS M mit $|M| \leq k'$ gegeben. Man beachte, daß V gerade die Menge der relevanten Positionen darstellt. Wir konstruieren nun eine zulässige Lösung der Länge $\leq k' + |V|$ für den durch G repräsentierten Auftrag. Im ersten Schritt werden alle gerichteten Kanten, die in einem Element des FVS enden, entfernt. Der so entstandene Graph $G' = (V, E')$ ist zyklenfrei, da jeder Zyklus in G nach Definition von M eine Kante (a, b) mit $b \in M$ besitzt. Im zweiten Schritt werden alle Knoten des nun zyklenfreien Graphen topologisch sortiert, d. h. wir definieren eine bijektive Abbildung der Knotennamen auf $\mathbb{N}$ $ord : V \to \{1, \ldots, |V|\}$ derart, daß mit $(v, w) \in E'$ auch $ord(v) < ord(w)$ gilt.

Die zulässige Lösung ergibt sich als $v_1, \ldots, v_{|V|}$, $v_{|V|+1}, \ldots, v_{|V|+|M|}$. Die Elemente $v_1, \ldots, v_{|V|}$ sind gerade die relevanten Positionen, die mindestens einmal angefahren werden. Dies erledigt alle Anfragen $(a, b) \in E'$ und führt dazu, daß alle Passagiere aus Anfragen $(a, b) \in E \setminus E'$ (repräsentiert durch die zuvor entfernten Kanten aus den Zyklen) zusteigen können. Die Elemente $v_{|V|+1}, \ldots, v_{|V|+|M|}$ sind genau die Elemente des FVS M. Diese müssen nun noch einmal angefahren werden, da bei Anfragen $(a, b) \in E \setminus E'$ die Beziehung $ord(b) < ord(a)$ gilt und somit die Bedingung 2 aus Definition 2 noch erfüllt werden muß. Die zulässige Lösung hat damit genau die Länge $|V| + |M| \leq |V| + k' = k$.

$\Longleftarrow$: Sei $G = (V, E)$ der Graph zu einem Auftrag mit Lösung $v_1, \ldots, v_l$ und der Länge $l \leq k$. Wir zeigen nun, daß die Positionen, die in der Lösung mindestens doppelt vorkommen, ein gesuchtes FVS bilden. Dazu muß man nur zeigen, daß aus jedem Zyklus Γ ein Knoten existiert, der in der Lösung mindestens doppelt vorkommt. Dies sieht man wie folgt: Man betrachtet das erste Vorkommen eines Knotens aus Γ. Dieser Knoten v_Γ muß noch einmal vorkommen, da es eine Anfrage (a, b) im Zyklus Γ gibt, mit $b = v_\Gamma$.

Das so konstruierte FVS hat die Länge $l' \leq k' = k - |V|$: Da alle $|V|$ relevanten Positionen mindestens einmal in der Lösung vorkommen, können maximal $k - |V|$ doppelt vorkommen. □

Aus der Sicht der Anwendung sind minimale Lösungen zwar nicht schlecht, allerdings steht hier in erster Linie die schnelle Bedienung der Passagiere im Vordergrund. Gesucht ist also eine Lösung, die die Summe der Bedienzeiten aller Passagiere minimiert. Seien also $T_i^{\odot}$ der Anmeldezeitpunkt eines Passagiers i und $T_i^{\ominus}$ sein Aussteigezeitpunkt, so minimiert die *optimale* Lösung die folgende Summe der Bedienzeiten:

$$T_\Sigma = \sum_{i=1}^{n} T_i^{\ominus} - T_i^{\odot} \tag{1}$$

Die Komplexität des Optimierungsproblems ist zur Zeit noch offen. Insbesondere gilt, daß nicht jede optimale Lösung auch eine minimale Lösung ist.

