Synthese von Aufzugssteuerungen mit Hilfe von constraintbasierten Suchverfahren

DIPLOMARBEIT
an der Universität Freiburg
im Fach Informatik

von
Bernhard Seckinger

Freiburg,
den 27. Mai 1999

Vorwort

Modern elevators are strange and complex entities.
...
This is because they operate on the curious principle of 'defocused temporal perception'. In other words they have the capacity to see dimly into the immediate future, which enables the elevator to be on the right floor to pick you up even before you knew you wanted it, thus eliminating all the tedious chatting, relaxing, and making friends that people were previously forced to do whilst waiting for elevators.

— DOUGLAS ADAMS, *The Restaurant at the End of the Universe* (1980)

Die Firma Schindler Aufzüge AG ist seit mehreren Jahren mit einem neuen Aufzugssystem – MICONIC-10 – erfolgreich auf dem Markt. Im Unterschied zu herkömmlichen Systemen geben die Passagiere hier schon vor Fahrtantritt an einem Eingabeterminal das Zielstockwerk an und werden dann direkt einem der möglichen Aufzüge zugewiesen. Hinter dieser Zuweisung verbirgt sich ein Planungsprozeß, der in der bisher eingesetzten Version Passagiere mit gleicher Ein- bzw. Ausstiegsposition zusammenfaßt und damit gegenüber konventionellen Aufzugssteuerungen eine beachtliche Reduktion der durchschnittlichen Reisezeit erreicht.

In dieser Arbeit werden wir weitere Planungsalgorithmen vorstellen, diese miteinander vergleichen und auf ihre Tauglichkeit in der Praxis untersuchen, sowie Vorschläge für Lösungen der dabei auftretenden Probleme andiskutieren.

Ursprünglich lautete die Aufgabe, die Aufzugssteuerung durch Schedulingalgorithmen zu untersuchen. Wie wir jedoch schnell bemerkten, führt dieser Ansatz zu erheblichen Problemen. In der Sprache Oz lassen sich Schedulingalgorithmen in Form von Constraint-Satisfaction-Problemen (CSP) leicht implementieren. Mit diesen CSP-Tools gelang es uns, einen Constraintsolver zu schreiben, mit dem sich die Aufzugssteuerung planen läßt. Durch sukzessive Verbesserungen ergab sich daraus eine anwendungsspezifische Vorwärtssuche.

Nachdem in der Einleitung die Aufgabenstellung präzisiert und einige Definitionen eingeführt wurden, beschäftigt sich das zweite Kapitel mit der Komplexität des Problems. Anschließend werden wir zeigen, warum wir den Ansatz mit Schedulingverfahren nicht weiter verfolgt haben. In den Kapiteln 4 und 5, die den Kern der Arbeit darstellen, werden wir zwei Algorithmen vorstellen: Der erste basiert auf Constraintprogrammierung, während der zweite, ein Vorwärtssuchalgorithmus, aus diesem durch Verbesserungen an der Distribuierungsstrategie entstanden ist. Dieser Algorithmus bietet seinerseits weitere Verbesserungsmöglichkeiten. Im daran anschließenden sechsten Kapitel zeigen wir anhand von Beispielen und Performancediagrammen, wie gut diese Algorithmen sind. Abschließend betrachten wir das Problem der Gewährleistung der Beförderung, welches durch ständiges Neuplanen auftritt und untersuchen das Verhalten des Vorwärtssuchalgorithmus unter (fast) realen Bedingungen.

Danksagung

Bedanken möchte ich mich bei Dr. Jana Köhler, die mich während dieser Diplomarbeit betreute und immer ein offenes Ohr für Fragen und Probleme hatte.

In den Monaten, in denen diese Arbeit entstand, fand ich nicht allzuviel Zeit für meine Eltern. Dafür, daß sie dies verständnisvoll aufnahmen, möchte ich ihnen danken.

Weiterer Dank gebührt meinen Freunden in und um Grips, die mir in den schweren Stunden seelische Unterstützung zukommen ließen, und mir die Möglichkeit gaben, mich nach anstrengenden Tagen schnell zu erholen.

Bernhard Seckinger
Freiburg
Mai 1999

Inhaltsverzeichnis

1 Einleitung

1.1 Beschreibung der Situation

Die Firma Schindler Aufzüge AG hat mit der Einführung von MICONIC-10 ein ganz neues Konzept der Aufzugssteuerung auf den Markt gebracht. Bei diesem sogenannten Zielrufkonzept geben die Passagiere schon vor Fahrtantritt das Zielstockwerk an und werden dann einem der Aufzüge aus der Gruppe zugewiesen. Betreten die Passagiere den Aufzug, so sind keine weiteren Knöpfe zu drücken – der Transport zum Zielstockwerk erfolgt automatisch.

Durch dieses Konzept ist es möglich, die Passagiere einzeln zu erfassen und somit auf deren Bedürfnisse einzugehen – es können beispielsweise spezielle Dienste für behinderte Personen angeboten werden. Es werden aber auch zahlreiche andere Verbesserungen möglich: So kann eine Überfüllung der Kabine vermieden werden, was zu einer Steigerung des Komforts führt. Ganz wichtig ist natürlich auch die Reduktion der Bedienzeiten, welche durch Bündelung von ähnlichen Anfragen und der daraus resultierenden Vermeidung von überflüssigen Halten entsteht. Eine Folge hiervon ist die Steigerung der Beförderungskapazität wodurch ein Aufzug eingespart und mehr nutzbarer Raum für andere Zwecke zur Verfügung gestellt werden kann.

Das auf einem Angebotsmodell basierende System ist in Abb. 1 im Überblick dargestellt. Gibt ein Passagier am Eingabeterminal das Zielstockwerk ein, so wird eine Anfrage, bestehend aus Ein- und Ausstiegsposition, sowie gegebenenfalls weiteren Anforderungen, an die zentrale Prozeßsteuerung geschickt, welche die Anfrage an die Planungseinheiten der einzelnen Aufzüge weiterleitet.

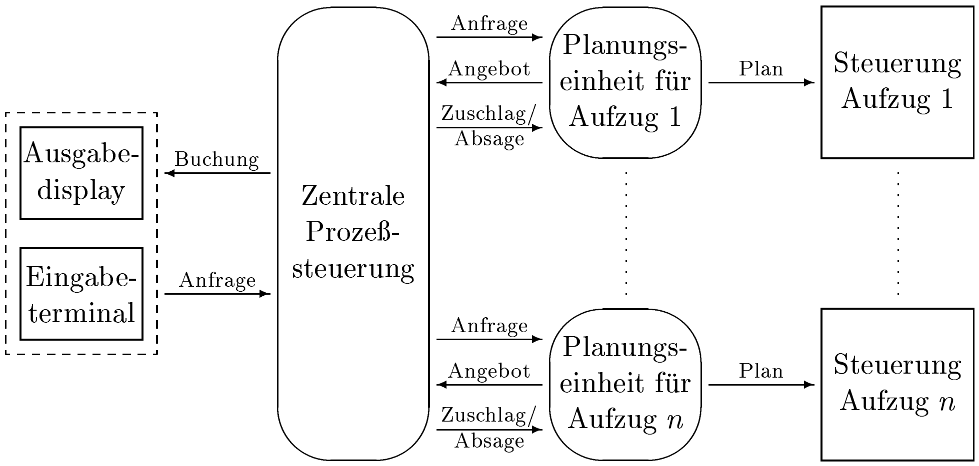


Abbildung 1: Eine schematische Darstellung des Aufzugssteuerungssystems.

Diese erstellen nun jede einen möglichst guten Plan und liefern ein darauf basierendes Angebot an die zentrale Steuerung zurück. Dabei müssen sie natürlich die auf diesen Aufzug gebuchten Anfragen, die noch nicht abgearbeitet wurden, sowie die Kapazität der Kabine berücksichtigen. Das kann insbesondere zur Folge haben, daß eine Anfrage abgelehnt wird.[1] Aus den verbleibenden Angeboten wählt die zentrale Steuerung nun das beste aus und erteilt diesem den Zuschlag und allen anderen eine Absage. Gleichzeitig wird dem Passagier auf dem Ausgabedisplay mitgeteilt, auf welchen Aufzug er gebucht wurde.

Der Nachteil dieses Systems besteht darin, daß hier nur lokal optimiert werden kann. Dies wird aber durch die dadurch entstehende Robustheit des Systems, z. B. bei Ausfall eines Aufzugs, mehr als wett gemacht.

Die Aufgabe des Planungsalgorithmus ist es nun, die Erstellung des Plans in einer Planungseinheit zu bewerkstelligen. Da man die Passagiere an den Eingabeterminals nicht zu lange warten lassen darf, steht hierfür nur eine Sekunde Planungszeit zur Verfügung. Deshalb wird im weiteren davon ausgegangen, daß die Planung nach einer Sekunde abgebrochen und der bis dahin beste Plan benutzt wird. Es wird also insbesondere angestrebt, möglichst früh einen möglichst guten Plan zu finden.

1.2 Definitionen

Mit Ausnahme von Kapitel 7 werden wir im folgenden nur noch die Planung und Angebotsberechung *eines* Aufzugs betrachten. Erhält die Planungseinheit eine Anfrage, so wird diese zu den noch nicht vollständig abgearbeiteten Anfragen hinzugenommen[2] und zusammen mit der aktuellen Position der Kabine liefert dies einen Auftrag, zu dem Lösungen – dies sind Pläne, die dem Auftrag gerecht werden – gesucht sind.

Um Verständnisfehler zu vermeiden, präzisieren wir hier die wichtigsten Begriffe:

Zu einem *Aufzug* zählt neben der *Kabine* auch noch der Schacht, die Maschinerie, die Steuerungs- und Planungseinheit, etc. Eine *Position* (auch Stockwerk, Etage) ist ein Haltepunkt der Kabine. Im folgenden ist die Menge der Positionen eines Aufzugs eine endliche Teilmenge von $\mathbb{Z}$. Als *Positionsfolge* wird eine endliche Folge von Positionen bezeichnet.

Die *Startposition* ist die Position, an der sich die Kabine zu Beginn des Plans befindet. Dieser Begriff benötigt aber noch weitere Erläuterungen: Zu dem

[1] Die Frage, was passiert, wenn alle Planungseinheiten ablehnen geht über den Rahmen dieser Arbeit hinaus und wird hier deshalb nicht weiter diskutiert.

[2] Passagiere, die schon in der Kabine sind, werden so behandelt, als ob sie bei diesem Halt zusteigen. Dies ändert nichts an der Planerstellung, vereinfacht aber die Aufgabenstellung.

Zeitpunkt, an dem eine Anfrage generiert wird, muß sich die Kabine nämlich nicht in einer Halteposition befinden. Zudem muß die Zeit berücksichtigt werden, die für das Planen benötigt wird, also etwa eine Sekunde, sowie die Zeit, die der Passagier benötigt, um zum richtigen Aufzug zu gehen. Legt man noch fest, daß Pläne nur während eines Halts geändert werden dürfen, so kann man die Startposition leicht berechnen.

Zudem nehmen wir an, daß an der Startposition die Türen bereits offen sind und insbesondere alle Passagiere, die dort warten, bereits zugestiegen sind. Diese Annahme ist realistisch, da der Aufzug entweder soeben dort angekommen ist, und dann natürlich die Türen geöffnet hat, oder aber er bearbeitet zur Zeit keine Anfragen, dann sind aber auch keine wartenden Personen auf dem Stockwerk.

Definition 1 Eine *Anfrage* ist ein Tupel (e, a), wobei e die Ein- und a die Ausstiegsposition ist. Ein *Auftrag* ist ein Tupel (s, A) bestehend aus der Startposition s und einer endlichen Menge A von Anfragen.

Wir werden einen Auftrag $(s, \{(e_1, a_1), \ldots, (e_n, a_n)\})$ der besseren Lesbarkeit wegen oftmals als $(s : e_1 \to a_1, \ldots, e_n \to a_n)$ notieren.

Definition 2 Eine *Lösung* eines Auftrags (s, A) ist eine Positionsfolge $(p_1, \ldots, p_l)$ mit den Eigenschaften

(i) $p_1 = s$

(ii) $\forall (e, a) \in A \ \exists i, j$ mit $i < j : p_i = e \wedge p_j = a$

Dabei ist l die *Länge* der Lösung.

Mit anderen Worten: Eine Lösung ist eine Positionsfolge, die bei der Startposition beginnt und allen Anfragen gerecht wird, indem jede Einstiegsposition einer Anfrage vor der dazugehörigen Ausstiegsposition auftritt.

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

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

Dabei ist das natürliche Verhalten der Passagiere zu beachten, da diese einsteigen, sobald die Kabine das erstemal die Einstiegsposition erreicht und danach aussteigen, sobald die Kabine das nächstemal in der Ausstiegsposition hält.

1.3 Die Problemstellung

Mit diesen Definitionen läßt sich nun die Aufgabenstellung präzise beschreiben. Wir definieren zuerst ein simplizistisches Grundproblem und bauen in Form von Erweiterungen darauf auf. Anschließend beschreiben wir noch drei Optimierungskriterien.

Grundproblem Unter dem Grundproblem verstehen wir einen Auftrag $\mathcal{A} = (s, A)$, zu dem wir eine zulässige Lösung $L = (p_1, \dots, p_l)$ suchen.

Wir betrachten folgende Erweiterungen:

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 Hotelgäste und Personal nicht gleichzeitig in der Kabine sein.

Begleitung Für einige Passagiere wird eine indirekte Begleitung zwingend vorgeschrieben. Zum Beispiel sollen Kinder nur in der Kabine sein dürfen, wenn auch ein Erwachsener anwesend ist. Die Begleitung kann dabei wechseln.

Umwegefreiheit Einzelne (oder auch alle) Passagiere dürfen nur in die Richtung ihres Ziels transportiert werden. Dabei können allerdings Zwischenstops erfolgen. Es wird also verboten, daß Passagiere in die falsche Richtung fahren.

Direktfahrt Einige Passagiere sollen direkt zu ihrem Ziel transportiert werden. Im Unterschied zur Umwegefreiheit dürfen dabei aber keine Zwischenstops erfolgen.

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

Kapazitätsbeschränkung Jedem Passagier wird ein gewisser Platzbedarf zugeordnet. Die Summe des Platzbedarfs der Personen in der Kabine darf die Kapazität der Kabine nicht übersteigen.

Es ist offensichtlich, daß bei einer konkreten Implementation auf Kapazitätsbeschränkung nicht verzichtet werden kann. Das gleiche gilt auch für die Umwegefreiheit, da die Passagiere verunsichert sind, wenn die Kabine in die falsche Richtung fährt.

Soll das Grundproblem um eine oder mehrere dieser Erweiterungen erweitert werden, so sind die Begriffe Anfrage, Auftrag und Lösung entsprechend zu erweitern.

Da wir an einem möglichst „guten“ Plan interessiert sind, benötigen wir ein Vergleichskriterium. Wir betrachten in dieser Arbeit drei verschiedene solche Kriterien: Das erste minimiert die Lösungslänge und ist mehr theoretischer Natur. In der Praxis möchte man die Reisezeiten der Passagiere – im folgenden Bedienzeiten genannt – möglichst niedrig halten. Es liegt also nahe, die Summe der Bedienzeiten zu minimieren. Wie wir jedoch in Kapitel 7 sehen werden, ist es vorteilhaft, die Summe der Quadrate der Bedienzeiten zu betrachten.

Definition 4 Eine Lösung L heißt *minimal*, falls die Länge dieser Lösung minimal unter allen Lösungen ist. Sie heißt *optimal* bzgl. einer Funktion T, dem Optimierungskriterium, falls $T(L)$ minimal unter allen Lösungen ist.

Wir weisen darauf hin, daß wir von minimalen bzw. optimalen Lösungen nicht verlangen, daß sie zulässig sind. Inwieweit dies dennoch zutrifft, werden wir in Satz 2 und Bemerkung 2 klären.

Wir betrachten ausschließlich die folgenden beiden Optimierungskriterien. Dazu sei $T_i^{\odot}$ der Zeitpunkt, zu dem die i-te Anfrage ausgelöst wurde, $T_i^{\ominus}$ der Ausstiegszeitpunkt des Passagiers und n die Anzahl der Anfragen:

$$T_{\Sigma}(L) = \sum_{i=1}^{n} \left(T_i^{\ominus} - T_i^{\odot}\right) \tag{1}$$

$$T_{\Sigma^2}(L) = \sum_{i=1}^{n} \left(T_i^{\ominus} - T_i^{\odot}\right)^2 \tag{2}$$

Im Falle von Kriterium (1) bzw. (2) sprechen wir auch von linerarer bzw. quadratischer Optimierung. Ist es in einem Zusammenhang gleich, welches der beiden Kriterien benutzt wird, so sprechen wir gelegentlich einfach von Optimierung bzw. optimaler Lösung.

Für die Zeitmessungen werden folgende Formeln zugrunde gelegt: Die Fahrt von Position x zu Position y dauert $\alpha \cdot |x-y| + \beta$ Sekunden. Sind $N^{\oplus}$ bzw. $N^{\ominus}$ die Anzahl der Passagiere die bei einem Halt ein- bzw. aussteigen, so dauert ein Halt $\eta \cdot (N^{\oplus} + N^{\ominus}) + \kappa$ Sekunden. Dabei gilt: $\alpha > 0$, $\beta, \eta, \kappa \geq 0$.