Einordnung des Problems

Das vorgestellte Problem kann unter verschiedenen Blickwinkeln betrachtet werden. Zunächst bietet sich eine Modellierung als Schedulingproblem an: Jede Anfrage entspricht einem Job, der auf einer Maschine (dem Aufzug) ausgeführt werden soll. Gesucht ist diejenige Parallelisierung des Auftrags, die zur kürzesten Gesamtbearbeitungszeit aller Anfragen führt. Bei genauerer Betrachtung stellt man jedoch fest, daß einige wesentliche Unterschiede der Aufzugssteuerung zu einer besonders schwierigen Instanz eines Schedulingproblems führen. Zunächst wird hier ein *multi-capacity* Problem definiert, da der Aufzug gleichzeitig mehrere Anfragen bearbeiten kann. Die meisten bekannten Schedulingalgorithmen gehen jedoch davon aus, daß eine Maschine immer nur einen Job bearbeitet. Desweiteren ist keine Start- und Endzeit für eine Anfrage vorgegeben, sondern diese muß gerade von der Aufzugssteuerung festgelegt werden. Bekannt ist nur die *minimale* Dauer für eine Anfrage, wenn diese ohne jede Interaktion mit anderen Anfragen ausgeführt wird. Sobald jedoch eine Interaktion stattfindet, zum Beispiel wenn zwei Passagiere gleichzeitig zusteigen, erhöht sich die Bedienzeit der Einzelanfragen. Es liegt also ein Problem vor, bei dem sich die Kosten dynamisch ändern und zwar in Abhängigkeit von der partiellen Lösung, die der Algorithmus gerade verfolgt.

Eine dritte und sehr entscheidende Abweichung ergibt sich aus dem rationalen Verhalten der Passagiere. Hält ein Aufzug auf einer Position und öffnet die Tür, so kann folgendes beobachtet werden:

- Alle Passagiere, die sich im Aufzug befinden und deren Ziel diese Position ist, werden aussteigen.
- Alle Passagiere, bei denen dies die Einstiegsposition ist und die dort noch warten, werden zusteigen.[1]

Das heißt also, die Steuerung kann nur indirekt auf die Anfragen einen Einfluß ausüben, indem sie die Reihenfolge der Positionen entsprechend festlegt. Wird eine Position angefahren, werden automatisch alle dort beginnenden Anfragen aktiv – ein *selektives Zusteigen* von Passagieren ist nicht *planbar*. Da für diese Klasse von Problemen keine Standardverfahren existieren, wurden eine Modellierung auf der Basis eines Constraint-Satisfaction Ansatzes und ein spezieller Vorwärtsplanungsalgorithmus entwickelt. Eine Implementierung erfolgte in der Programmiersprache Oz (Smolka 1995; Schulte, Smolke, & Würtz 1998; Würtz 1998).

[1] Bei Überfüllung des Aufzugs werden zwar einige Passagiere nicht zusteigen können, es läßt sich jedoch nicht vorhersagen, welche Passagiere dies sein werden.

Aufzugssteuerung als Constraintproblem

Eine Modellierung auf der Basis von Constraints ergibt sich natürlicherweise aus den Restriktionen, die jede zulässige Lösung zu erfüllen hat. Zunächst einmal beginnt jede Fahrt an der aktuellen Position des Aufzugs und muß jede Einstiegsposition eines Passagiers vor seiner Ausstiegsposition anfahren. Dies sind genau die Bedingungen aus Definition 2. Jeder Aufzug kann nur eine bestimmte Anzahl an Passagieren befördern, d. h. hält er auf einem Stockwerk, sollte genug freie Kapazität verfügbar sein, so daß alle dort wartenden Passagiere zusteigen können. Desweiteren sollte ein Modell gefunden werden, in das sich die folgenden Erweiterungen leicht einbetten lassen:

Zutrittslimitierung: Zu jedem Passagier gibt es eine (evtl. leere) Menge von Positionen, die dieser nicht erreichen darf. Zum Beispiel sollen Hotelgäste nicht in die Verwaltung gelangen.

Gegenseitiger Ausschluß: Für bestimmte Passagiergruppen soll deren gemeinsamer Transport unterbunden werden. Zum Beispiel sollen sich Hotelgäste und Personal nicht begegnen.

Begleitung: Für einige Passagiere wird eine indirekte Begleitung zwingend vorgeschrieben. Zum Beispiel sollen Kinder nur zusteigen dürfen und im Aufzug fahren, wenn ein Erwachsener anwesend ist. Während der Fahrt kann der Begleiter wechseln, aber ein „Festhalten“ ist nicht möglich, denn jeder Begleiter wird an seinem Zielstockwerk aussteigen.

Direktfahrt: Steigt ein Passagier ein, für den Direktfahrt vereinbart wurde, so wird er ohne Zwischenstop zu seiner Ausstiegsposition befördert.

Umwegefreiheit: Von der aktuellen Steuerung werden Passagiere generell ohne Umwege bedient, d. h. sie können lediglich Zwischenstops erfahren auf ihrem direktem Weg zum Ziel. Für die Modellierung soll auch untersucht werden, was passiert, wenn Passagiere Umwege fahren dürfen. Die Umwegefreiheit ist also nicht mehr garantiert, sondern muß explizit gefordert werden.