2 Komplexitätsbetrachtungen

2.1 Triviale und zulässige Lösungen

Wir betrachten zunächst das Grundproblem. Sei also $(s : e_1 \to a_1, \ldots, e_n \to a_n)$ ein Auftrag. Offensichtlich läßt sich, bei geeigneter Repräsentation, in konstanter Zeit eine Lösung, nämlich $(s, e_1, a_1, \ldots, e_n, a_n)$ berechnen. Wir werden diese Lösung im folgenden als *triviale* Lösung bezeichnen. Die triviale Lösung enthält aber in der Regel Irrfahrten, ist also nicht zulässig. Daß man aus einer Lösung effizient eine zulässige Lösung berechnen kann, zeigt folgender Satz:

Satz 1 Zu jeder Lösung $L = (p_1, \ldots, p_l)$ eines Auftrags $(s : e_1 \to a_1, \ldots, e_n \to a_n)$ läßt sich in $O(l \cdot n)$ eine zulässige Lösung L' berechnen.

Bemerkung 1 Ohne Beweis erwähnen wir, daß mit Hashtabellen der Aufwand auf $O(l + n)$ reduziert werden kann.

Beweis des Satzes Folgender Algorithmus leistet das Gewünschte: Nach dem Scanline-Prinzip durchläuft man L 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. □

Korollar 1 Bei Benutzung des soeben beschriebenen Algorithmus gilt mit den Bezeichnungen aus Satz 1: Die Länge von L' ist kleiner oder gleich der Länge von L. Es gilt: $T_\Sigma(L') \le T_\Sigma(L)$ und $T_{\Sigma^2}(L') \le T_{\Sigma^2}(L)$.

Beweis Da der Algorithmus nur Positionen aus der Lösung entfernt, gilt offensichtlich die erste Aussage. Dadurch werden die Bedienzeiten aber auf keinen Fall länger, womit die anderen beiden Aussagen gezeigt sind. □

Daß auch eine Umkehrung gilt, zeigt folgender Satz:

Satz 2 Sei $\mathcal{A}$ ein Auftrag. Dann gilt:

(i) Jede minimale Lösung von $\mathcal{A}$ ist zulässig.

(ii) In der Menge der optimalen Lösungen bzgl. des Kriteriums (1) (bzw. (2)) gibt es mindestens eine zulässige Lösung.

Beweis

(i) Sei L minimale Lösung zu $\mathcal{A}$ und L' die mittels des Algorithmus aus dem Beweis von Satz 1 gewonnene zulässige Lösung. Nach Korollar 1 ist L' aber höchstens so lang wie L und da L minimal ist folgt $L = L'$.

(ii) Sei L optimale Lösung zu $\mathcal{A}$ bzgl. (1) (bzw. (2)) und L' die mittels des Algorithmus aus dem Beweis von Satz 1 gewonnene zulässige Lösung. Nach Korollar 1 gilt $T_\Sigma(L') \leq T_\Sigma(L)$ (bzw. $T_{\Sigma^2}(L') \leq T_{\Sigma^2}(L)$) und aus der Optimalität von L folgt die Gleichheit. $\square$

Bemerkung 2 Genauso beweist man, daß, wenn man in den Formeln am Ende von 1.3 $\beta > 0$ oder $\kappa > 0$ wählt, *jede* optimale Lösung bzgl. (1) bzw. (2) zulässig ist.

Beispiel 1 Gegeben sei der Auftrag $(6 : 4 \to 5, 2 \to 3, 1 \to 4, 1 \to 2, 5 \to 1, 3 \to 1)$. Die triviale Lösung ist $(6, 4, 5, 2, 3, 1, 4, 1, 2, \mathbf{5}, 1, 3, 1)$. Diese ist aber nicht zulässig, da z. B. bei der fettgedruckten **5** schon alle Anfragen erledigt sind und somit weder Passagiere ein- noch aussteigen.

Mit dem Algorithmus aus dem Beweis von Satz 1 erhält man $(6, 4, 5, 2, 3, 1, 4, 2)$ als zulässige Lösung. Zu diesem Beispiel gibt es noch weitere 2529 zulässige Lösungen.

Daß diese Lösung nicht minimal ist, zeigt die Lösung $L_1 = (6, 1, 2, 3, 4, 5, 1)$, welche in der Tat eine der sechs minimalen Lösungen ist.

Sucht man für dieses Beispiel die optimalen Lösungen[3], so erhält man $L_2 = (6, 5, 3, 1, 2, 3, 4, 5)$. Weitere optimale Lösungen gibt es nicht. Es gilt: $T_\Sigma(L_2) = 140$ bzw. $T_{\Sigma^2}(L_2) = 3634$. Für die soeben angegebene minimale Lösung gilt: $T_\Sigma(L_1) = 154$ bzw. $T_{\Sigma^2}(L_1) = 4398$.

Wie man sieht, müssen optimale Lösungen nicht immer minimal sein. Es kann sogar sein – wie dieses Beispiel ebenfalls zeigt – daß keine minimale Lösung optimal ist.

2.2 NP-Härte

Wir werden nun zeigen, daß die Suche nach minimalen Lösungen des Grundproblems NP-hart ist. Dazu benötigen wir folgendes NP-vollständige Problem aus der Graphentheorie:

[3] Für die Zeitmessung wurde in den Formeln des letzten Absatz von 1.3 $\alpha = \beta = \eta = \kappa = 1$ gesetzt.

Definition 5 (FEEDBACK VERTEX SET (FVS))

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

Gesucht: Ein Feedback Vertex Set von G, das ist eine Teilmenge $V' \subseteq V$ mit $|V'| \leq k$ derart, daß $|V'|$ mindestens einen Knoten aus jedem gerichteten Zyklus in G enthält.

Ohne weiter auf diese einzugehen, geben wir hier einige Eigenschaften von FVS an, die für eine weitergehende Beschäftigung mit dem Thema interessant sein könnten:

Karp hat in [Kar72] gezeigt, daß FVS NP-vollständig ist. Beschränkt man sich auf reduzierbare Graphen, so liegt FVS allerdings in P [Sha79]. Dieses Ergebnis läßt sich aber nicht für unsere Zwecke nutzen, da das Aufzugsproblem in der Regel keine reduzierbaren Graphen liefert.

In [ENSS95] und [Sey95] werden zwar Approximationsalgorithmen mit Aufwand $O(\log |V| \cdot \log\log |V|)$ angegeben, Kann zeigt in [Kan92] aber, daß das zugehörige NP-Optimierungsproblem APX-hart ist, Approximationsalgorithmen, die beliebig gute Resultate liefern, also nicht zu erwarten sind.

Bevor wir nun das Problem AUFZUG-LÖSUNGSLÄNGE definieren können, benötigen wir noch den Begriff der relevanten Position. Dies sind die Positionen, die im Auftrag vorkommen. Genauer:

Definition 6 Sei $\mathcal{A} = (s, A)$ ein Auftrag. Eine Position p heißt *relevant*, falls $p = s$ oder $\exists (e, a) \in A : p = e \vee p = a$. Mit $\mathrm{Rel}(\mathcal{A})$ bezeichnen wir die Menge aller relevanten Positionen von $\mathcal{A}$.

Definition 7 (AUFZUG-LÖSUNGSLÄNGE (ALL))

Gegeben: Ein Auftrag $\mathcal{A} = (s, A)$ und eine natürliche Zahl $l \leq 2 \cdot |\mathrm{Rel}(\mathcal{A})| + 1$.

Gesucht: Eine zulässige Lösung mit Länge $\leq l$.

Satz 3 ALL ist NP-vollständig.

Beweis Wir zeigen zunächst ALL $\in$ NP. Dazu wird nichtdeterministisch eine Folge $p_1, \ldots, p_{l'}$ mit $l' \leq l$ festgelegt. Von dieser läßt sich nun in polynomieller Zeit entscheiden, ob es sich um eine zulässige Lösung handelt, beispielsweise, indem man den Aufzugsbetrieb simuliert.

Wir zeigen nun FVS $\leq_p$ ALL. Sei dafür $G = (V, E)$ ein gerichteter Graph und $k \leq |V|$ eine natürliche Zahl. Die Knoten seien o.B.d.A. von 1 bis $|V|$ durchnummeriert. Weiter können wir annehmen, daß G keine isolierten Knoten enthält, da wir sonst zu dem entsprechend eingeschränkten Graphen übergehen können.

Wir ordnen G nun einen Auftrag $\mathcal{A} = (s, A)$ und eine Zahl $l \in \mathbb{N}$ zu. Wir setzen $s := 0$, $A := E$ und $l := k + |V| + 1$. Wir bemerken, daß $\mathrm{Rel}(\mathcal{A}) = \{0\} \cup V$.

Wir zeigen nun:

$$\begin{array}{ll} & \textit{Es gibt ein FVS von } G \textit{ mit maximal } k \textit{ Elementen.} \\ \Longleftrightarrow & \textit{Es gibt eine zulässige Lösung } (p_1, \ldots, p_{l'}) \textit{ von } \mathcal{A} \textit{ mit } l' \leq l. \end{array}$$

„$\Longrightarrow$“ Sei S ein FVS von G mit $|S| \leq k$. Wir konstruieren nun einen Graphen $G' = (V', E')$. Dazu setzen wir $V' := V$ und $E' := \{(a, b) \in E : b \notin S\}$. Dieser Graph ist zyklenfrei, da aus jedem Zyklus in G mindestens eine Kante entfernt wurde. Es existiert also eine topologische Sortierung $\mathrm{ord} : V' \to \{1, \ldots, |V'|\}$, so daß mit $(a, b) \in E'$ auch $\mathrm{ord}(a) < \mathrm{ord}(b)$ gilt, welche (z. B. nach [OW90]) in $O(|V'| + |E'|)$ berechnet werden kann. Wir setzen nun $\tilde{p}_1 := 0$, $\tilde{p}_{\mathrm{ord}(e)+1} := e$ für $e \in V'$ und belegen $\tilde{p}_{|V|+2}, \ldots, \tilde{p}_{|V|+|S|+1}$ mit den Elementen aus S. Es ist nun $\tilde{l} := |V| + |S| + 1$ und $(\tilde{p}_1, \ldots, \tilde{p}_{\tilde{l}})$ eine Lösung von $\mathcal{A}$. Dies zeigen wir, indem wir die Bedingungen von Definition 2 prüfen:

(i) $\tilde{p}_1 = 0 = s$.

(ii) Sei $(e, a) \in A$.

1.Fall $(e, a) \in E'$. Dann ist $\mathrm{ord}(e) < \mathrm{ord}(a)$. Wähle $i := \mathrm{ord}(e) + 1$ und $j := \mathrm{ord}(a) + 1$.

2.Fall $(e, a) \notin E'$, d. h. $a \in S$, also gibt es ein $\tilde{p}_j$, $|V|+2 \leq j \leq |V|+|S|+1$ mit $\tilde{p}_j = a$. Wähle $i := \mathrm{ord}(e) + 1$.

Die so gewählten i und j erfüllen die Bedingung (ii).

Mit dem Algorithmus aus dem Beweis von Satz 1 läßt sich nun aus $(\tilde{p}_1, \ldots, \tilde{p}_{\tilde{l}})$ eine zulässige Lösung $(p_1, \ldots, p_{l'})$ berechnen. Nach Kor. 1 ist $l' \leq \tilde{l} = |V| + |S| + 1 \leq |V| + k + 1 = l$.

„$\Longleftarrow$“ Sei $L = (p_1, \ldots, p_{l'})$ eine zulässige Lösung von $\mathcal{A}$ mit $l' \leq l$. Wir setzen $S := \{p_i : 2 \leq i \leq l' \wedge \exists j \in \{2, \ldots, l'\} \setminus \{i\} : p_i = p_j\}$. S ist also gerade die Menge, der in L mindestens doppelt vorkommenden Positionen. Wir behaupten: S ist ein FVS. Dazu müssen wir zeigen, daß es zu jedem Zyklus aus G

und zu jeder zulässigen Lösung L von $\mathcal{A}$ einen Knoten gibt, der in L doppelt vorkommt. Sei Γ ein Zyklus in G. Alle Knoten aus Γ müssen in L mindestens einmal vorkommen, da sie in $\mathrm{Rel}(\mathcal{A})$ sind. Sei e_0 der Knoten aus Γ, der als erstes in L vorkommt. Da Γ ein Zyklus ist, gibt es eine Kante $(e_1, e_0) \in \Gamma$. L muß also zwingend die Form $(\ldots, e_0, \ldots, e_1, \ldots, e_0, \ldots)$ haben, und somit kommt e_0 doppelt in der Lösung vor.

Wir schätzen nun $|S|$ ab: Jedes Element aus $\mathrm{Rel}(\mathcal{A})$ kommt in L vor. Sei nun $p \in \mathrm{Rel}(\mathcal{A})$. Ist $p \in S$, so kommt p mindestens zweimal in L vor. S kann also höchstens $l' - |\mathrm{Rel}(\mathcal{A})|$ Elemente haben. Mithin ist $|S| \leq l' - |\mathrm{Rel}(\mathcal{A})| = l' - 1 - |V| \leq l - 1 - |V| = k$. □

Wie wir schon in Beispiel 1 gesehen haben, muß eine optimale Lösung nicht minimal sein. Deshalb läßt sich auch von Satz 3 nicht auf die Komplexität des Optimierungsproblems schließen – diese ist in der Tat noch offen.

Es liegt nahe, weitere Problemklassen zu untersuchen, wie z. B. die in Abschnitt 1.3 schon angedeutete: Alle Anfragen sind umwegefrei und die Kapazität des Aufzugs wird beachtet. Auch hier ist die Komplexität noch nicht bekannt.

3 Abgrenzung zu Schedulingproblemen

Schedulingprobleme werden gewöhnlich in drei Klassen eingeteilt (siehe z. B. [BESW94]): Die *reinen Schedulingprobleme*, die *reinen Ressourcenzuteilungsprobleme* und Probleme, die eine *Mischform dieser beiden* sind. Beim Ressourcenzuteilungsproblem werden die Aufgaben auf die Ressourcen zugeteilt, zudem sind die Anfangszeiten der Aufgaben bekannt. Bei unserem Aufzugsproblem haben wir aber nur eine Ressource (den Aufzug) und wollen die Anfangszeiten der Aufgaben berechnen. Somit haben wir es, wenn überhaupt, mit einem reinen Schedulingproblem zu tun. In [Wür98] wird ein reines Schedulingproblem so definiert:

Definition 8 Ein reines *Schedulingproblem* ist wie folgt definiert. T ist eine Menge von Aufgaben, R ist eine Menge von Ressourcen und H ist der Schedulinghorizont. Wir schreiben t für eine Aufgabe in T und r für eine Ressource in R. Es sind außerdem die folgenden Funktionen auf T und R gegeben.

$\mathrm{res} : T \to R$	$\mathrm{res}(t)$ ist die Ressource, auf der eine Aufgabe t ausgeführt wird
$\mathrm{cap} : R \to \mathbb{N}^+$	$\mathrm{cap}(r)$ ist die Kapazität einer Ressource r [...]
$\mathrm{dur} : T \to \mathbb{N}$	$\mathrm{dur}(t)$ ist die Ausführungszeit einer Aufgabe t [...]
$\mathrm{use} : T \to \mathbb{N}$	$\mathrm{use}(t)$ ist der Ressourcenverbrauch einer Aufgabe t [...]
[...]	

Für jede Aufgabe $t \in T$ ist eine Startzeit aus $\mathbb{N}$ zu finden, so daß zu keinem Zeitpunkt die auf einer Ressource $r \in R$ ausgeführten Aufgaben mehr von der Ressource verbrauchen, als es die Kapazität $\mathrm{cap}(r)$ zuläßt. Zusätzlich können eine Reihe weiterer Bedingungen gegeben sein.

Beim Aufzugsproblem haben wir genau eine Ressource, den Aufzug, die Aufgaben entsprechen den Anfragen. Die Kapazität der Ressource ist die Kapazität des Aufzugs und der Ressourcenverbrauch der Aufgaben ist der Platzverbrauch der Passagiere. Bei der Ausführungszeit stoßen wir jedoch auf ein Problem: Diese ist à priori nicht bekannt, sondern kann erst mit Kenntnis der Lösung berechnet werden.

Es ist üblich, zwischen disjunktiven und kumulativen Schedulingproblemen zu unterscheiden. Wärend bei ersteren jede Ressource immer nur eine Aufgabe gleichzeitig bearbeitet, ist die parallele Bearbeitung bei zweiteren möglich. Das Aufzugsproblem gehört offensichtlich zu den kumulativen Schedulingproblemen. Diese gelten, da weniger Constraints vorhanden sind, als schwieriger. Nach [AB93] darf ein kumulatives Schedulingproblem auch variable Ausführungszeiten haben, dort sind aber keine Lösungsalgorithmen angegeben.

Die in [Wür98] vorgestellten Algorithmen – verallgemeinertes Edge-Finding und ein auf Histogramm-Propagierung basierendes Verfahren – benutzen an wesentlicher Stelle, daß die Dauer der Aufgaben bekannt ist, und lassen sich somit nicht für unsere Ziele einsetzen. Alle weiteren uns bekannten Algorithmen sind Varianten dieser beiden, auf die das eben gesagte auch zutrifft.

Ein weiteres Problem, welches bei Benutzung von Schedulingalgorithmen Probleme bereitet, ist das rationale Verhalten der Passagiere. So beobachtet man, daß, wenn ein Aufzug an einer Position hält,

- alle Passagiere in der Kabine, deren Ausstiegsposition diese ist, aussteigen, und
- alle Passagiere auf diesem Stockwerk zusteigen.

Ein selektives Ein- und Aussteigen der Passagiere ist damit nicht planbar, dies ist aber eine der Voraussetzungen, von denen bei Schedulingproblemen ausgegangen wird.

Aufgrund dieser Probleme haben wir beschlossen, diesen Ansatz nicht weiter zu verfolgen.

4 Der Constraintsolver

4.1 Constraintprogrammierung

Constraintprogrammierung besteht im wesentlichen aus einer Menge von Variablen, Constraints auf diesen und einer Distribuierungsstrategie. In diesem Abschnitt werden wir diese Begriffe und wie sie zusammenhängen kurz erläutern. Dabei haben wir uns auf die für dieses Kapitel wichtigen Eigenschaften beschränkt. Für eine detailiertere Beschreibung siehe man in [Pug98], [SSW98] oder [Wür98] nach, wobei letztere beide sich auf die von uns benutzte Programmiersprache Oz [Smo95, Har98] beziehen.

Ein Constraintproblem besteht aus einer Menge von Variablen und Bedingungen auf oder zwischen diesen, den Constraints. Dabei dürfen die Variablen nur Werte aus einem endlichen Bereich (engl. finite domain) annehmen. Constraints, die diese Bereiche festlegen, werden als Grundconstraints bezeichnet.

Theoretisch ist jede Formel der Prädikatenlogik erster Stufe ein Constraint, in der Praxis möchte man aber mit diesem Constraint den Suchraum einschränken, was nur mit bestimmten Formeln gut funktioniert. Insbesondere bereiten Constraints mit Existenzquantoren Probleme und werden deshalb in Constraintprogrammiersprachen meist nicht unterstützt.

Beispiel 2 Wir suchen alle Paare $(x, y) \in \mathbb{N}^2$, für die gilt $x > y$. Dabei soll $x \in \{1, \ldots, 6\}$ und $y \in \{2, \ldots, 10\}$ gelten. Dies ist ein Constraintproblem mit den Variablen x und y. Die Grundconstraints für diese sind $x \in \{1, \ldots, 6\}$ und $y \in \{2, \ldots, 10\}$. Der einzige weitere Constraint lautet $x > y$.

Um ein solches Problem zu lösen, benutzt der Constraintsolver einen Constraintspeicher. In diesem werden die Wertebereiche der Variablen und für einen oder mehrere Constraints ein Projektor gespeichert. Dieser Projektor ist ein Agent, der aus den Wertebereichen der Variablen Werte entfernt, die *seinen* Constraints widersprechen.

In obigem Beispiel würde der Projektor für $x > y$ die Wertebereiche von x und y auf $x \in \{3, \ldots, 6\}$ und $y \in \{2, \ldots, 5\}$ einschränken. Diesen Vorgang nennt man Propagierung. Aus Effiziensgründen wird aber oftmals nicht verlangt, daß ein Projektor alle Werte entfernt, die aufgrund des Constraints entfernt werden könnten.

Gibt es mehrere Projektoren, so können diese im Wechselspiel die Wertebereiche immer weiter einschränken. Hätte man in obigem Beispiel den zusätzlichen

Constraint $3y > 2x$ gilt, so könnte ein Projektor für diesen Constraint schließen, daß $y \in \{3, \ldots, 5\}$ gelten muß. Der erste Projektor kann nun wiederum $x \in \{4, \ldots, 6\}$ schließen.

Dieses für Constraintsolver typische Wechselspiel endet irgendwann, nämlich dann, wenn kein Projektor die Bereiche der Variablen weiter einschränken kann oder der Wertebereich einer Variablen leer wird. Es könnte nun passieren, daß dieser stabile Zustand von der Reihenfolge, in der die Projektoren agieren, abhängt. Daß dies nicht der Fall ist, beweist Würtz in [Wür98].

Betrachten wir nochmal Beispiel 2: Dort hat unser Projektor die Anzahl der potentiellen Lösungen auf 16 eingeschränkt. Es gibt aber nur 10 Lösungen. Mit Hilfe der Projektoren alleine lassen sich diese nicht herausfiltern. Deshalb geht man dazu über, eine Fallunterscheidung zu machen – die Distribuierung. Dazu wählt man einen neuen Constraint C aus und dupliziert den Constraintspeicher. Dem Original fügt man nun den Projektor des Constraints C und der Kopie den Projektor des Constraints $\neg C$ hinzu. Nun können die Projektoren in beiden Constraintspeichern die Wertebereiche weiter einschränken.

Diese Distribuierung führt man solange fort, bis in jeder Version des Constraintspeichers maximal eine Lösung übrig bleibt.

In Abb. 2 sieht man einen Ausschnitt des Suchbaums zu Beispiel 2. Der Constraint C lautet hier $x = 3$.[4] Damit erhält man sofort $y = 2$ und hat eine

[4] Es handelt sich hier um die sogenannte *naive* Distribuierungstrategie. Bei einer *Split*-Strategie würde man etwa den Constraint $x \leq 4$ wählen.

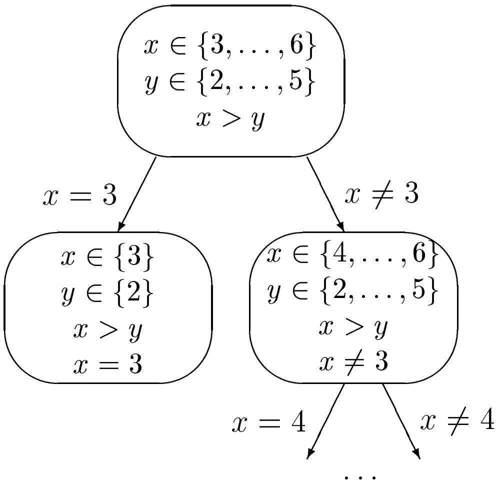


Abbildung 2: Der Anfang des Suchbaums für Beispiel 2. In den Knoten ist der Constraintspeicher aufgeführt, an den Kanten stehen die Constraints, mit denen distribuiert wird.

Lösung gefunden. Mit dem Constraint $x \neq 3$ erhält man $x \in \{4, \ldots, 6\}$ und $y \in \{2, \ldots, 5\}$. Nun muß man die Distribuierung in einem weiteren Schritt fortsetzen.

Durch dieses Zusammenspiel von Distribuierung und Propagierung können wir alle Lösungen bestimmen. Es ist offensichtlich, daß wir durch geschickt gewählte Projektoren und eine intelligente Distribuierungsstrategie – das ist das Verfahren mit dem der neue Constraint C bestimmt wird – die Suche erheblich beschleunigen können.

4.2 Die Constraints für das Grundproblem

Wir werden nun ein Constraintmodell für das Aufzugsproblem angeben und dieses anhand des folgenden Beispiels erläutern:

Beispiel 3 Die Kabine habe die Kapazität $k = 3$ und stehe an Position $s = 1$. Die acht zu bearbeitenden Anfragen sind in Tabelle 1 zu sehen.

Wir beschreiben nun ein Constraintmodell für das Aufzugsproblem.

Die Constraints sind auf vier Variablen definiert: `Pos` und `Länge` beschreiben den gesuchten Plan. Dabei ist `Pos` ein Tupel mit `Länge` Einträgen, welches die Folge der Positionen angibt. Da sich mit diesen beiden Variablen einige Constraints nur schlecht angeben lassen, benutzen wir noch `Start` und `Ende`. Diese beiden Tupel haben für jede *Anfrage* einen Eintrag. Hierin wird gespeichert, bei welchem Element von `Pos` die Passagiere zu- bzw. aussteigen.

	e	a	V	G	B	w	p
A_1	2	4	–	–	–	*direkt*	1
A_2	2	6	–	–	–	*irgendwie*	1
A_3	1	4	–	–	–	*vip*	2
A_4	7	2	$\{3, 4, 6\}$	–	–	*irgendwie*	2
A_5	5	3	–	–	$\{2, 4\}$	*irgendwie*	1
A_6	6	7	–	$\{1, 2, 3, 4\}$	–	*umwegefrei*	1
A_7	1	7	$\{5\}$	–	–	*irgendwie*	1
A_8	4	5	–	–	–	*irgendwie*	1

Tabelle 1: Anfragen zu Beispiel 3. In den Spalten stehen von links nach rechts: Die Einstiegsposition (e), die Ausstiegsposition (a), die verbotenen Stockwerke (V), die ausgeschlossenen Anfragen (G), Anfragen, die die Begleitung übernehmen können (B), die Art der Beförderung (w) und der Platzbedarf (p).

Betrachten wir die Lösung (1,4,7,2,4,5,3,6,7) zu Beispiel 3, so haben wir folgende Belegung für die Variablen: $\texttt{Pos}$ besteht gerade aus den Elementen der Lösung, also $\texttt{Pos}_1 = 1$, $\texttt{Pos}_2 = 4$, $\texttt{Pos}_3 = 7$, ... Die $\texttt{Länge}$ ist 9. Der Passagier der Anfrage A_1 möchte von Position 2 nach Position 4 fahren, er steigt also beim *vierten* Element von $\texttt{Pos}$ ein und beim *fünften* wieder aus. Somit ist $\texttt{Start}_1 = 4$ und $\texttt{Ende}_1 = 5$.

Wir geben nun die Grundconstraints für diese Variablen an. Dazu sei $\mathcal{A} = (s, \{(e_1, a_1), \ldots, (e_n, a_n)\})$ ein Auftrag mit n Anfragen und $\mathrm{Rel}(\mathcal{A})$ wie in Definition 6 die Menge der relevanten Positionen von $\mathcal{A}$:

(G1) $\qquad \texttt{Länge} \in \{|\mathrm{Rel}(\mathcal{A})|, \ldots, 2n+1\}$

Da jede relevante Position mindestens einmal in der Lösung vorkommen muß, muß $\texttt{Pos}$ mindestens soviele Einträge haben. Andererseits können durch jede Anfrage maximal zwei Positionen vorgegeben werden, macht zusammen mit der Startposition $2n+1$ als obere Abschätzung.

Eine Position ist eines der durch die Kabine erreichbaren Stockwerke. Damit erhalten wir, falls $\texttt{MinPos}$ das unterste und $\texttt{MaxPos}$ das oberste Stockwerk ist, die Abschätzung

(G2) $\qquad \forall i \in \{1, \ldots, \texttt{Länge}\} : \texttt{Pos}_i \in \{\texttt{MinPos}, \ldots, \texttt{MaxPos}\}$

Für $\texttt{Start}$ und $\texttt{Ende}$ können wir folgende Abschätzung angeben, da an der Startposition noch niemand aussteigt und beim letzten Halt niemand mehr zusteigt:

(G3) $\qquad \forall i \in \{1, \ldots, n\} : \texttt{Start}_i \in \{1, \ldots, \texttt{Länge} - 1\}$

(G4) $\qquad \forall i \in \{1, \ldots, n\} : \texttt{Ende}_i \in \{2, \ldots, \texttt{Länge}\}$

Zur Modellierung des Grundproblems benutzen wir folgende Constraints:

Die Bedingung (i) $p_1 = s$, aus Definition 2, läßt sich direkt umsetzen:

(G5) $\qquad \texttt{Pos}_1 = s$

Für die Bedingung (ii) $\forall (e, a) \in A\ \exists i, j$ mit $i < j : p_i = e \wedge p_j = a$, benötigen wir für jede Anfrage drei Constraints:

(G6) $\qquad \forall i \in \{1, \ldots, n\} : \texttt{Start}_i < \texttt{Ende}_i$

(G6′) $\forall i \in \{1, \ldots, n\} : \mathtt{Pos}_{(\mathtt{Start}_i)} = e_i$

(G6″) $\forall i \in \{1, \ldots, n\} : \mathtt{Pos}_{(\mathtt{Ende}_i)} = a_i$

Die Bedingung (iii) aus Definition 3 – es gibt keine p_i, $1 < i \leq l$, bei denen Personen weder ein- noch aussteigen – läßt sich nicht so einfach umsetzen, da hier Existenzquantoren in den Constraints vorkommen. Wir begnügen uns deshalb damit, nur einen Spezialfall von unzulässigen Lösungen durch Constraints auszufiltern. Dies erfordert allerdings, daß die Lösungen des Constraintsolvers überprüft werden. Das kann etwa mit dem im Beweis von Satz 1 skizzierten Algorithmus geschehen.

Der Spezialfall, den wir betrachten, verbietet lediglich, daß zwei aufeinanderfolgende Positionen gleich sind:

(G7) $\forall i \in \{1, \ldots, n-1\} : \mathtt{Pos}_i \neq \mathtt{Pos}_{(i+1)}$

Läßt man den Constraintsolver mit den bisherigen Constraints laufen, so kann es passieren, daß eine Lösung mehrfach auftritt. Sei etwa $(1 : 1 \rightarrow 2, 2 \rightarrow 4, 3 \rightarrow 2)$ ein Auftrag, so wird die Lösung $(1, 2, 3, 2, 4)$ doppelt ausgegeben. Der Unterschied besteht darin, daß einmal $\mathtt{Start}_2 = 2$ und einmal $\mathtt{Start}_2 = 4$ ist.[5] Der zweite Fall widerspricht jedoch dem rationalen Verhalten der Passagiere, diese steigen natürlich ein, sobald die Kabine das *erstemal* an der Einstiegsposition hält.

Wir benötigen deshalb Constraints für das rationale Verhalten der Passagiere:

(G8) $\forall i \in \{1, \ldots, n\}\ \forall j \in \{1, \ldots, \mathtt{Start}_i - 1\} : \mathtt{Pos}_j \neq e_i$

(G9) $\forall i \in \{1, \ldots, n\}\ \forall j \in \{\mathtt{Start}_i, \ldots, \mathtt{Ende}_i - 1\} : \mathtt{Pos}_j \neq a_i$

Der Constraint (G8) besagt, daß alle Positionen, die die Kabine anfährt bevor der n-te Passagier einsteigt, nicht dessen Einstiegsposition sein dürfen (da er sonst dort schon zusteigen würde). Mit anderen Worten: Der Passagier steigt zu, sobald die Kabine zum erstenmal bei ihm anhält.

Gleiches gilt für das Aussteigen: Passagiere, die in der Kabine sind, steigen aus, sobald das Ziel erreicht wurde, dies modelliert (G9).

Zu guter Letzt geben wir noch einige redundante Constraints an. Diese sind für das Lösungfinden zwar nicht notwendig, beschleunigen die Suche aber erheblich, da sie Werte von `Start` und `Ende` determinieren und somit der Suchbaum stark eingeschränkt wird.

[5] Sie wird sogar noch ein drittesmal ausgegeben, mit $\mathtt{Ende}_1 = 4$ und $\mathtt{Start}_2 = 2$. Die vierte denkbare Konstellation mit $\mathtt{Ende}_1 = \mathtt{Start}_2 = 4$ wird nicht ausgegeben, da diese nicht zulässig ist.

$$\text{(G10)} \qquad \forall i \in \{1,\ldots,n-1\}\ \forall j \in \{i+1,\ldots,n\}: \\ (e_i = e_j \Longrightarrow \texttt{Start}_i = \texttt{Start}_j) \wedge (e_i \neq e_j \Longrightarrow \texttt{Start}_i \neq \texttt{Start}_j)$$

$$\text{(G10}'\text{)} \qquad \forall i \in \{1,\ldots,n-1\}\ \forall j \in \{i+1,\ldots,n\}: \\ (e_i = a_j \Longrightarrow \texttt{Start}_i \leq \texttt{Ende}_j) \wedge (e_i \neq a_j \Longrightarrow \texttt{Start}_i \neq \texttt{Ende}_j)$$

$$\text{(G10}''\text{)} \qquad \forall i \in \{1,\ldots,n-1\}\ \forall j \in \{i+1,\ldots,n\}: \\ (a_i = e_j \Longrightarrow \texttt{Ende}_i \geq \texttt{Start}_j) \wedge (a_i \neq e_j \Longrightarrow \texttt{Ende}_i \neq \texttt{Start}_j)$$

$$\text{(G10}'''\text{)} \qquad \forall i \in \{1,\ldots,n-1\}\ \forall j \in \{i+1,\ldots,n\}: \\ a_i \neq a_j \Longrightarrow \texttt{Ende}_i \neq \texttt{Ende}_j$$

Diese Constraints modellieren folgende Beobachtungen:

(G10) Passagiere, die auf dem gleichen Stockwerk warten, steigen gleichzeitig ein. Passagiere, die auf verschiedenen Stockwerken warten, steigen *nicht* gleichzeitig ein.

(G10′) und (G10″) Ist die Einstiegsposition des einen Passagiers gleich der Ausstiegsposition des anderen, so steigt ersterer spätestens dann zu, wenn zweiterer aussteigt. Sind diese beiden Positionen verschieden, so steigt ersterer *nicht* ein, wenn zweiterer aussteigt.

(G10‴) Haben zwei Passagiere unterschiedliche Ausstiegspositionen, so steigen sie nicht gleichzeitig aus.

4.3 Die Erweiterungen

Wir geben nun, soweit möglich, Constraints für die Erweiterungen an.