VIP Service: Einige Passagiere werden vorrangig befördert, zum Beispiel Feuerwehr und medizinisches Personal in Notfallsituationen. Alle anderen Passagiere erfahren gegebenenfalls eine Verzögerung.

Constraints

Seien n die Anzahl der Anfragen in (A, p) und r die Anzahl der sich aus dem Auftrag ergebenden relevanten Positionen (Definition 5). Zu jeder Anfrage $i = (a_i, b_i)$ ergeben sich $\texttt{Startposition}.i = a_i$ und $\texttt{Zielposition}.i = b_i$ als globale Konstanten. Die folgenden Variablen werden zur Modellierung der Anwendung eingeführt:

- `Länge`, ein Integer aus dem Intervall $[r, 2n+1]$. Es ist klar, daß jede zulässige Lösung innerhalb dieser Länge liegen muß, d. h. jede relevante Position muß mindestens einmal angefahren werden und jede zulässige Lösung besteht maximal aus $2n+1$ Positionen, falls die triviale Lösung zulässig ist.
- `Positionsfolge`, ein Tupel der Größe `Länge` mit Integern aus der Menge der relevanten Stockwerke (siehe Definition 5). Dies ist gerade die gesuchte zulässige Lösung.
- `Start` und `Ende`, zwei n-Tupel mit Integern aus dem Intervall $[1, \texttt{Länge} - 1]$ bzw. $[2, \texttt{Länge}]$. Diese Variablen geben an, bei welchem Element aus `Positionsfolge` die Passagiere ein- bzw. aussteigen.
- `Kapazität`, ein Tupel mit `Länge-1` Integern aus dem Intervall $[0, \texttt{Maximale Kapazität}]$. In dieser Variablen wird gespeichert, wieviele Personen bei der Fahrt von einer Position zur nächsten in der Kabine sind (genauer: welche Kapazität genutzt wird).

Auf den Variablen werden nun Constraints definiert, die die Eigenschaften der Anwendung deklarativ modellieren. Einige wichtige Constraints sind die folgenden:

1. $\forall i \in \{1, \ldots, N\} : \texttt{Start}.i <: \texttt{Ende}.i$
   Die Passagiere steigen erst aus, nachdem sie eingestiegen sind, das heißt, jede Einstiegsposition liegt vor der zugehörigen Ausstiegsposition.
2. $\forall i \in \{1, \ldots, N\}, \forall j \in \{1, \ldots, \texttt{Start}.i - 1\} :$
   $\texttt{Positionsfolge}.j \neq: \texttt{Startposition}.i$
   Die Passagiere steigen ein, sobald der Aufzug das erstemal die Startposition anfährt, das heißt alle vorhergenden Positionen in der Positionsfolge können nicht gleich dieser Startposition sein.
3. $\forall i \in \{1, \ldots, N\}, \forall j \in \{\texttt{Start}.i, \ldots, \texttt{Ende}.i - 1\} :$
   $\texttt{Positionsfolge}.j \neq: \texttt{Zielposition}.i$
   Sind die Passagiere in der Kabine, so steigen sie aus, sobald der Aufzug die Zielposition anfährt, das heißt, vorhergehende Positionen aus der Positionsfolge können nicht den Wert der Zielposition von i annehmen.
4. $\forall i \in \{1, \ldots, N\} :$
   $\texttt{Positionsfolge}.(\texttt{Start}.i) =: \texttt{Startposition}.i$ und
   $\texttt{Positionsfolge}.(\texttt{Ende}.i) =: \texttt{Zielposition}.i$
   Die Position der Kabine muß zum Einstiegszeitpunkt die Position sein, an der der Passagier wartet. Gleiches gilt für die Ausstiegsposition.
5. $\forall i \in \{1, \ldots, N-1\}, \forall j \in \{i+1, \ldots, N\} :$
   falls $\texttt{Startposition}.i = \texttt{Startposition}.j$, so ist
   $\texttt{Start}.i =: \texttt{Start}.j$
   Zwei Passagiere, die auf dem gleichen Stockwerk warten, werden zum gleichen Zeitpunkt einsteigen.

Die einzelnen Erweiterungen lassen sich durch das Hinzufügen weiterer Constraints modellieren. Zum Beispiel wird die Zutrittsbeschränkung durch folgenden Constraint modelliert:

$\forall i \in \{1,\ldots,N\}, \forall j \in \{$`Start`$.i,\ldots,$`Ende`$.i\}$:
`Positionsfolge`$.j \notin$ *Menge der verbotenen Stockwerke für* i.