4.3.1 Zutrittslimitierung

Um die Erweiterung Zutrittslimitierung implementieren zu können, müssen wir die Definition von Anfrage ausweiten: Wir schreiben $A = (e, a, V)$. Dabei sind e und a wie bisher Ein- und Ausstiegsposition und V ist eine Menge von Stockwerken – die Stockwerke, die verboten sind. Zu jeder Anfrage läßt sich nun ein Constraint angeben:

$$\text{(Z)} \qquad \forall i \in \{1,\ldots,n\}\ \forall j \in \{\texttt{Start}_i,\ldots,\texttt{Ende}_i\}: \texttt{Pos}_j \notin V$$

Dies heißt: Solange der i-te Passagier in der Kabine ist, darf die Position der Kabine nicht in V liegen.[6]

4.3.2 Gegenseitiger Ausschluß

Auch hier müssen wir den Begriff der Anfrage ausweiten – zu $A = (e, a, G)$. Dabei ist $G \subseteq \{1, \ldots, n\}$ und gibt an, welche der n Anfragen des Auftrags nicht gleichzeitig mit A bearbeitet werden dürfen. Die Bedingung, daß sich zwei Anfragen ausschließen, kann auch so formuliert werden: Die Bearbeitung der einen dieser beiden Anfragen darf erst begonnen werden, wenn die Bearbeitung der anderen bereits abgeschlossen ist. Dies ergibt die Bedingung:[7]

$$\text{(A)} \qquad \forall i \in \{1, \ldots, n\}\ \forall j \in G_i : (\texttt{Ende}_i \leq \texttt{Start}_j) \vee (\texttt{Ende}_j \leq \texttt{Start}_i)$$

Im Rückblick auf Kapitel 3 bemerken wir, daß dieser Constraint typisch für disjunktive Schedulingprobleme ist.

4.3.3 Begleitung

Analog zur Erweiterung Gegenseitiger Ausschluß weiten wir eine Anfrage aus, zu $A = (e, a, B)$, $B \subseteq \{1, \ldots, n\}$. Dabei gibt B die Anfragen an, die die Begleitung übernehmen können. $B = \emptyset$ bedeutet: Dieser Passagier benötigt keine Begleitung[8]. Als Constraint käme hier in Frage:

$$\text{(B)} \qquad \forall i \in \{1, \ldots, n\} \text{ mit } B_i \neq \emptyset\ \forall j \in \{\texttt{Start}_i, \ldots, \texttt{Ende}_i - 1\}\ \exists k \in B_i : \texttt{Start}_k \leq j < \texttt{Ende}_k$$

In Worten: Zu jedem Passagier, der Begleitung benötigt, gibt es an jeder Position, an der dieser Passagier in der Kabine ist, einen Begleiter, der vor oder an dieser Position zusteigt und nach dieser Position aussteigt.

Hier bereitet wieder der Existenzquantor Probleme bei einer konkreten Umsetzung in Oz, weshalb wir diesen Constraint nicht implementieren konnten. Wir überprüfen stattdessen die Lösungen des Constraintsolvers – ähnlich wie wir dies auch schon bei der Überprüfung der Zulässigkeit getan haben.

---

[6] Möchte man erlauben, daß Passagiere von Stockwerken wegfahren dürfen, die sie nicht erreichen dürfen, so muß dieser Constraint lauten:
$\forall i \in \{1, \ldots, n\}\ \forall j \in \{\texttt{Start}_i + 1, \ldots, \texttt{Ende}_i\} : \texttt{Pos}_j \notin V$

[7] Sollen sich die Passagiere auch nicht beim Ein- und Aussteigen begegnen, so sind die $\leq$ durch $<$ zu ersetzen.

[8] Ist in einem Auftrag eine Anfrage, die Begleitung benötigt, aber keine, die diese erbringen kann, so gehen wir davon aus, daß dieser Auftrag schon im voraus als unlösbar ausgesondert wurde.

4.3.4 Umwegefreiheit, Direktfahrt und VIP Service

Diese drei Erweiterungen können, da sie sehr ähnlich sind, gemeinsam behandelt werden. Diesmal erweitern wir eine Anfrage zu $A = (e, a, w)$ mit $w \in \{\mathit{irgendwie}, \mathit{umwegefrei}, \mathit{direkt}, \mathit{vip}\}$.

Wir erhalten folgende Constraints:

$$\text{(U1)} \qquad \forall i \in \{1, \ldots, n\} : w_i = \mathit{umwegefrei} \Longrightarrow \\ (e_i < a_i \Longrightarrow \forall j \in \{\texttt{Start}_i, \ldots, \texttt{Ende}_i - 1\} : \texttt{Pos}_j < \texttt{Pos}_{(j+1)}) \\ \wedge (e_i > a_i \Longrightarrow \forall j \in \{\texttt{Start}_i, \ldots, \texttt{Ende}_i - 1\} : \texttt{Pos}_j > \texttt{Pos}_{(j+1)})$$

Möchte der i-te Passagier *umwegefrei* fahren, so muß, solange dieser Passagier in der Kabine ist, die nächste Position näher am Ziel sein, als die aktuelle.

$$\text{(D)} \qquad \forall i \in \{1, \ldots, n\} : w_i = \mathit{direkt} \Longrightarrow \texttt{Ende}_i = \texttt{Start}_i + 1$$

Das heißt, bei einem Passagier, der *direkt* fahren möchte, folgt die Ausstiegsposition direkt auf die Einstiegsposition.

$$\text{(V)} \qquad \forall i \in \{1, \ldots, n\} : w_i = \mathit{vip} \Longrightarrow \\ (s = e_i \Longrightarrow \texttt{Ende}_i = 2) \wedge (s \neq e_i \Longrightarrow (\texttt{Start}_i = 2 \wedge \texttt{Ende}_i = 3))$$

Hat ein Passagier VIP Service gebucht, so gibt es zwei Fälle:

1. Die Startposition der Kabine ist die Einstiegsposition des Passagiers. Dann ist die nächste Position der Kabine die Ausstiegsposition.
2. Die Startposition der Kabine ist *nicht* die Einstiegsposition des Passagiers. Dann muß die Kabine sofort zu dessen Einstiegsposition fahren und danach sofort zu dessen Ausstiegsposition.

Im Falle der Umwegefreiheit können wir noch einen redundanten Constraint angeben, der abschätzt, um wieviel $\texttt{Ende}_i$ maximal größer sein kann, als $\texttt{Start}_i$, nämlich gerade die Anzahl der Stockwerke von der Einstiegsposition e_i bis zur Ausstiegsposition a_i:

$$\text{(U2)} \qquad \forall i \in \{1, \ldots, n\} : w_i = \mathit{umwegefrei} \Longrightarrow \texttt{Ende}_i \leq \texttt{Start}_i + |e_i - a_i| + 1$$

4.3.5 Kapazitätsbeschränkung

Für die Erweiterung Kapazitätsbeschränkung müssen wir zusätzlich zur Definition von Anfrage auch die Definition von Auftrag ausweiten: $A = (e, a, p)$ und $\mathcal{A} = (s, A, k)$. Dabei gibt $p \in \mathbb{N}$ den Platzbedarf und $k \in \mathbb{N}$ die Kapazität der Kabine an.

Für die Modellierung benötigen wir eine weitere Variable `Kap`, die zu jeder Fahrt der Kabine den benötigten Platzbedarf enthält. Der Grundconstraint ist

(K1) $\forall i \in \{1, \ldots, \texttt{Länge} - 1\} : \texttt{Kap}_i \in \{0, \ldots, k\}$

Wir berechnen nun für jede Fahrt den benutzten Platz und erhalten damit folgenden Constraint. (Dabei ist `CalcKap` eine Funktion, die aus `Start` und `Ende` berechnet, wieviel Platz die Passagiere in der Kabine bei der i-ten Fahrt benötigen, also gleich der Summe der p_i dieser Passagiere.)

(K2) $\forall i \in \{1, \ldots, \texttt{Länge} - 1\} : \texttt{Kap}_i = \texttt{CalcKap}(i)$

4.4 Die Distribuierungsstrategie

Es wird aufgefallen sein, daß wir bei den Grundconstraints für `Start`, `Ende`, `Pos` und `Kap` die Variable `Länge` benutzten. Damit dies funktioniert, muß `Länge` zu diesem Zeitpunkt schon determiniert sein. Wir distribuieren deshalb noch vor den Grundconstraints (G2)–(G4) und (K1) naiv über `Länge`. Dies bewirkt, daß wir für jede Belegung von `Länge` nach Lösungen suchen. Naive Distribuierung bedeutet dabei, daß wir mit dem kleinsten Wert die Suche beginnen – also in der Terminologie von Abschnitt 4.1 den Constraint C als $\texttt{Länge} = \mathit{kleinsterWert}$ wählen – und sukzessive größere Werte ausprobieren. Dadurch ist sichergestellt, daß die erste gefundene Lösung eine minimale Lösung ist.

Ist nun `Länge` determiniert, so können die Projektoren damit beginnen, den Suchraum zu verkleinern. Irgendwann wird dabei ein stabiler Zustand erreicht. Nun distribuieren wir, wiederum naiv, über `Start` und `Ende`. Bei der naiven Distribuierung über einem Tupel wird dabei zuerst der erste Eintrag vom kleinsten zum größten Wert durchprobiert, dann der nächste, etc.

Sind nun alle Einträge von `Start` und `Ende` determiniert, so werden durch (G6′) und (G6″) die Einträge in `Pos` berechnet. Es besteht nun die Möglichkeit, daß einige dieser Einträge noch nicht determiniert sind. Wir könnten deshalb einfach noch über `Pos` distribuieren, aber es geht auch eleganter:

Ist nämlich ein Eintrag von `Pos` noch nicht determiniert, so steigt bei diesem Halt weder jemand zu, noch aus, denn von diesem Passagier stünde ja `Start` und `Ende` schon fest, und damit auch die Position der Kabine. Also wissen wir, daß nur noch unzulässige Lösungen gefunden werden können. Wir brechen die Suche ab. Dies geschieht mit reflektierenden Constraints. Das sind Constraints, die den Wertebereich einer Variablen überprüfen. In unserem Fall muß gelten: $\forall i \in \{1, \ldots, n\} : |\mathrm{Dom}(\texttt{Pos}_i)| = 1$. Dabei ist $\mathrm{Dom}(X)$ der Wertebereich der Variablen X.

Schematisch sieht die Distribuierung also so aus:

- Den Grundconstraint für `Länge` dem Constraintspeicher hinzufügen.
- Naive Distribuierung über `Länge`.
- Die restlichen Constraints dem Constraintspeicher hinzufügen.
- Naive Distribuierung über `Start` und `Ende`.
- Überprüfen auf nichtdeterminierte Positionen.
- Überprüfen auf Zulässigkeit und die Erweiterung Begleitung.

Wie schon am Ende von Abschnitt 4.1 angedeutet, kann durch eine intelligente Distribuierungsstrategie die Suche erheblich beschleunigt werden. Dazu zwei Bemerkungen:

1) Von den Standardstrategien (naiv, first fail, sowie diversen split-Strategien [SSW98]) ergaben Tests, daß die naive am besten ist.

2) Bei der Untersuchung, warum dies so ist, sowie dem Versuch, eine bessere Nichtstandardstrategie zu entwickeln, ist der spezifische Vorwärtssuchalgorithmus entstanden, den wir im nächsten Kapitel beschreiben werden. Dort werden wir auch diese Untersuchung genauer schildern.

4.5 Optimierung

Ähnlich wie bei der Erweiterung Begleitung und beim Überprüfen der Zulässigkeit läßt sich die Optimierung nicht brauchbar durch Projektoren ausdrücken. Deshalb beschränken wir uns darauf, die Lösung mit der bisher besten zu vergleichen und, falls die neue Lösung schlechter ist, diese abzulehnen. Die letzte, auf diese Weise gefundene Lösung, ist dann die optimale Lösung. Es handelt sich also um eine *Branch and Bound* Strategie, bei der sich der Anteil des „Bound“ allerdings auf die Blätter im Suchbaum beschränkt.

Wir verwenden eine Variable `Bedienzeitsumme`, die gerade den Wert von T_Σ bzw. T_{Σ^2} enthält. Durch den Suchalgorithmus wird nun jeder Kopie des Constraintspeichers zusätzlich der Constraint `Bedienzeitsumme`<*Bedienzeitsumme der bisher besten Lösung* injiziert. Damit wird sichergestellt, daß neue Lösungen besser sind.

4.6 Ein Beispiel

Wir skizzieren nun, wie der Constraintsolver den Auftrag aus Beispiel 3 löst:

Zuerst wird $\texttt{Länge} \in \{7, \ldots, 17\}$ festgesetzt und darüber distribuiert, also der Constraintspeicher verdoppelt und die Constraints $\texttt{Länge} = 7$ bzw. $\texttt{Länge} \neq 7$ hinzugefügt. Bei einer linksorientierten Tiefensuche wird der erste dieser beiden zuerst bearbeitet.

Für $\texttt{Länge} = 7$ bzw. $\texttt{Länge} = 8$ gibt es keine Lösungen. Um dies zu erkennen, benötigt der Constraintsolver 20 bzw. 50 Distribuierungsschritte. Wir werden hier nicht weiter darauf eingehen, was in dieser Zeit berechnet wird – dafür verweisen wir auf das nächste Kapitel, wo wir, an einem anderen Beispiel, durch diese Analyse den Vorwärtssuchalgorithmus motivieren.

Da alle zulässigen Lösungen die Länge 9 haben, gibt es für $\texttt{Länge} \in \{10, \ldots, 17\}$ ebenfalls keine Lösung. Da wir aber davon ausgehen, daß die Suche nach einer Sekunde abgebrochen wird, haben wir keine Anstrengung unternommen, den daraus resultierenden Suchprozeß zu reduzieren.

Wir gehen nun davon aus, daß, durch die Distribuierung, $\texttt{Länge} = 9$ festgelegt wurde. Aus (G2)–(G4) und (K1) folgt nun:

- $\forall i \in \{1, \ldots, 9\} : \texttt{Pos}_i \in \{1, \ldots, 7\}$
- $\forall i \in \{1, \ldots, 8\} : \texttt{Start}_i \in \{1, \ldots, 8\}$ und $\texttt{Ende}_i \in \{2, \ldots, 9\}$
- $\forall i \in \{1, \ldots, 8\} : \texttt{Kap}_i \in \{0, \ldots, 3\}$

(G5) setzt $\texttt{Pos}_1 = 1$ fest. Da Anfrage 3 VIP Service beansprucht, wird durch (V) $\texttt{Pos}_2 = 4$ determiniert. Aus (G7) folgt noch $\texttt{Pos}_3 \neq 4$.

Mehr können die Constraints zu diesem Zeitpunkt nicht ausschließen – es wird also ein Distribuierungsschritt nötig: Zuerst versucht der Constraintsolver $\texttt{Start}_1$ festzulegen. Die Werte 1 und 2 kommen nicht in Frage, da A_1 an Position 2 startet, die ersten beiden Positionen aber schon (wegen des VIP Services) zu 1 bzw. 4 festgelegt wurden, und damit $\neq 2$ sind.

Aus $\texttt{Start}_1 = 3$ folgt, daß der Plan mit $(1, 4, 2, \ldots)$ beginnt. Nach diesen Schritten finden sich die Passagiere der Anfragen A_1, A_2, A_7 und A_8 in der Kabine mit einem Gesamtplatzbedarf von $4 > 3 = k$, womit auch $\texttt{Start}_1 = 3$ ausgeschlossen ist.

Erst $\texttt{Start}_1 = 4$ ermöglicht einen konsistenten Plan. Nun kann durch (G10) $\texttt{Start}_2 = 4$ geschlossen werden. (D) liefert $\texttt{Ende}_1 = 5$. Durch (G6), (G7), (G10), sowie (U1) und (U2) können noch einige Werte für die Variablen ausgeschlossen werden.

Nun wird weiter distribuiert. Da $\texttt{Start}_1$ und $\texttt{Start}_2$ schon determiniert sind, wird der kleinste Wert, den $\texttt{Start}_3$ annehmen kann, ausprobiert – dies ist 1.

Dieses Spiel setzt sich fort, bis im 18. Distribuierungsschritt die Lösung $(1, 4, 7, 2, 4, 5, 3, 6, 7)$ gefunden wird.

In Abb. 3 sieht man den zugehörigen Suchbaum. Dabei sind Teilbäume, die nicht auf dem Weg zur Lösung liegen, durch Dreiecke symbolisiert. In dem am weitesten rechts befindlichen Dreieck sind übrigens noch drei weitere Lösungen versteckt: $(1, 4, 7, 5, 2, 4, 3, 6, 7)$, $(1, 4, 6, 7, 2, 4, 5, 3, 6)$ und $(1, 4, 6, 7, 5, 2, 4, 3, 6)$.

4.7 Korrektheit und Vollständigkeit

Zum Abschluß des Kapitels zeigen wir noch, daß der Constraintsolver korrekt und vollständig ist.

Satz 4 Der Constraintsolver ist korrekt, d. h. jede Lösung, die der Constraintsolver ausgibt, ist tatsächlich eine zulässige Lösung, die gegebenenfalls auch die Erweiterungen respektiert, und er ist vollständig, d. h. falls L eine zulässige Lösung ist, so wird diese vom Constraintsolver berechnet.

Beweis

Korrektheit: Sei $L = (p_1, \ldots, p_l)$ eine vom Constraintsolver berechnete Positionsfolge. Wegen (G5) und (G6) ist dies eine Lösung. Wegen der Routine, die auf Zulässigkeit überprüft, ist sie sogar zulässig. Zusammen mit (G8) und (G9) werden die notwendigen Eigenschaften für die Erweiterungen durch (Z), (A), (U1), (D), (V), (K1) und (K2) gesichert. Für die Erweiterung Begleitung werden die Lösungen überprüft. L ist also eine zulässige Lösung.