Die Distribuierungsstrategie

Für die Variablen ist nun eine Belegung gesucht, die alle vorhandenen Constraints erfüllt. Aus der Belegung von `Positionsfolge` können unmittelbar die zulässigen Lösungen gewonnen werden. Minimale zulässige Lösungen ergeben sich aus Belegungen von `Positionsfolge` und minimalen Belegungen von `Länge`.

Der Constraintsolver distribuiert zunächst naiv über `Länge`, d. h. es wird zuerst der kleinste Wert für `Länge` probiert, dann der zweitkleinste etc. Dies stellt sicher, daß die erste gefundene Lösung minimal ist. Nachdem Länge determiniert, also festgelegt ist, können die restlichen Variablen dem Constraintspeicher hinzugefügt werden und die Constraintpropagierung schränkt die möglichen Werte der Variablen entsprechend dem Wert von `Länge` ein. Anschließend wird über die Tupel `Start` und `Ende` wiederum naiv distribuiert. Das heißt, es wird zunächst der erste Eintrag in `Start` vom kleinsten bis zum größten möglichen Wert entsprechend der Constraints durchprobiert, dann der zweite usw. Dies legt für jede Anfrage fest, welcher Position der Positionsfolge ihr Ein- und Ausstiegspunkt zugeordnet wird.

Der Constraintsolver terminiert, nachdem alle Positionsfolgen der Länge `Länge` erzeugt wurden. Allerdings werden unter diesen nur die zulässigen Lösungen tatsächlich auch als solche gewertet. Dies wird dadurch erreicht, daß bei jeder generierten Lösung überprüft wird, ob `Positionsfolge` determiniert ist. Jeder Eintrag der `Positionsfolge` ist determiniert, wenn es einen Passagier gibt, der an dieser Position ein- oder aussteigt. Ist dies nicht der Fall, so wurde eine unzulässige Positionsfolge erzeugt, die irrelevante Stops enthält.

Es hat sich gezeigt, daß intelligentere Distribuierungsstrategien (z. B. first fail oder diverse Splitstrategien) keinen Vorteil bringen, da die Constraintpropagierung den Suchraum nur sehr geringfügig einschränken kann. Prinzipiell ist zwar die Anfangsposition jeder Folge durch die Position des Aufzugs bestimmt, für alle weiteren Positionen kommen jedoch zunächst beliebige Permutationen der restlichen relevanten Positionen in Frage. Erst nachdem Start und Ende für eine Anfrage bekannt sind, ergeben sich Einschränkungen für die weitere Vervollständigung der Lösung.

Der Constraintsolver expandiert den Suchbaum maximal bis zur Länge von Lösungen von $2n + 1$. Eine Optimimierung entsprechend des Kriteriums (1) erfolgt nicht. Allerdings enthält die Menge aller Lösungen natürlich auch die *minimalen* Lösungen. Ebenso kann das Optimierungskriterium für alle zulässigen Lösungen berechnet und so die optimale Lösung bestimmt werden.

Die erste Lösung, die durch diese Strategie (bei linksorientierter Tiefensuche) gefunden wird, hat minimale Länge. Möchte man alle zulässigen Lösungen suchen, um etwa mittels *Branch and Bound* die optimale Lösung zu finden, so muß bis zur vollständigen Determinierung aller Einträge von `Positionsfolge` gesucht werden, da erst danach die Kosten der Lösung berechenbar sind.

Laufzeitverhalten des Constraintsolvers

Die folgenden zwei Beispiele illustrieren das typische Laufzeitverhalten des Constraintsolvers:

Beispiel 2 (Ohne Erweiterungen) Wir betrachten den folgenden Auftrag: ({(1,4), (2,5), (2,4), (5,3), (2,1)}, 2). Die Tür ist geöffnet. Alle Passagiere auf Stockwerk 2 sind bereits eingestiegen. Die verfügbare Kapazität des Aufzugs ist generell ausreichend und muß nicht berücksichtigt werden.

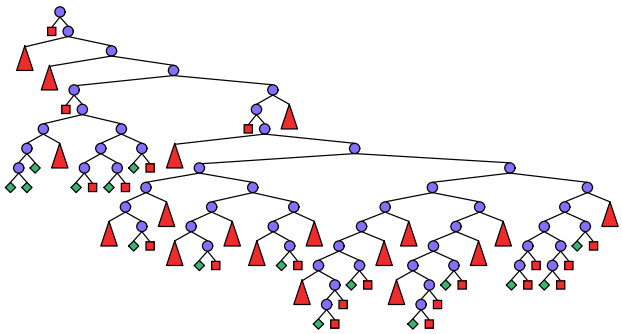

Abbildung 2: Der partiell expandierte Suchbaum des Constraintsolvers zu Beispiel 2. Der vollständige Baum besteht aus 64.777 Knoten. Ein auf der Spitze stehendes Quadrat symbolisiert eine Lösung. Ein auf der Seite stehendes Quadrat bedeutet, daß hier die Suche abgebrochen wurde, da sie keine bessere Lösung enthalten kann. Ein Dreieck kennzeichnet einen Unterbaum, der keine Lösungen enthält.

Abb. 2 zeigt den Suchbaum zu Beispiel 2. Die erste zulässige Lösung wird nach 0.04 s gefunden.[2] Nach etwa einer Sekunde sind *alle* zulässigen Lösungen gefunden. Nach 52.82 s ist allerdings erst bewiesen, daß der Suchraum keine weiteren zulässigen Lösungen enthält. Eine frühere Beendigung der Suche kann natürlich auf Codeebene realisiert werden, indem man die Größe der noch zu erwartenden Lösungen abschätzt. In der Praxis ist dies allerdings nicht unbedingt notwendig, da für die Berechnung des Angebots eines Aufzugs maximal nur eine Sekunde zur Verfügung steht. Insofern würde man die Suche nach einer Sekunde abbrechen und mit der bis dahin besten Lösung das Angebot des Aufzugs erhalten.

Beispiel 3 (Mit Erweiterungen) In diesem Beispiel sind für jede Anfrage bestimmte Erweiterungen gefordert, die jede zulässige Lösung einhalten muß.

[2] Alle Laufzeiten wurden auf einer Sparc 4 gemessen.

Anfrage	Erweiterungen
(2,4)	Direktfahrt
(2,6)	—
(1,4)	VIP, doppelte Kapazität
(7,2)	kein Zutritt auf 3, 4, 6
(5,3)	—
(6,7)	umwegefrei, nicht gleichzeitig mit den Passagieren der ersten vier Anfragen, doppelte Kapazität
(1,7)	kein Zutritt auf 5
(4,5)	—

Die Kabine steht auf Stockwerk 1 und die beiden Passagiere auf diesem Stockwerk sind bereits eingestiegen. Die maximale Kapazität beträgt drei Einheiten. Jeder Passagier benötigt eine Einheit, falls nicht anders angegeben. Den Suchbaum zu diesem Beispiel zeigt Abb. 3.

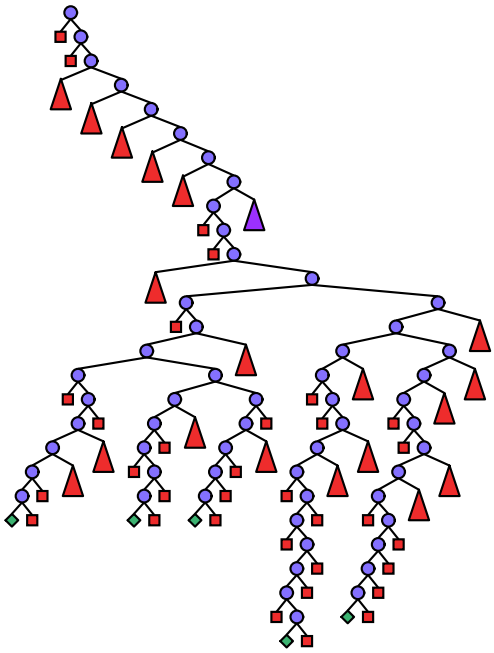

Abbildung 3: Der partiell expandierte Suchbaum des Constraintsolvers zu Beispiel 3. Die erste zulässige Lösung wird nach 0.46 s gefunden, nach etwa 2.5 s sind die fünf zulässigen Lösungen entdeckt. Der Beweis, daß es keine weiteren zulässigen Lösungen gibt, wurde nach einer Stunde erfolglos abgebrochen; zu diesem Zeitpunkt bestand der expandierte Baum aus mehr als 900.000 Knoten.