Vollständigkeit: Sei $L = (p_1, \ldots, p_l)$ eine zulässige Lösung. Auf diese Lösung treffen alle Constraints zu. Durch die Distribuierung wird sichergestellt, daß alle Lösungen innerhalb der durch die Grundconstraints angegebenen Bereiche gefunden werden. Deshalb findet der Algorithmus auch L. □

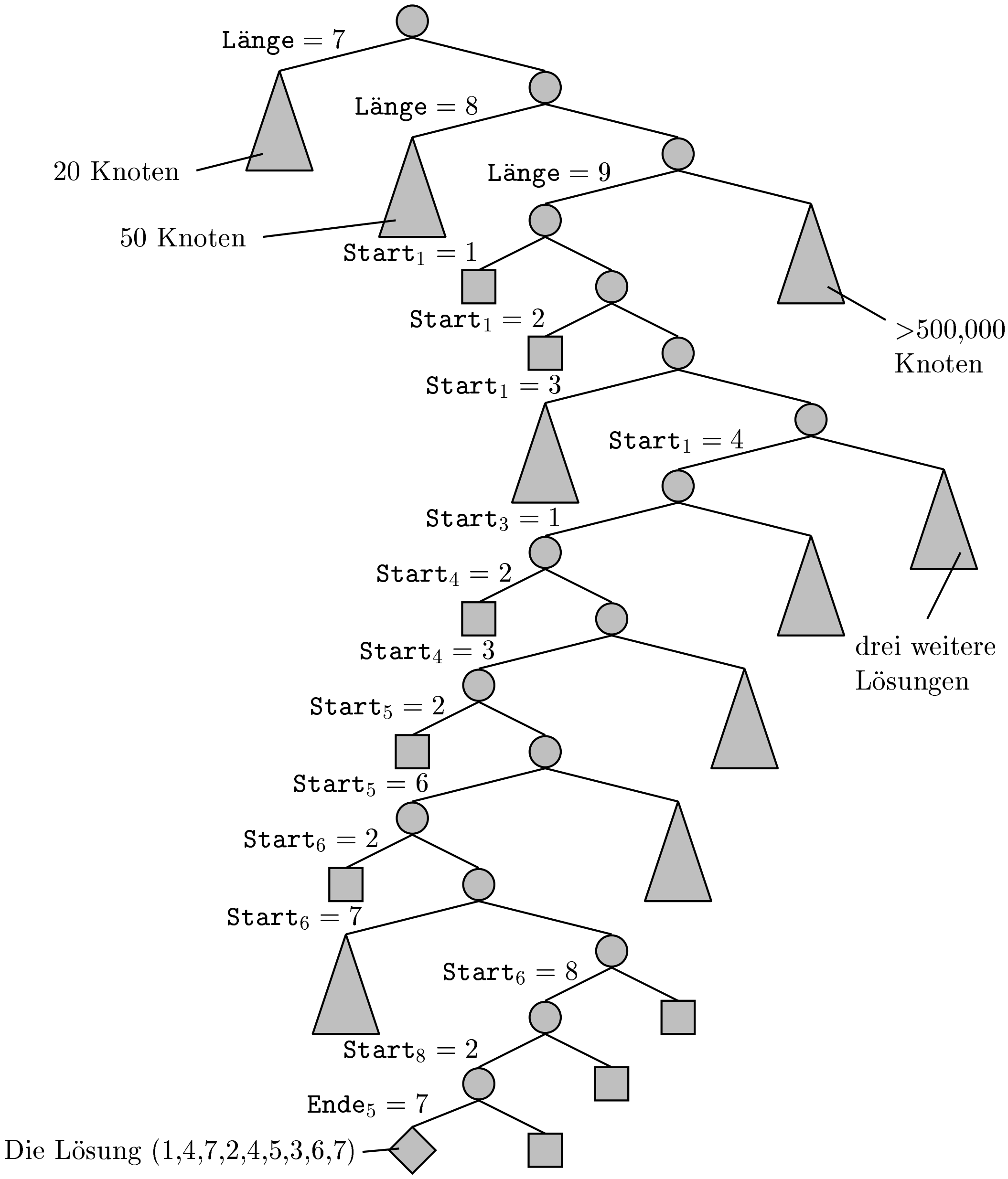


Abbildung 3: Der Ast im Suchbaum zu Beispiel 3, der zur Lösung $(1, 4, 7, 2, 4, 5, 3, 6, 7)$ führt. An den linken Zweigen steht der Constraint, der beim Distribuieren dem Constraintspeicher hinzugefügt wurde. An den rechten Ast gehört der negierte Constraint, den wir weggelassen haben, damit die Abbildung nicht zu überladen ist.

5 Die Vorwärtssuche

5.1 Vom Constraintsolver zur Vorwärtssuche

Um den Constraintsolver zu verbessern, schauten wir uns etwas genauer an, wie dieser einige Beispiele löst. Als wir die Beobachtungen, die wir dabei machten, umgesetzt hatten, mußten wir feststellen, daß das Ergebnis ein Vorwärtssuchalgorithmus war. Wir werden nun zuerst an einem Beispiel die Beobachtungen aufzeigen und dann den daraus entstandenen Algorithmus beschreiben.

Dazu betrachten wir detailiert, wie der Constraintsolver Beispiel 1 aus Kapitel 2 bearbeitet ($6 : 4 \to 5, 2 \to 3, 1 \to 4, 1 \to 2, 5 \to 1, 3 \to 1$). In Tabelle 2 sind die Wertebereiche der Variablen aus dem Constraintspeicher nach einigen Distribuierungsschritten zu sehen.

In der obersten Zeile steht der Constraint, der zuletzt dem Constraintspeicher durch die Distribuierung hinzugefügt wurde, darunter sind die, danach vom Constraintsolver berechneten Wertebereiche der Variablen. In der ersten Hauptspalte gilt etwa für die erste Anfrage ($4 \to 5$), daß $\texttt{Start}_1$ die Werte 1 bis 5 und $\texttt{Ende}_1$ die Werte 2 bis 6 annehmen kann. Für `Pos` gilt: $\texttt{Pos}_1 = 6$ (da dies die Startposition ist), $\texttt{Pos}_2$ liegt im Bereich 1 bis 5, etc.

Hier fallen die folgenden Eigenheiten auf:

1. Dadurch, daß zuerst über `Länge` distribuiert wird, wird sichergestellt, daß die erste gefundene Lösung minimal ist. Zuvor wird aber in der Regel

	`Länge` = 6		$\texttt{Start}_1 = 2$		$\texttt{Start}_2 = 3$		$\texttt{Start}_3 = 4$	
	`Start`	`Ende`	`Start`	`Ende`	`Start`	`Ende`	`Start`	`Ende`
$4 \to 5$	1–5	2–6	2	3–6	2	4–6	2	5–6
$2 \to 3$	1–5	2–6	1,3–5	3–6	3	4–6	3	5–6
$1 \to 4$	1–5	2–6	1,3–5	2–6	1,4–5	2,4–6	4	5–6
$1 \to 2$	1–5	2–6	1,3–5	3–6	1,4–5	3–6	4	5–6
$5 \to 1$	1–5	2–6	1,3–5	3–6	1,4–5	4–6	1–5	4–6
$3 \to 1$	1–5	2–6	1,3–5	3–6	1,4–5	4–6	1–5	4–6
`Pos`	6,1–5,1–6, 1–6,1–6,1–6		6,4,(1–3,5,6), 1–6,1–6,1–6		6,4,2,(1,3–6), 1–6,1–6		6,4,2,1,2–6,1–6	
`Länge`	6		6		6		6	

Tabelle 2: Die Variablen aus dem Constraintspeicher zu Beispiel 1. In den Hauptspalten sind einige Zustände, die im Laufe der Suche eintreten, festgehalten.

nach kürzeren Lösungen gesucht. Daß diese aussichtslose Suche nicht umgangen werden kann, folgt aus Satz 3 und der Annahme $P \neq NP$.

Möchte man also die Suche nach *optimalen* Lösungen beschleunigen, so liegt es nahe, die Distribuierung über `Länge` aufzugeben, und ein neues Konzept zu entwickeln, wie die Länge von `Pos` determiniert werden kann.

Am Beispiel sieht man, daß die Suche nur deshalb abgebrochen wird, weil die Wertebereiche von `Start` und `Ende` zu klein sind und somit keine Werte für diese Variablen existieren, so daß alle Constraints erfüllt sind. Ohne diese Einschränkung würde die Suche direkt zu der zulässigen Lösung $(6, 4, 2, 1, 5, 3, 4, 2, 1)$ führen.[9] Das erste der am Ende von Abschnitt 1.1 angegebenen Ziele – einen Plan möglichst früh zu finden – wäre damit erreicht.

2. Immer dann, wenn ein neuer Eintrag von `Pos` determiniert wird, ergeben sich auch viele Änderungen bei den anderen Variablen. In der Tabelle sind genau die ersten vier dieser Schritte aufgeführt, da sich zwischendurch fast nichts ändert. So liegt etwa zwischen dem zweiten und dritten Tabelleneintrag noch ein Schritt der mit dem zweiten fast identisch ist, es wird lediglich bei der zweiten Anfrage $(2 \to 3)$ der Wert 1 für `Start` ausgeschlossen.

   Es liegt deshalb nahe, statt über `Start` und `Ende` über `Pos` zu distribuieren. Dabei kann es vorkommen, daß schon alle Werte von `Start` und `Ende` feststehen, `Pos` aber noch nicht vollständig durchsucht wurde, wie wir dies in Abschnitt 4.4 schon erwähnten. Wenn beispielsweise $(1, 2, 5, 4, 3)$ eine zulässige Lösung ist, so kann während der Suche der Plan $(1, 2, ?, 5, 4, 3)$ lauten, wobei der Wert für die Stelle mit dem Fragezeichen noch berechnet werden muß. Es gibt hier aber keinen Wert mehr, mit dem diese Lösung zulässig wird. Bei naiver Distribuierung über `Pos` würden diese Fälle aber alle durchsucht. Deshalb ist es sinnvoll, statt naiver Distribuierung eine spezifische Strategie zu verwenden.

3. Bei der Constraintprogrammierung ist es oft vorteilhaft, die Constraints so anzulegen, daß sie den Suchraum möglichst klein halten. Deshalb haben wir die Constraints des Constraintsolvers so formuliert, daß möglichst viele Werte von `Start` bzw. `Ende` ausgeschlossen werden können. Möchte man aber über `Pos` distribuieren, so sind die Constraints so anzupassen, daß sie möglichst viele Werte von `Pos` ausschließen. Dies gelingt besonders gut für den ersten nicht determinierten Eintrag von

[9] In der Tat führt der Constraintsolver genau die gleichen Rechnungen mit `Länge`=7,8,9 noch einmal durch, bis endlich die Lösung gefunden wird.

`Pos` – den nächsten Halt –, da für diesen berechnet werden kann, welche Passagiere sich bei der Fahrt dorthin in der Kabine befinden. Dies legt es nahe, während der Suche den Status der Anfragen zu überwachen.

Nachfolgend beschreiben wir den Algorithmus, der aus diesen Überlegungen heraus entstanden ist.

5.2 Der Algorithmus

Wie schon beim Constraintsolver beschreiben wir den Algorithmus zuerst für das Grundproblem und geben dann im nächsten Abschnitt an, wie sich die Erweiterungen darin einbetten lassen.

In Abb. 4 ist der Algorithmus in Pseudocode angegeben. Wir verwenden eine Variable `Status`, die zu jeder Anfrage speichert, wie weit die Abarbeitung vorangeschritten ist, d. h. ob der Passagier noch auf den Aufzug *wartet*, sich in der Kabine befindet – also *fährt* – oder ob er schon *am Ziel* ist.

Algorithmus Vorwärtssuche($\mathcal{A}$):
Eingabe: Ein Auftrag $\mathcal{A}$ bestehend aus n Anfragen.
Ausgabe: Alle zulässigen Lösungen.

$\forall i \in \{1, \ldots, n\} : \texttt{Status}_i := \textit{wartet}$
Suche(`StartPos`, 1)

Unterroutine Suche(`NeuePos`, `Tiefe`):

$\texttt{Pos}_{\texttt{Tiefe}} := \texttt{NeuePos}$

Status aktualisieren

$P :=$ Einstiegsposition der *wartenden* Passagiere
    $\cup$ Ausstiegsposition der *fahrenden* Passagiere

if $P = \emptyset$ **then**
    Gebe $\texttt{Pos}_1, \ldots, \texttt{Pos}_{\texttt{Tiefe}}$ als Lösung aus
else
    $\forall p \in P :$ Suche(p, $\texttt{Tiefe} + 1$)

Abbildung 4: Der Vorwärtssuchalgorithmus für das Grundproblem.

Wurde durch die Suche eine neue Position p festgelegt, so kann der `Status` der Anfragen aktualisiert werden, indem

- alle *wartenden* Passagiere mit Einstiegsposition p den Status *fahrend* erhalten (sie steigen zu)
- alle *fahrenden* Passagiere mit Ausstiegsposition p den Status *am Ziel* erhalten (sie steigen aus).

Um zu bestimmen, welche Position als nächstes angefahren werden kann, betrachten wir zu allen *wartenden* Passagieren deren Einstiegsposition und zu allen *fahrenden* Passagieren deren Ausstiegsposition. Ist die so erhaltene Menge P leer, so haben alle Passagiere den Status *am Ziel* und wir haben eine Lösung.

Ansonsten müssen wir für die nächste Position in unserem Plan alle Elemente aus P durchprobieren. Wir spezifizieren hier noch nicht, in welcher Reihenfolge dies geschieht – wir werden darauf in Abschnitt 5.7 eingehen, wo wir diverse Heuristiken untersuchen werden.

5.3 Die Erweiterungen

Für die nachfolgenden Ausführungen ist es vorteilhaft, die Personen, die in der Kabine sind, einfach referenzieren zu können. Wir setzen deshalb

$$F := \{i \in \{1, \dots, n\} : \texttt{Status}_i = \mathit{fahrend}\}.$$

5.3.1 Erweiterungen, die die Folgeposition einschränken

Einige der Erweiterungen schränken die möglichen Folgepositionen ein. Dies sind Zutrittslimitierung, Direktfahrt, Umwegefreiheit und VIP Service.

Bei Zutrittslimitierung geschieht dies wie folgt (dabei ist V_i, wie in 4.3.1 definiert, die Menge der verbotenen Stockwerke für die i-te Anfrage)[10]:

$$P := P \setminus \bigcup_{i \in F} V_i$$

Aus P werden die verbotenen Stockwerke der *fahrenden* Passagiere entfernt.

Sind `NeuePos` und `Tiefe` die Parameter der Unterroutine Suche und ist w_i wie in Unterabschnitt 4.3.4 definiert einer der Werte *irgendwie*, *umwegefrei*, *direkt*

[10] Diese, wie auch die folgenden Anweisungen sind im Algorithmus aus Abb. 4 zwischen die Zeilen 8 und 9 einzufügen.

oder *vip*, so lassen sich die drei Erweiterungen Direktfahrt, Umwegefreiheit und VIP Service so implementieren:

$\forall i \in F$ mit $w_i = \textit{direkt} : P := P \cap \{a_i\}$

$\forall i \in F$ mit $w_i = \textit{umwegefrei}$:
  if $e_i < a_i$ **then**
    $P := P \cap \{\texttt{NeuePos} + 1, \ldots, a_i\}$
  else
    $P := P \cap \{a_i, \ldots, \texttt{NeuePos} - 1\}$

$\forall i \in \{1, \ldots, n\}$ mit $w_i = \textit{vip}$:
  if $(\texttt{Tiefe} = 1 \wedge \texttt{StartPos} \neq e_i)$ **then**
    $P := P \cap \{e_i\}$
  if $(\texttt{Tiefe} = 1 \wedge \texttt{StartPos} = e_i) \vee (\texttt{Tiefe} = 2 \wedge \texttt{StartPos} \neq e_i)$ **then**
    $P := P \cap \{a_i\}$

Für Direktfahrten kann die nächste Position sofort festgesetzt werden. Bei Umwegefreiheit darf die nächste Position nur zwischen der Position, an der sich die Kabine im Moment befindet und der Ausstiegsposition sein. Beim VIP Service steht die zweite bzw. die dritte Position der Kabine fest.

Es kann nun passieren, daß $P = \emptyset$ wird. In diesem Fall wird die Suche ergebnislos abgebrochen und gegebenenfalls an anderer Stelle weitergesucht.

5.3.2 Erweiterungen, die die aktuelle Situation überprüfen

Bei den Erweiterungen gegenseitiger Ausschluß, Begleitung und Kapazitätsbeschränkung schränken wir die nächste Position nicht weiter ein, sondern überprüfen, ob die aktuelle Situation den Anforderungen entspricht.

Dies geschieht wie in Abbildung 5 angegeben (dabei seien G_i die auszuschließenden Anfragen, B_i die Begleitung gebenden Anfragen[11], p_i der Platzbedarf und k die Kapazität der Kabine, wie in 4.3.2, 4.3.3 und 4.3.5 definiert).

Haben zwei Passagiere den Status *fahrend*, die sich gegenseitig ausschließen, so kann die Suche in diesem Ast abgebrochen werden. Das gleiche gilt, wenn ein Passagier in der Kabine ist, der Begleitung benötigt, aber kein begleitender Passagier anwesend ist.