Domänenspezifische Vorwärtssuche

Grundansatz für die domänenspezifische Vorwärtssuche ist eine Überwachung des Zustands der Passagiere. Jeder Passagier kann sich nur in einem der folgenden Zustände befinden: *wartend*, *fahrend*, *am Ziel*. Jeder Knoten im Suchbaum entspricht einer relevanten Position, wobei die Wurzel mit der aktuellen Position des Aufzugs p bereits bestimmt ist. Die Nachfolgerknoten repräsentieren die möglichen nächsten Positionen. Das Wissen über den Status der Passagiere an der aktuellen Position schränkt die möglichen Werte für die Nachfolgerknoten sehr stark ein. Grundsätzlich kommen nur noch die relevanten Positionen in Frage, an denen Passagiere warten oder an denen fahrende Passagiere aussteigen wollen. Die Belegung der Nachfolgerknoten mit den möglichen Werten liefert eine vollständige Fallunterscheidung der nächsten Position. Ausgehend von jedem einzelnen Knoten werden wiederum die nun noch verbleibenden Möglichkeiten bestimmt. Die Suche terminiert, wenn die Menge der noch verbleibenden Positionen leer ist und alle zulässigen Lösungen gefunden wurden.

Die folgenden Erweiterungen ermöglichen es, die zulässigen nächsten Positionen weiter einzuschränken:

Zutrittslimitierung Die Menge der Positionen, die alle fahrenden Passagiere nicht erreichen dürfen, ist als Wert jedes Nachfolgerknotens unzulässig.

Direktfahrt Befindet sich ein Passagier im Aufzug, für den eine Direktfahrt vereinbart wurde, so steht der Wert des (einzig verbleibenden) Nachfolgerknotens sofort fest. Weitere Möglichkeiten sind unzulässig – es tritt keine Verzweigung im Suchraum auf.

Umwegefreiheit Befindet sich ein Passagier im Aufzug, der keine Umwege fahren darf, so können die Nachfolgerknoten nur noch mit Werten belegt werden, die zwischen der aktuellen Position und dem Ziel des Passagiers liegen. Fahren mehrere Passagiere ohne Umweg, so muß der Durchschnitt der individuell zulässigen Werte betrachtet. Fährt zum Beispiel ein Passagier von 3 nach 8 und ein anderer Passagier von 3 nach 1 und steigen beide in 3 zu, so erhält man $[3,8] \cap [3,1] = 3$, das heißt der Aufzug kann die aktuelle Position nicht verlassen. Der Nachfolgerknoten könnte nur den Wert des aktuellen Knotens erhalten, was unzulässig ist. Die Suche in diesem Zweig kann abgebrochen werden, da sie keine Lösung enthält.

Die Behandlung von VIPs ähnelt der der Direktfahrt und legt die nächsten zwei Positionen des Aufzugs entsprechend der Ein- und Ausstiegsposition des VIP sofort fest.

Die übrigen Erweiterungen schränken den Lösungsraum dadurch ein, daß sie zu unzulässigen Zuständen führen. Wird ein unzulässiger Zustand im Suchbaum erzeugt, so wird die Suche in diesem Zweig abgebrochen, da von diesem Zustand aus keine zulässige Lösung mehr erreichbar ist. Die folgenden Zustände sind zum Beispiel unzulässig:

Gegenseitiger Ausschluß: Zwei sich gegenseitig ausschließende Passagiere befinden sich im Zustand *fahrend*.

Begleitung: Ein Passagier, der eine Begleitung benötigt, ist *fahrend*, aber kein potentieller Begleiter ist *fahrend*.

Generell können immer nur soviele Passagiere *fahrend* sein, wie es die verfügbare Kapazität des Aufzugs zuläßt.

Als einzige Heuristik verwendet das Verfahren folgende Regel:

> *Wähle denjenigen Halt als nächstes, bei dem möglichst viele Passagiere ihren* Zustand *wechseln. Bei Gleichstand bevorzuge den Halt, zu dem die Fahrzeit am kürzesten ist.*

Zum Optimieren wird *Branch and Bound* mit Abschätzung benutzt. Dazu wird die bisherige Fahrzeit der Kabine berechnet. Steigt nun ein Passagier aus, so ist dessen Bedienzeit $T_i^{\ominus} - T_i^{\odot}$ exakt berechenbar. Zur Abschätzung der Gesammtsumme T_Σ geht man so vor: Falls ein Passagier noch *wartet* oder *fährt*, so addiert man zur aktuellen Summe der Bedienzeiten die Zeit, die mindestens benötigt wird, um die Anfrage dieses Passagiers zu bedienen. Dies ergibt eine untere Abschätzung, da durch eine Interaktion von Anfragen die jeweilige Bedienzeit nur zunehmen kann. Für jede partielle Lösung erhält man somit eine untere Abschätzung der Summe der Bedienzeiten. Ist diese Abschätzung bereits schlechter als der Wert der zum Zeitpunkt bekannten besten vollständigen Lösung, so braucht die partielle Lösung nicht weiter vervollständigt werden, da ihre Kosten dadurch nur noch weiter steigen können.