Schließlich wird noch der Gesamtplatzverbrauch der Passagiere in der Kabine berechnet. Ist dieser größer als die Kapazität der Kabine, so kann die Suche ebenfalls abgebrochen werden.

[11] $B_i = \emptyset$ bedeuted: Dieser Passagier benötigt keine Begleitung.

```
Kap := 0
∀i ∈ F :
    ok := false
    ∀j ∈ F :
        if j ∈ G_i then exit
        if j ∈ B_i then ok := true
    if ok = false and B_i ≠ ∅ then exit
    Kap := Kap + p_i
if Kap > k then exit
```

Abbildung 5: Pseudocode für Erweiterungen, die die aktuelle Situation überprüfen. Dieser Code ist in Abb. 4 zwischen die Zeilen 4 und 5 einzufügen.

5.4 Optimierung

Wir optimieren mit *Branch and Bound*. Dazu benötigen wir eine Abschätzung für die Optimierungskriterien (1) und (2). Diese berechnen wir, indem wir die Bedienzeiten der einzelnen Passagiere abschätzen.

Da uns der Anfang des Bedienzeitraums bekannt ist – dies ist der Rufzeitpunkt $T_i^{\odot}$ – benötigen wir lediglich eine Abschätzung für den Ausstiegszeitpunkt $T_i^{\ominus}$. Dazu unterscheiden wir wieder zwischen den drei Statii der Passagiere:

- Ist der Passagier schon *am Ziel*, so kennen wir $T_i^{\ominus}$ und können die Bedienzeit angeben.
- *Fährt* der Passagier, so kann er frühestmöglich aussteigen wenn die Kabine direkt zu dessen Ausstiegsposition fährt. Wir nehmen die Bedienzeit, die in diesem Fall entsteht, als Abschätzung.
- *Wartet* der Passagier noch, so können wir genau wie im letzten Fall vorgehen, allerdings muß diesmal die Kabine zuerst zur Einstiegsposition und dann zur Ausstiegsposition fahren.

Mit den Formeln am Ende von Abschnitt 1.3 erhalten wir für eine optimale Lösung L^* die Abschätzung:

$$T_\Sigma(L^*) \geq \sum_{i=1}^{n} (\tilde{T}_i^{\ominus} - T_i^{\odot}) \quad \text{bzw.} \quad T_\Sigma^2(L^*) \geq \sum_{i=1}^{n} (\tilde{T}_i^{\ominus} - T_i^{\odot})^2 \tag{3}$$

dabei ist

$$\tilde{T}_i^{\ominus} = \begin{cases} T + \alpha(|p - e_i| + |e_i - a_i|) + 2(\beta + \kappa) + \eta & \phantom{\text{falls } \texttt{Status}_i =\ } \textit{wartet} \\ T + \alpha|p - a_i| + \beta + \kappa & \text{falls } \texttt{Status}_i = \textit{fährt} \\ T_i^{\ominus} & \phantom{\text{falls } \texttt{Status}_i =\ } \textit{amZiel} \end{cases}$$

Hierbei ist T der Zeitpunkt nach Abarbeitung des bisherigen Plans und p die Position, an der sich die Kabine dann befindet.

Mit dieser Abschätzung könnte man übrigens eine A^*-Suche implementieren, da es sich um eine *zulässige* Heuristik handelt. Bei unserem Problem ist es aber wichtig, möglichst früh eine Lösung zu finden, weshalb sich die A^*-Suche nicht so gut eignet.

5.5 Beispiel

In Tabelle 3 haben wir nochmal die Anfragen von Beispiel 3 wiedergegeben. Wir sehen uns nun an, wie der Algorithmus dieses Beispiel berechnet. Die Startposition ist 1, die nächste Position muß, da A_3 VIP Service beansprucht, 4 sein (siehe Abb. 6). Nun sind A_7 und A_8 in der Kabine.

Die nächste Position kann nun nur 2, 5, 6 oder 7 sein, da A_1, A_2, A_4, A_5 und A_6 bei dieser Position einsteigen bzw. A_7 und A_8 aussteigen wollen. Position 5 darf aber nicht angefahren werden, weil A_7 in der Kabine ist.

	e	a	V	G	B	w	p
A_1	2	4	–	–	–	*direkt*	1
A_2	2	6	–	–	–	*irgendwie*	1
A_3	1	4	–	–	–	*vip*	2
A_4	7	2	$\{3,4,6\}$	–	–	*irgendwie*	2
A_5	5	3	–	–	$\{2,4\}$	*irgendwie*	1
A_6	6	7	–	$\{1,2,3,4\}$	–	*umwegefrei*	1
A_7	1	7	$\{5\}$	–	–	*irgendwie*	1
A_8	4	5	–	–	–	*irgendwie*	1

Tabelle 3: Anfragen zu Beispiel 3. In den Spalten stehen von links nach rechts: Die Einstiegsposition (e), die Ausstiegsposition (a), die verbotenen Stockwerke (V), die ausgeschlossenen Anfragen (G), Anfragen, die die Begleitung übernehmen können (B), die Art der Beförderung (w) und der Platzbedarf (p). Die Position der Kabine ist $s = 1$ und deren Kapazität beträgt $k = 3$.

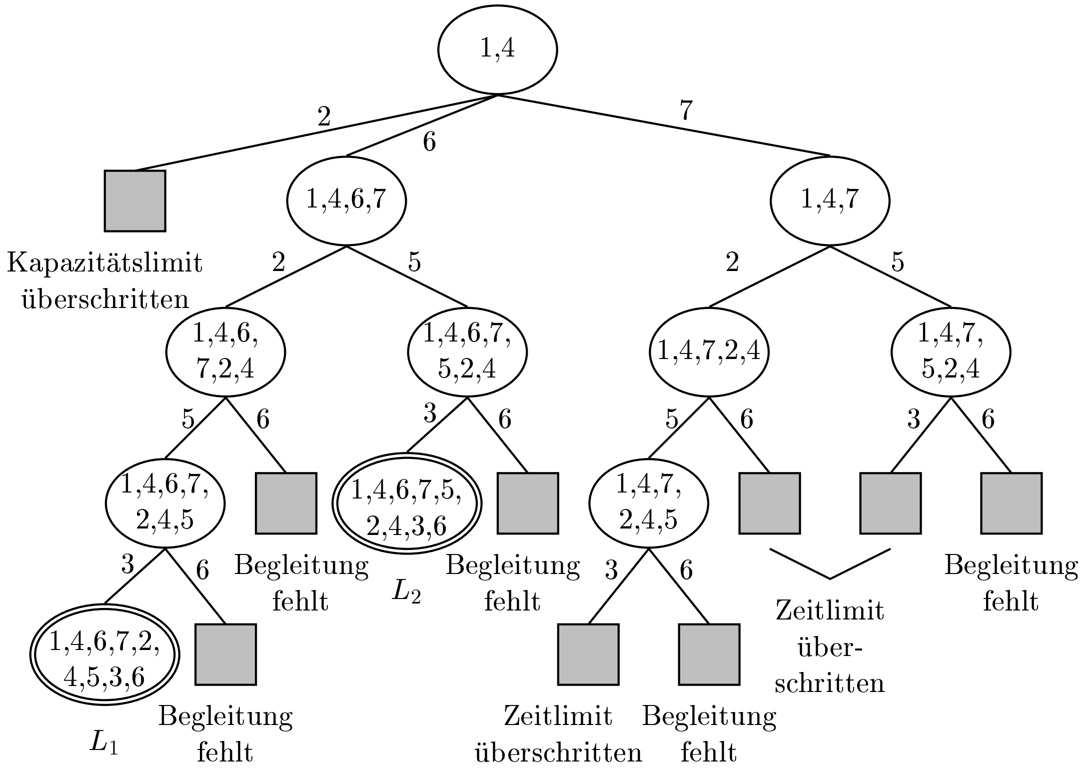


Abbildung 6: Der vollständige Suchbaum, wenn Beispiel 3 mit dem Vorwärtssuchalgorithmus bearbeitet wird.

Nun verzweigt die Suche: Fährt der Aufzug an Position 2, so sind vier Passagiere (A_1, A_2, A_7, A_8) mit einer Gesamtplatzverbrauch von 4 in der Kabine, die Suche kann sofort abgebrochen werden, da nur eine Kapazität von 3 zur Verfügung steht.

Fährt die Kabine Position 6 an, so steigt A_6 zu, und da dieser umwegefrei nach 7 fahren will, ist die nächste Position 7. Dort steigen A_6 und A_7 aus, A_4 steigt zu.

Nun sind nur noch die Positionen 2 und 5 für die nächste Fahrt übrig. Fährt die Kabine an Position 2, so muß die nächste Position 4 sein, da A_1 direkt fahren möchte. Danach kann die nächste Position nur 5 oder 6 sein. Bei 5 besteht noch die Möglichkeit anschließend 3 oder 6 anzufahren. Erstere führt zu der Lösung $L_1 = (1, 4, 6, 7, 2, 4, 5, 3, 6)$ mit $T_\Sigma(L_1) = 242$ bzw. $T_{\Sigma^2}(L_1) = 8868$, zweitere führt zum Abbruch der Suche, da nun A_5 ohne Begleitung in der Kabine ist.

Auf der Suche nach weiteren Lösungen findet der Algorithmus als nächstes $L_2 = (1, 4, 6, 7, 5, 2, 4, 3, 6)$ mit $T_\Sigma(L_2) = 232$ bzw. $T_{\Sigma^2}(L_2) = 8132$. L_2 ist die optimale Lösung, die Zweige zu den restlichen beiden Lösungen werden deshalb wegen Abschätzung (3) vom Algorithmus gestutzt.

5.6 Korrektheit und Vollständigkeit

Satz 5 Der oben beschriebene Algorithmus ist korrekt, d. h. jede Lösung, die dieser ausgibt, ist tatsächlich eine zulässige Lösung, die gegebenenfalls auch die Erweiterungen respektiert, und er ist vollständig, d. h. falls L eine zulässige Lösung ist, so wird diese vom Algorithmus berechnet.

Beweis

Korrektheit: Sei $L = (p_1, \ldots, p_l)$ eine vom Algorithmus berechnete Positionsfolge. Wegen der Zeilen 2 und 3 in Abb. 4 ist $\texttt{Pos}_1 = \texttt{StartPos}$ und somit (i) aus Definition 2 erfüllt. Um die zweite Bedingung zu sehen, sei (e, a) eine Anfrage. Zu Beginn der Suche ist der Status dieser Anfrage *wartend* wegen Zeile 1; wurde eine Lösung gefunden, so ist der Status *amZiel*, da P leer ist (Zeile 6), aber zu jeder Anfrage, die nicht den Status *amZiel* hat, *mindestens* ein Element aus P gehört (Zeile 5). Da Statusübergänge nur von *wartend* zu *fahrend* und von *fahrend* zu *amZiel* möglich sind, und da diese nur auftreten, wenn die Kabine an Ein- und Ausstiegsposition des Passagiers steht, gibt es i und j wie in Definition 2(ii). Dies ist genau die Suchtiefe bei den beiden Statusübergängen.

Wir zeigen nun die Korrektheit der Erweiterungen. Dazu benutzen wir die Notation aus Abschnitt 4.3:

Zutrittslimitierung Sei (e, a, V) eine Anfrage. Wir müssen nun zeigen, daß der zugehörige Passagier die Stockwerke aus V nicht erreichen kann. Ist dieser Passagier in der Kabine, hat also den Status *fahrend*, so wird aber gerade V aus den Folgepositionen entfernt.

Gegenseitiger Ausschluß An jeder Position wird überprüft, ob sich zwei fahrende Passagiere gegenseitig ausschließen.

Begleitung An jeder Position wird überprüft, ob Personen, die eine Begleitung benötigen, nicht alleine in der Kabine sind.

Umwegefreiheit Der Passagier darf nur in Richtung seines Ziels transportiert werden. Die Folgepositionen werden aber gerade auf die Positionen zwischen der aktuellen und der Zielposition eingeschränkt.

Direktfahrt Ist ein Passagier in der Kabine, der Direktfahrt gebucht hat, so ist die Folgeposition die Ausstiegsposition dieses Passagiers.

VIP Service Steigt ein Passagier, der VIP Service beansprucht, an der Startposition zu, so muß die nächste Position die Ausstiegsposition sein. Ansonsten muß die zweite Position die Ein- und die dritte die Ausstiegsposition sein.

Kapazitätsbeschränkung Bei jeder Fahrt wird überprüft, ob die Kapazität überschritten wurde.

Da die Abschätzung für T_Σ bzw. T_{Σ^2} eine untere Schranke sind, liefert *Branch and Bound* eine optimale Lösung. Das diese tatsächlich optimal unter *allen* Lösungen ist, zeigt der folgende Vollständigkeitsbeweis, zusammen mit Satz 2.

Vollständigkeit: Sei $L = (p_1, \ldots, p_l)$ eine zulässige Lösung. Wir zeigen nun durch Induktion, das der Algorithmus L berechnet. Es gilt nach Definition 2: $p_1 = s$.

Sei nun für das Anfangsstück $(p_1, \ldots, p_k)$, $1 \leq k < l$ schon gezeigt, daß der Algorithmus dieses Anfangsstück berechnet. Da L zulässig ist, muß in p_{k+1} jemand zu- oder aussteigen, also eine Anfrage ihren Status wechseln. Nach Bearbeitung der Zeile 5 in Abb. 4 gilt: $p_{k+1} \in P$. Somit bricht der Algorithmus nicht an dieser Stelle ab, da $P \neq \emptyset$. Da $p_{k+1} \in P$, ist aber $(p_1, \ldots, p_{k+1})$ ein Anfangsstück einer Folge, die vom Algorithmus berechnet wird.

Die Erweiterungen Zutrittslimitierung, Umwegefreiheit, Direktfahrt und VIP Service können P einschränken. Ist dadurch $p_{k+1} \notin P$, so ist $(p_1, \ldots, p_{k+1})$ aber auch kein Anfangsstück einer zulässigen Lösung, die diese Einschränkung respektiert.

Eine Lösung, die die Erweiterungen Begleitung, gegenseitiger Ausschluß oder Kapazitätsbeschränkung erfüllt, genügt nicht den Abbruchbedingungen aus Abschnitt 5.3.2, wird also berechnet.

Somit wird L vom Algorithmus berechnet. □

5.7 Heuristiken

Wie schon in Abschnitt 5.2 angekündigt, steuern wir die Suche durch eine Heuristik an. Uns geht es dabei vorallem darum, möglichst früh eine möglichst gute Lösung zu finden. Als Bewertungkriterien benutzen wir dabei:

Wechsel des Status (W) Wir bevorzugen Folgepositionen, bei denen möglichst viele Passagiere ihren Status wechseln, denn die Summe der Bedienzeiten wird kleiner, je früher möglichst viele Passagiere *am Ziel* sind. Dazu müssen sie möglichst früh den Status wechseln.

Einsteigen (E) Wir bevorzugen Folgepositionen, bei denen möglichst viele Passagiere einsteigen, denn wenn möglichst früh möglichst viele Passagiere einsteigen, dann können die, die an die gleiche Position wollen, gleichzeitig aussteigen. (Erst alle einsammeln, dann alle ans Ziel bringen.)

Aussteigen (A) Wir bevorzugen Folgepositionen, bei denen möglichst viele Passagiere aussteigen, denn wie bei W sollen die Passagiere möglichst früh *ans Ziel* kommen, allerdings vermeidet man hier eine Überfüllung der Kabine.

Später einsteigen (S) Wir bevorzugen Folgepositionen, bei denen möglichst wenig Passagiere aussteigen wollen, die noch nicht in der Kabine sind, denn es ist oft vorteilhaft, zuerst diese Passagiere einzuladen, und dann alle Passagiere gemeinsam aussteigen zu lassen.

Distanz (D) Wir bevorzugen Folgepositionen, die näher an der aktuellen Position sind. Vorallem bei langsamen Aufzügen kann die Dauer einer Fahrt eine wichtige Rolle spielen.

Aus diesen Kriterien haben wir die folgenden beiden Heuristiken konstruiert:

> *Wähle denjenigen Halt als nächstes, bei dem möglichst viele Passagiere ihren Status wechseln. Bei Gleichstand bevorzuge den Halt, zu dem die Fahrzeit am kürzesten ist. [SK99]*

Im folgenden bezeichnen wir diese Heuristik als *WD-Heuristik*, da hier die Kriterien W und D in dieser Reihenfolge angewendet werden. Diese Heuristik konzentriert sich darauf, Statuswechsel möglichst früh stattfinden zu lassen.

Weiter haben wir noch folgende Heuristik betrachtet:

> *Wähle denjenigen Halt als nächstes, bei dem möglichst viele Passagiere ihren Status wechseln. Bei Gleichstand bevorzuge den Halt, bei dem möglichst viele Passagiere aussteigen. Ergibt dies immernoch Gleichstand, so wähle denjenigen Halt, zu dem am wenigsten noch wartende Passagiere fahren wollen. Ist auch hier Gleichstand, so wähle den Halt, zu dem der Fahrtweg am kürzesten ist.*

Mit anderen Worten: Wir wenden die Kriterien W, A, S und D in dieser Reihenfolge an. Deshalb bezeichnen wir diese Heuristik als *WASD-Heuristik*.

Von den Kriterien W, E und A sollte eine Heuristik nur maximal zwei benutzen, da das dritte durch diese beiden festgelegt ist, denn die Anzahl der Passagiere, die ihren Status wechseln ist gerade die Summe der Anzahl der Passagiere, die einsteigen und der Anzahl der Passagiere, die aussteigen. Wir haben uns dafür entschieden, E wegzulassen, da dieses Kriterium der Erweiterung Kapazitätslimit entgegenwirkt und nur bei Aufzügen mit sehr großer Kapazität sinnvoll ist.

Die Anordnung „W vor A“ ist sinnvoll, da sonst die Gefahr besteht, daß der Aufzug die Passagiere möglichst einzeln transportiert. Da die Kriterien S und D nur eine untergeordnete Rolle spielen, sollten diese erst beachtet werden, wenn die anderen Kriterien Gleichstand ergeben. Daß wir S vor D angeordnet haben, liegt daran, daß wir bei etlichen Beispielen festgestellt haben, daß D keine wesentliche Beschleunigung der Suche ergab.

6 Vergleich und Performance

In diesem Kapitel werden wir den Constraintsolver mit dem Vorwärtssuchalgorithmus vergleichen und für zweiteren einige Performanceuntersuchungen durchführen. Wir untersuchen dabei nicht nur die Dauer, bis eine (bzw. die optimale) Lösung gefunden wurde, sondern betrachten auch die Qualität der besten Lösung nach einer Sekunde und untersuchen den Einfluß der Heuristiken bei der Vorwärtssuche.

Für diese Untersuchungen betrachten wir drei Beispielklassen:

A) In der Klasse A) sind nur Aufträge, die dem *Grundmodell* (ohne Erweiterungen) entsprechen. Solche Aufträge liefern die größten Suchbäume, da diese durch die Erweiterungen nur verkleinert werden. Andererseits ist hier jedes Blatt im Suchbaum eine zulässige Lösung, so daß dieser durch die Optimierung sehr gut gestutzt werden kann.

B) Interessant sind natürlich auch Aufträge, die *alle Erweiterungen* erlauben. Hier kann es passieren, daß es in einem riesigen Suchbaum nur eine einzige Lösung gibt.

C) In der dritten Beispielklasse untersuchen wir Aufträge, bei denen *alle Anfragen umwegefrei* sind und beachten die Erweiterung *Kapazitätsbeschränkung*. Durch die Umwegefreiheit wird der Suchbaum stark verkleinert, was sich, wie wir in Abschnitt 6.2 sehen werden, positiv auf die Suchdauer auswirkt.

Für die Optimierung benutzen wir in diesem Kapitel ausschließlich T_Σ, da dieses Optimierungskriterium aussagekräftiger ist. Für die Berechnung der Bedienzeiten setzen wir in den Formeln am Ende von Abschnitt 1.3 $\alpha = \beta = \eta = \kappa = 1$.

6.1 Vergleich der beiden Verfahren

Für den Vergleich benutzen wir das am DFKI in Saarbrücken entwickelte Oz [Smo95, Har98], da sich beide Verfahren damit gut implementieren lassen. Leider wird Oz nach einigen Messungen sehr langsam[12] weshalb es nur mit großem Aufwand möglich ist, viele Messungen durchzuführen. Wir beschränken uns deshalb in diesem Abschnitt darauf, aus jeder Beispielklasse nur ein Beispiel durchzurechnen:

[12] Wir vermuten, daß dies daher kommt, daß Oz alle Zwischenergebnisse abspeichert und somit wegen Speichermangel auf die Swap-Partition zurückgreifen muß.

A) Für den Vergleich ohne Erweiterungen benutzen wir den Auftrag aus Beispiel 1: $(6 : 4 \rightarrow 5, 2 \rightarrow 3, 1 \rightarrow 4, 1 \rightarrow 2, 5 \rightarrow 1, 3 \rightarrow 1)$

B) Ein Beispiel, das alle Erweiterungen einschließt, ist der Auftrag aus beispiel 3, den wir hier nochmal wiedergeben:

Die Kabine habe die Kapazität $k = 3$ und stehe an Position $s = 1$. Die acht zu bearbeitenden Anfragen sind in folgener Tabelle zu sehen:

	e	a	V	G	B	w	p
A_1	2	4	–	–	–	*direkt*	1
A_2	2	6	–	–	–	*irgendwie*	1
A_3	1	4	–	–	–	*vip*	2
A_4	7	2	$\{3, 4, 6\}$	–	–	*irgendwie*	2
A_5	5	3	–	–	$\{2, 4\}$	*irgendwie*	1
A_6	6	7	–	$\{1, 2, 3, 4\}$	–	*umwegefrei*	1
A_7	1	7	$\{5\}$	–	–	*irgendwie*	1
A_8	4	5	–	–	–	*irgendwie*	1

C) In der Klasse C) untersuchen wir folgendes Beispiel:

Beispiel 4 Die Kabine habe die Kapazität 3, alle Passagiere benötigen für eine Einheit Platz. Die Startposition ist 14 und die umwegefreien Anfragen lauten: $0 \rightarrow 10$, $3 \rightarrow 15$, $6 \rightarrow 0$, $9 \rightarrow 2$, $12 \rightarrow 6$, $2 \rightarrow 11$.

In Tabelle 4 haben wir die gemessenen Daten zusammengetragen.[13] Dies sind die Dauer, bis die erste bzw. die optimale Lösung gefunden wurde, sowie die Dauer, bis bewiesen wurde, daß die optimale Lösung tatsächlich optimal ist, d. h. die Dauer, bis die Suche beendet wurde. Weiter haben wir angegeben, wieviele Knoten im Suchbaum expandiert werden mußten, um die erste bzw. die optimale Lösung zu finden.

Wir haben hier auch noch die Qualität der besten Lösung nach einer Sekunde angegeben.[14] Die Angabe bezieht sich darauf, um wieviel Prozent T_Σ bei dieser Lösung größer ist, als bei der optimalen. Die Angabe 8% bedeutet also, daß die durchschnittliche Bedienzeit[15] um 8% länger ist, als dies bei der optimalen Lösung der Fall wäre.

[13] Die Messungen wurden auf einer Ultra 1, 176 MHz, vorgenommen.

[14] Achtung: Je kleiner diese Zahl ist, umso besser ist die Lösung.

[15] Eigentlich bezieht sich diese Angabe auf die Summe der Bedienzeiten. Da diese sich von der durchschnittlichen Bedienzeit nur um einen konstanten Faktor unterscheidet, sind beide Aussagen aber äquivalent. Bei der Benutzung von T_{Σ^2} geht diese Beziehung verloren.

	A)		B)		C)	
	CS	VS	CS	VS	CS	VS
1. Lösung	220ms	20ms	450ms	20ms	20,4s	80ms
opt. Lösung	2,7s	520ms	630ms	30ms	147s	640ms
opt. L. m. Bew.	>1h	680ms	>1h	30ms	>1h	650ms
Knoten 1. Lösung	153	5	91	5	9.347	15
Knoten opt. L.	>500.000	476	>500.000	10	>500.000	395
Qualität nach 1s	8%	0%	0%	0%	n.a.	0%

Tabelle 4: Die Meßergebnisse. CS bedeutet Constraintsolver und VS bedeutet Vorwärtssuche.

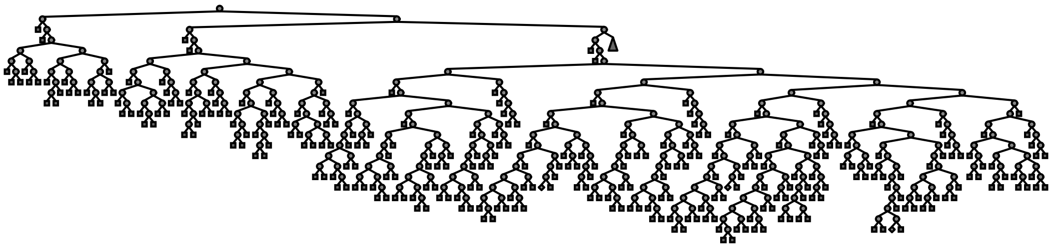


Abbildung 7: Suchbaum des Constraintsolvers zu Beispiel B). Der Teilbaum mit `Länge` ≥ 10 ist durch ein Dreieck symbolisiert.

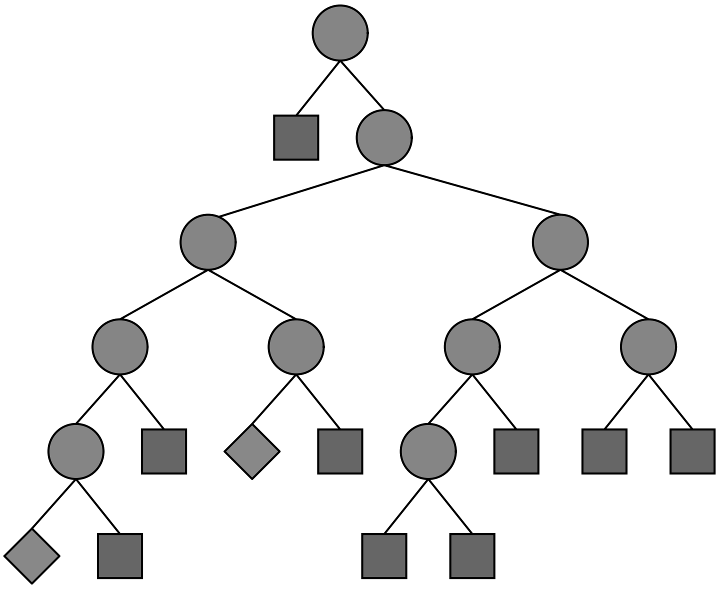


Abbildung 8: Suchbaum der Vorwärtssuche zu Beispiel B). Siehe auch Abbildung 6 auf Seite 33.

Aus den Meßergebnissen ergeben sich folgende Beobachtungen:

- Die Vorwärtssuche ist deutlich schneller als der Constraintsolver. Besonders deutlich wird dies bei Beispiel C, wo der Constraintsolver über zwei Minuten benötigt, um die optimale Lösung zu finden – der Vorwärtssuchalgorithmus erledigt dies in unter einer Sekunde.

- Der Constraintsolver findet zwar relativ schnell die optimale Lösung, benötigt aber eine kleine Ewigkeit, um zu beweisen, daß diese tatsächlich optimal ist. Dies liegt unter anderem daran, daß wir keine Anstrengung unternommen haben, die Zeit, die der Constraintsolver für den Optimalitätsbeweis benötigt, zu verbessern, da wir davon ausgehen, daß die Suche ohnehin nach einer Sekunde abgebrochen wird. Für die Praxis ist der Optimalitätsbeweis somit ohne Bedeutung.

- Die Suchbäume des Constraintsolvers sind sehr viel größer als die der Vorwärtssuche. In den Abbildungen 7 und 8 sind die Suchbäume zu Beispiel B) abgebildet.

6.2 Performance der Vorwärtssuche

Wir haben den Vorwärtssuchalgorithmus in Pascal reimplementiert und damit für die drei Klassen des letzten Abschnitts Performancediagramme berechnet. Für die Messungen haben wir, an jedem Koordinatenschnittpunkt, den Durchschnitt von 50 Beispielen berechnet. Wir haben dazu die WASD-Heuristik benutzt.

6.2.1 Performance des Grundproblems

In Abbildung 9 ist das Performancediagramm für die Klasse A) abgebildet.[16] Man erkennt den für Suchbäume typischen exponentiellen Verlauf. Auf den ersten Blick scheint dies auch dann der Fall zu sein, wenn wir die Anzahl der Anfragen bzw. die Anzahl der Stockwerke fest lassen. Das stimmt aber nicht, denn:

- Läßt man die Anzahl der Stockwerke fest, so gibt es nur eine feste Zahl von verschiedenen Anfragen (bei k Stockwerken sind dies $k(k-1)$). Hat man also soviele Anfragen erreicht, so bleibt die Rechenzeit nahezu konstant, da identische Anfragen kaum zusätzliche Zeit benötigen.

[16] Die Messungen wurden auf einem PowerPC, 180 MHz vorgenommen.

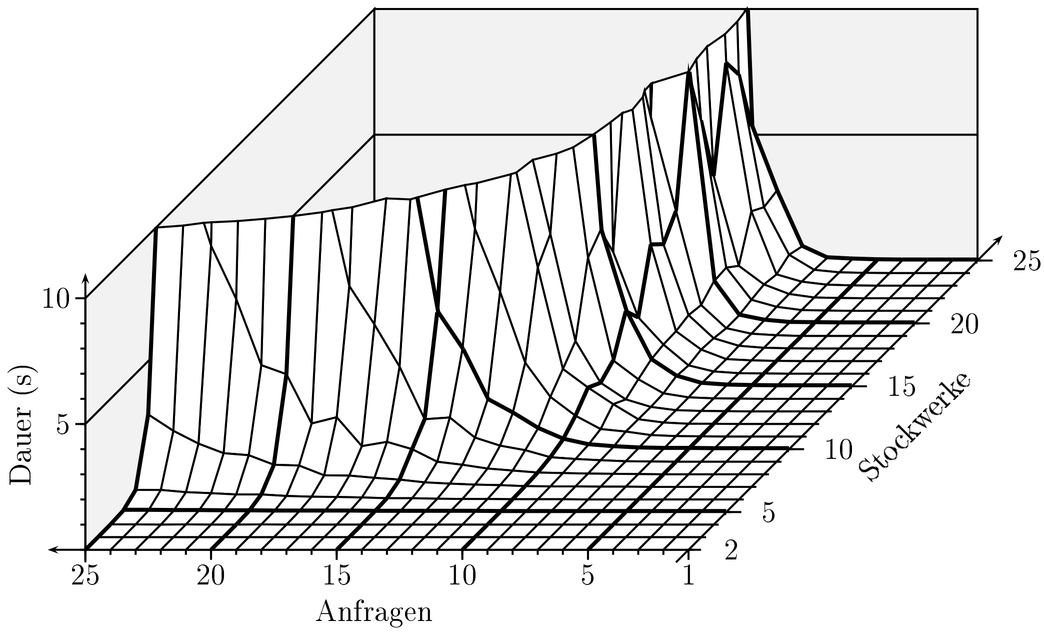


Abbildung 9: Performance-Diagramm für die Klasse A).

- Läßt man die Anzahl der Anfragen fest, so hat man für große Gebäude den Effekt, daß bei fast allen Aufträgen die Ein- und Ausstiegsposition der verschiedenen Anfragen unterschiedlich sind. Mithin ändert sich auch nichts an der Laufzeit, wenn noch einige Stockwerke dazu kommen. Dies kann man in Abbildung 9 erkennen, wenn man die Anzahl der Anfragen auf 10 fest hält. Ab 19 Stockwerken schwankt die Kurve nur noch leicht.

Bei Gebäuden mit bis zu sechs Stockwerken ist der Computer problemlos in der Lage, innerhalb einer Sekunde die optimale Lösung zu berechnen. In Abbildung 10 kann man sehen, daß der Algorithmus auch noch bei größeren Gebäuden akzeptable Ergebnisse liefert. In dieser Abbildung ist die Qualität der besten Lösung nach einer Sekunde abgetragen. Wie schon im letzten Abschnitt beschrieben, gibt der Wert an, um wieviel Prozent T_Σ bei dieser Lösung größer ist, als bei der optimalen – hier allerdings über 50 Beispielen gemittelt.

6.2.2 Alle Erweiterungen

Für die Klasse, in der alle Erweiterungen zugelassen sind, haben wir ebenfalls Performance und Qualität gemessen. Die Ergebnisse sind in den Abbildungen 11 und 12 zu sehen. Dabei sieht jeder Auftrag so aus: Die Kapazität der

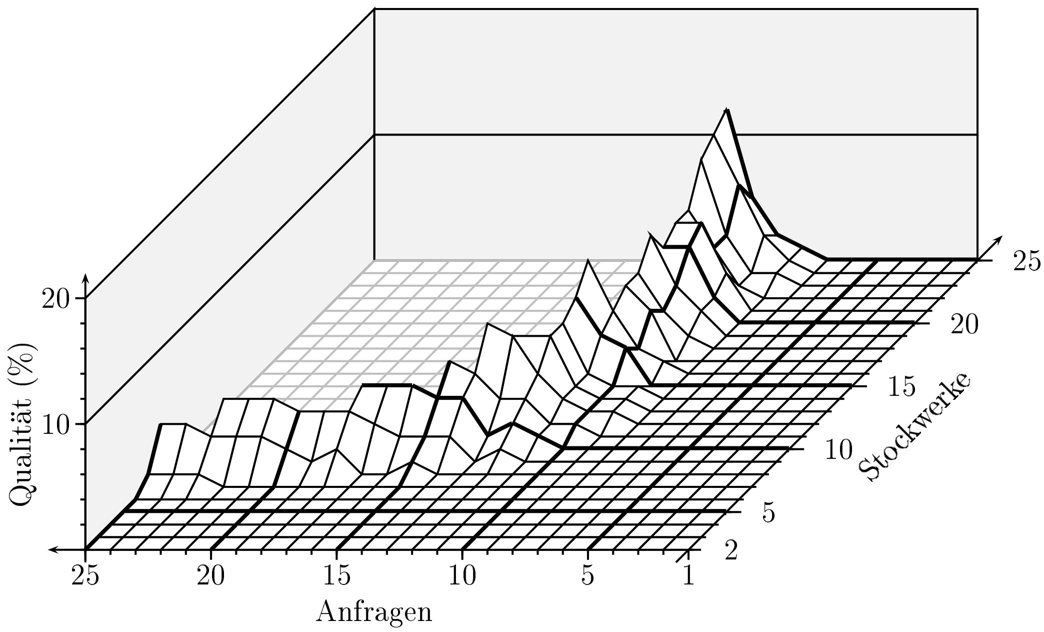


Abbildung 10: Qualität der Lösung nach einer Sekunde mit Beispielen aus der Klasse A).