Laufzeitverhalten der Vorwärtssuche

Die Vorwärtssuche kann entweder alle zulässigen Lösungen generieren (Abbildungen 4 und 6 zeigen die Suchbäume der Beispiele 2 und 3) oder sie kann gezielt das Optimum bezüglich des Kriteriums der minimalen Bedienzeiten ermitteln (Abbildungen 5 und 7).

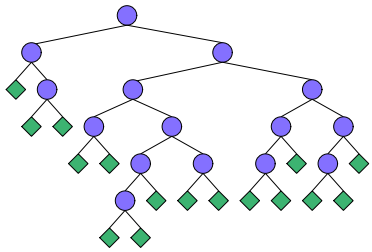

Abbildung 4: Der vollständig expandierte Suchbaum der anwendungsspezifischen Vorwärtssuche für Beispiel 2 mit nur 31 Knoten.

Im Vergleich zum Constraintsolver werden wesentlich kleinere Suchbäume generiert, was die Berechnung entsprechend beschleunigt: Die erste zulässige Lösung wird bei beiden Beispielen nach etwa 0.02 s gefunden. Um alle zulässigen Lösungen zu finden, benötigt der Algorithmus 0.07 s bzw. 0.06 s. Dies liegt daran, daß die Vorwärtssuche die Einschränkungen auf der Lösungsmenge besser propagieren kann als das domänenunabhängige Constraintverfahren.

Optimiert die Vorwärtssuche T_Σ und verwendet die *Branch and Bound* Abschätzung der Mindestkosten, so kann der Suchraum weiter beschränkt werden.

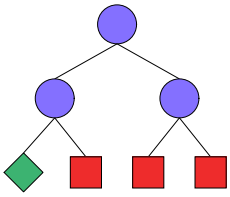

Abbildung 5: Beispiel 2 mit Optimierung. Vergleicht man diesen Suchbaum mit dem aus Abb. 4, so zeigt sich die Suchraumbeschränkende Wirkung des Optimierungskriteriums.

Das Optimum für Beispiel 2 ist die Lösung 2,1,4,5,3 mit $T_\Sigma = 21$. Diese Lösung wird (gesteuert durch die Heuristik) in einem sehr frühen Stadium der Suche gefunden (nach nur ca. 0.04 s) und beschränkt damit den Suchraum erheblich, da alle weiteren Lösungen eine höhere Kostenabschätzung haben.

Für die Berechnung der Bedienzeiten wurde vereinfachend angenommen, daß die Fahrt von einem Stockwerk zum nächsten eine Einheit dauert. Für das Ein- und Aussteigen, das Öffnen und Schließen der Türen wurde von null Einheiten ausgegangen.

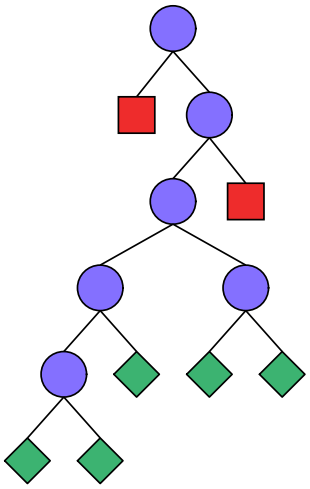

Abbildung 6: Der vollständig expandierte Suchbaum der anwendungsspezifischen Vorwärtssuche für Beispiel 3 mit nur 13 Knoten.

Für Beispiel 3 beobachtet man ein anderes Verhalten der Suche. Abb. 6 zeigt den vollständig explorierten Suchraum zur Generierung aller zulässigen Lösungen.

In diesem Beispiel wird die optimale Lösung als letzte gefunden, so daß sich der gleiche Suchraum wie bei

der Suche ohne Optimierung ergibt, vgl. Abb. 7. Allerdings werden zwei der ursprünglich als Lösung gewerteten Blätter, nämlich (2) und (3) nun nicht als Lösung gewertet, da ihre Kosten höher als die der Lösung in Blatt (1) sind.

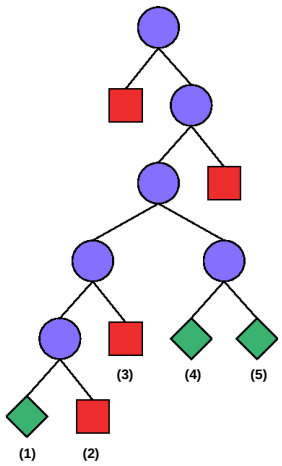

Abbildung 7: Beispiel 3 mit Optimierung. Hier konnte die Optimierung leider keinen Zeitgewinn bringen, da die optimale Lösung erst als letzte gefunden wird.

Die nachfolgende Tabelle zeigt die fünf Lösungen und ihre Kosten T_Σ in der Reihenfolge, in der sie von der Suchstrategie gefunden werden.

Knoten	**Lösung**	$\mathbf{T_\Sigma}$
(1)	1,4,7,2,4,5,6,7,3	98
(2)	1,4,7,2,4,5,3,6,7	102
(3)	1,4,7,2,4,6,7,5,3	102
(4)	1,4,7,5,2,4,6,7,3	92
(5)	1,4,7,5,2,4,3,6,7	90

Zusammenfassung und Ausblick

In diesem Artikel wird das Problem der Steuerung von Aufzugsanlagen mit Zielrufkonzept untersucht. Eine erste komplexitätstheoretische Analyse zeigt, daß es sich um ein NP-hartes Problem handelt. Eine Behandlung des Problems auf der Basis zustandsbasierter Suchverfahren bietet sich deshalb an. Zwei solcher Suchverfahren, die beide den Weg des Aufzugs planen, wurden vorgestellt:

Erstens, ein constraintbasierter Ansatz, der das Problem deklarativ modelliert und mit einem allgemeinen Constraintsolver löst, dessen Strategie zur Exploration des Suchraums nicht speziell an die Anwendung angepaßt wurde.

Zweitens, ein Vorwärtssuchverfahren, daß Information über die Struktur der zulässigen Lösung zur Verfügung hat und gezielt ausnutzt, um den Suchraum zu beschränken.

Bei der Constraintprogrammierung wird versucht, durch Constraints möglichst viel Wissen über das Problem deklarativ aufzuschreiben, welches dann von sogenannten Projektoren – Agenten, deren Aufgabe es ist, das Wissen aus einem Constraint auf den Suchraum zu übertragen – verarbeitet wird. Beim Problem der Aufzugssteuerung sind die Projektoren aber nicht oder nur sehr geringfügig in der Lage, den Suchraum einzuschränken, da die Information, die sie dafür benötigen, erst gegen Ende der Suche zur Verfügung steht. Dadurch wird fast der gesamte Suchraum durchsucht und das Verfahren ist relativ ineffizient, insbesondere wenn keine Anpassung des Constraintsolvers an die gewünschte Optimierung erfolgt.

Im Unterschied dazu ist die Vorwärtssuche sehr effizient und kann die Optimierungsfunktion zur gezielten Beschränkung des Suchraums nutzen. Ein wesentlicher Nachteil liegt auf der Hand: Die Erweiterungen sind fest im Code „verdrahtet“ und somit ist das Verfahren sehr inflexibel im Hinblick auf zukünftige Erweiterungen.

Ein flexibles und effektives Verfahren scheint möglich, indem eine deklarative Modellierung des Problems, zum Beispiel in einem Planungsformalismus, mit einer allgemeinen Vorwärtssuche kombiniert wird. Die Beschränkung der Suche ergibt sich in einem solchen Verfahren daraus, daß in einem unzulässigen Zustand keine Aktionen mehr anwendbar sind. Eine Verbindung von Vorwärtssuche und *Branch and Bound* ermöglicht dann die Generierung der optimalen Lösung.

Literatur

Garey, M. R., und Johnson, D. S. 1979. *Computers and Interactability, A Guide to the Theory of NP-Completeness.* New York: W.H. Freeman and Company.

Schulte, C.; Smolke, G.; und Würtz, J. 1998. Finite Domain Constraint Programming in Oz – A Tutorial. *DFKI Oz Documentation Series.*

Smolka, G. 1995. The Oz Programming Model. In van Leeuwen, J., ed., *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science.* Berlin: Springer-Verlag. 324–343.

Würtz, J. 1998. *Lösen kombinatorischer Probleme mit Constraintprogrammierung in Oz.* Ph.D. Dissertation, DFKI, Saarbrücken.