Kabine ist sechs Einheiten. Jede Anfrage hat mit Wahrscheinlichkeit 1/2 einen Verbrauch von einer Einheit, mit Wahrscheinlichkeit 1/3 einen Verbrauch von zwei Einheiten und mit Wahrscheinlichkeit 1/6 einen Verbrauch von drei Einheiten. Der Passagier darf mit Wahrscheinlichkeit 1/20 ein Stockwerk nicht erreichen. Ebenfalls mit Wahrscheinlichkeit 1/20 darf ein Passagier nicht mit einem anderen bestimmten Passagier gemeinsam befördert werden. Mit der gleichen Wahrscheinlichkeit benötigt ein Passagier Begleitung, die jeder andere Passagier mit Wahrscheinlichkeit 1/2 erbringen kann. 33% der Passagiere sollen *irgendwie* transportiert werden, 55% *monoton* und 11% *direkt*. Das verbleibende Prozent beansprucht *VIP Service*. Für die Diagramme haben wir nur Aufträge zugelassen, die auch eine Lösung besitzen, da schnelles Erkennen der Unlösbarkeit für die Praxis wenig Bedeutung hat.

Vergleicht man diese Abbildungen mit denjenigen aus dem letzten Abschnitt, so fällt auf, daß die Berechnung im Schnitt schneller geworden ist. Allerdings haben wir feststellen müssen, daß einige wenige Beispiele sehr lange für die Berechnung benötigen. Dies liegt daran, daß in diesen Fällen erst sehr spät überhaupt eine Lösung gefunden wird, weshalb die *Branch and Bound*-Strategie nicht zur Beschleunigung genutzt werden kann.

Von den 23.450 berechneten Beispielen konnte der Computer nur in 68 Fällen, das sind etwa 0,29%, nach einer Sekunde keine Lösung angeben, obwohl eine existierte.

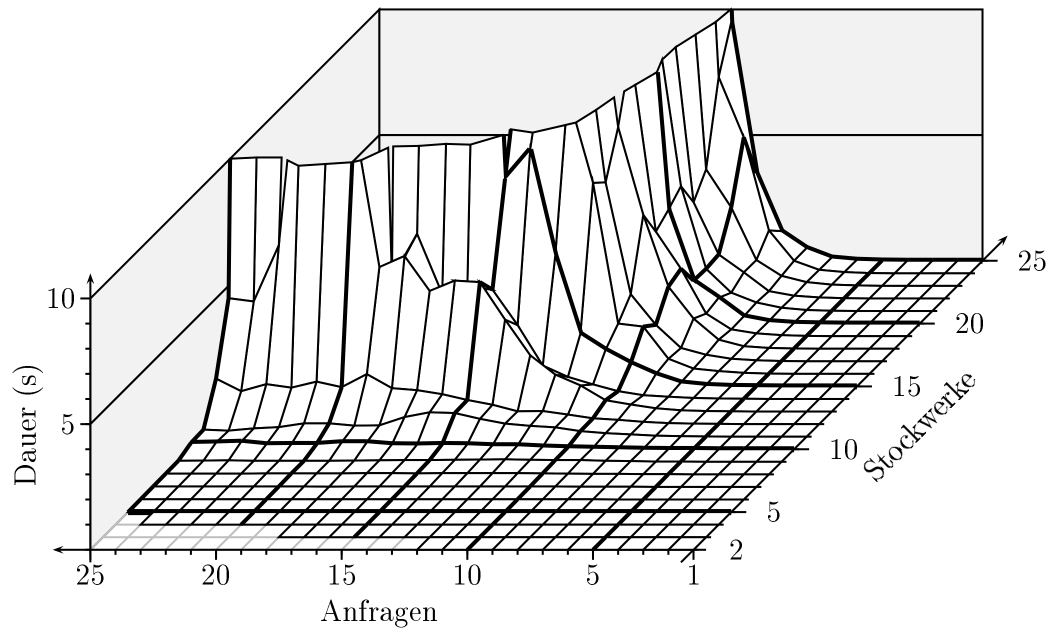


Abbildung 11: Performance-Diagramm für die Klasse B). In diesem wie auch in den folgenden drei Diagrammen gibt es für wenige Stockwerke, aber viele Anfragen keine lösbaren Aufträge, da die Kapazität jedesmal überschritten wird.

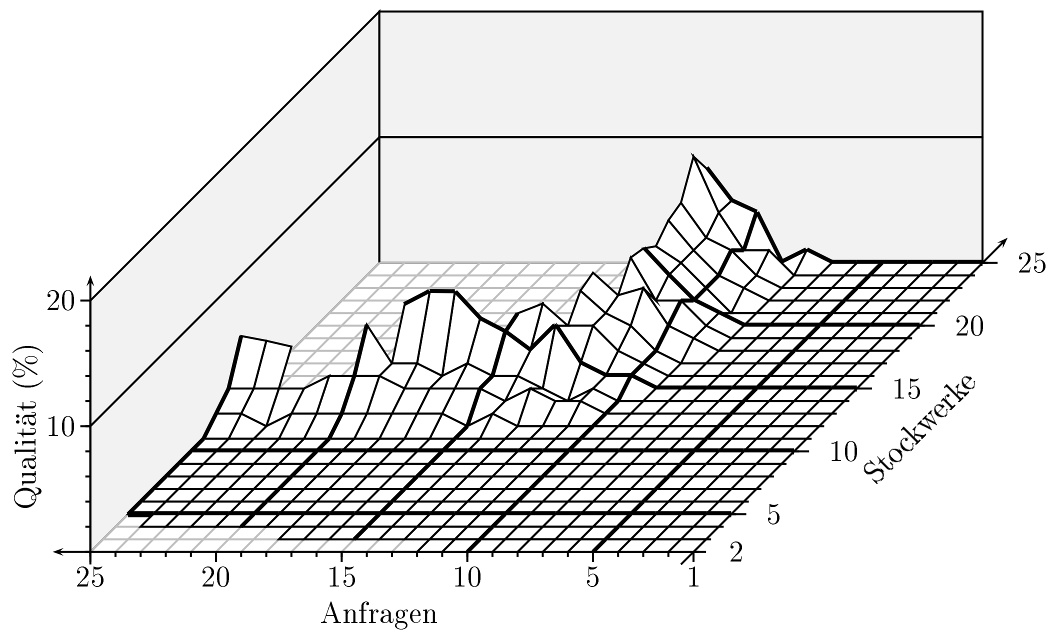


Abbildung 12: Qualität der Lösung nach einer Sekunde mit Beispielen aus der Klasse B).

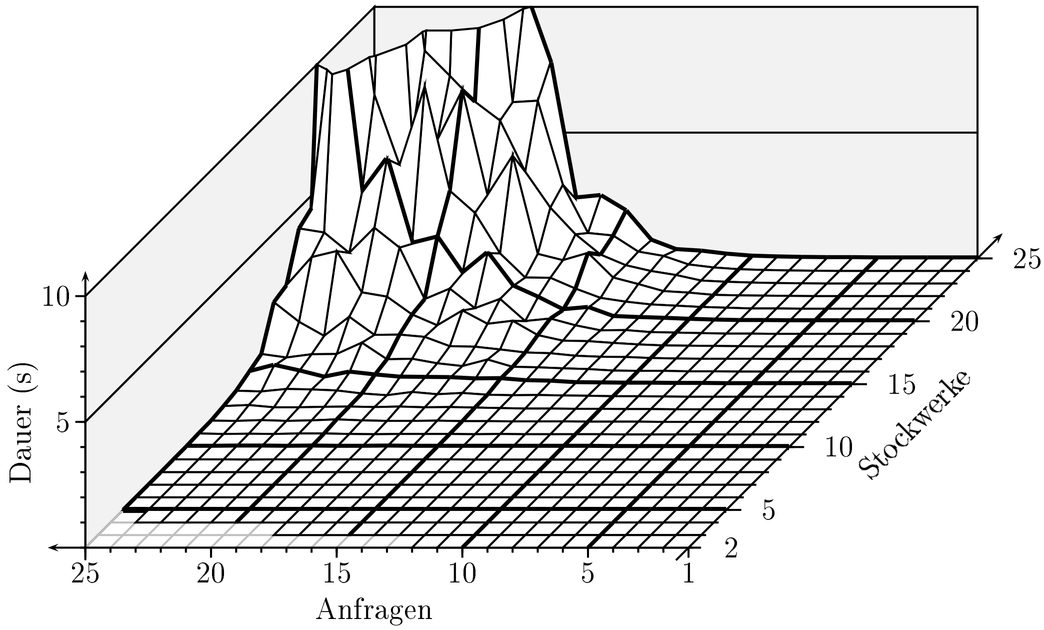


Abbildung 13: Performance-Diagramm für die Klasse C).

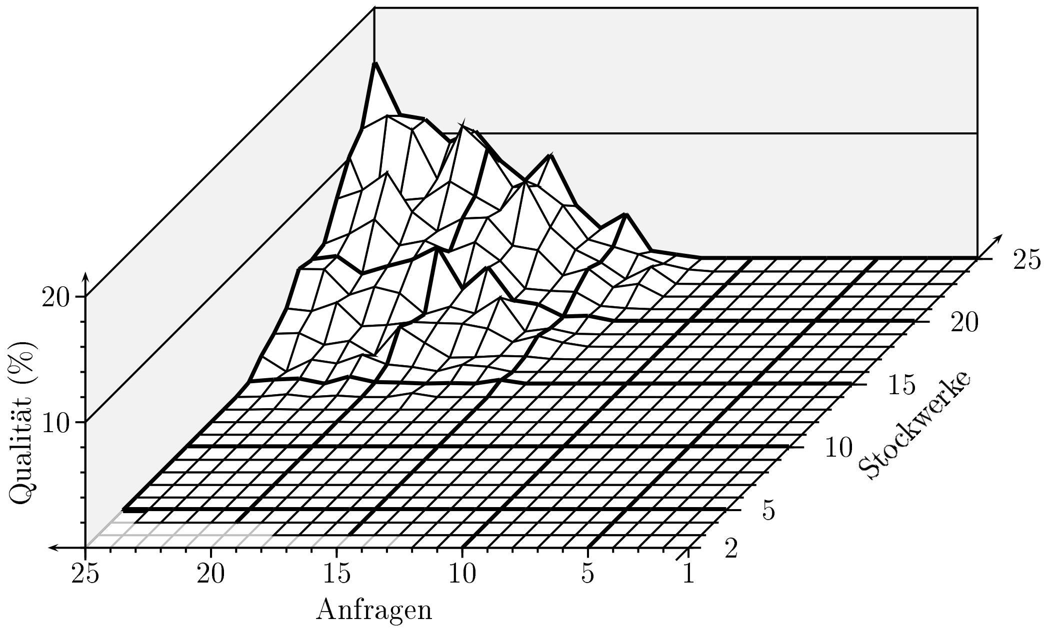


Abbildung 14: Qualität der Lösung nach einer Sekunde mit Beispielen aus der Klasse C).

6.2.3 Umwegefreiheit und Kapazitätsbeschränkung

In der Klasse C) haben wir für die Aufträge eine Kapazität von 6 gewählt, jeder Passagier hat einen Platzverbrauch von 1 und möchte umwegefrei befördert werden. Hier wird der Performancegewinn noch sehr viel deutlicher: Bei bis zu 16 Stockwerken berechnet der Algorithmus noch in unter einer Sekunde die beste Lösung (siehe Abbildung 13). Aber auch dann läßt die Qualität der gelieferten Lösung nur wenig nach (5–10%, siehe Abbildung 14).

Diese starke Beschleunigung wird dadurch erreicht, daß alle Passagiere umwegefrei fahren, denn umwegefreie Anfragen verkleinern den Suchbaum erheblich. Zudem wird hier in der Regel sehr früh[17] eine erste zulässige Lösung gefunden, weshalb durch die Optimierung der Suchbaum nochmal stark eingeschränkt werden kann.

6.2.4 Wie gut sind die Heuristiken?

Um die Heuristiken gut vergleichen zu können, haben wir die Werte auf der Hauptdiagonalen nochmals in zweidimensionalen Diagrammen wiedergegeben. In den Abbildungen 15–17 sieht man, daß mit der WASD-Heuristik eine durchschnittliche Reduktion um nicht ganz 15% erreicht wird. Mit der WD-Heuristik kann man diese Reduktion auf etwa 30% verdoppeln. Diese Reduktion war aber nicht der eigentliche Grund für die Heuristik, da wir daran interessiert sind, möglichst früh eine möglichst gute Lösung zu finden. Aber durch eine gute Lösung, die früh gefunden wurde, erreicht man natürlich auch eine Beschleunigung, da nun die *Branch and Bound*-Strategie den Suchbaum verkleinern kann.

Um die Güte der Heuristiken wirklich zu sehen, betrachten wir die Qualität der besten Lösung nach einer Sekunde. Hier stellen wir für die verschiedenen Klassen ganz unterschiedliches Verhalten fest:

- In der Klasse A) haben beide Heuristiken einen Qualitätsgewinn von etwa 40%. Dabei zeigt die WASD-Heuristik bei großen Werten ein etwas besseres Verhalten, während bei kleineren Werten die WD-Heuristik etwas besser ist (s. Abb. 18).
- In der Klasse B) sind beide Heuristiken etwa gleich gut und zeigen mit etwa 50% Reduktion einen deutlichen Qualitätszuwachs gegenüber der Berechnung ohne Heuristik (s. Abb. 19).

[17] Nur wenn aufgrund der Kapazitätsbeschränkung sehr viele Zweige im Suchbaum erfolglos verlassen werden müssen, kann es vorkommen, daß eine erste Lösung auf sich warten läßt. In den 31.250 durchgerechneten Beispielen war aber kein einziges, bei dem die erste Lösung erst nach einer Sekunde gefunden wurde.

- In der Klasse C) unterscheiden sich die Heuristiken sehr stark. Während die WASD-Heuristik nur eine Verbesserung von 13,2% erreicht, ist die WD-Heuristik hier besonders gut: Bei den berechneten Beispielen nahm die Qualität der Lösung um 62,9% zu (s. Abb. 20). Da diese Klasse realitätsnahe Beispiele beschreibt, empfehlen wir für eine konkrete Umsetzung die WD-Heuristik.

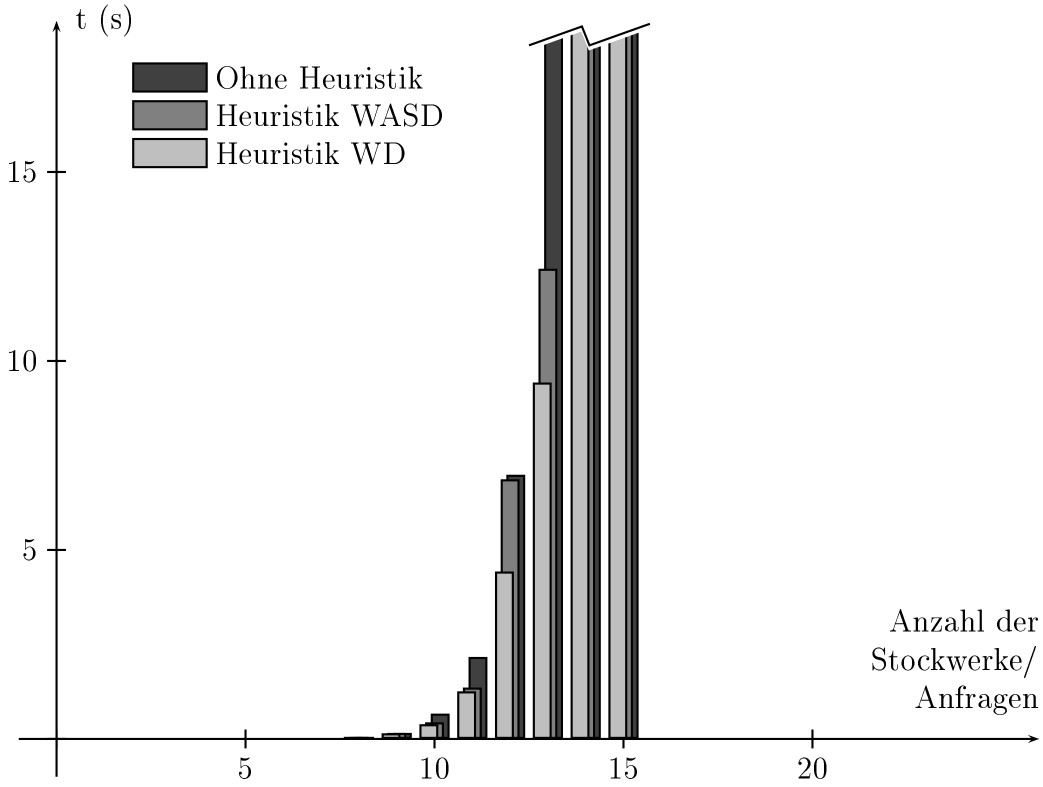


Abbildung 15: Dauer bei Klasse A). Die x-Achse gibt, hier, wie auch bei den folgenden fünf Diagrammen, die Anzahl der Stockwerke *und* die Anzahl der Anfragen an. Die angegebene Zeit ist die Zeit, bis das Optimum mit Beweis gefunden wurde. Mit WASD hat man eine Reduktion um 18,8%, mit WD um 30,4%.

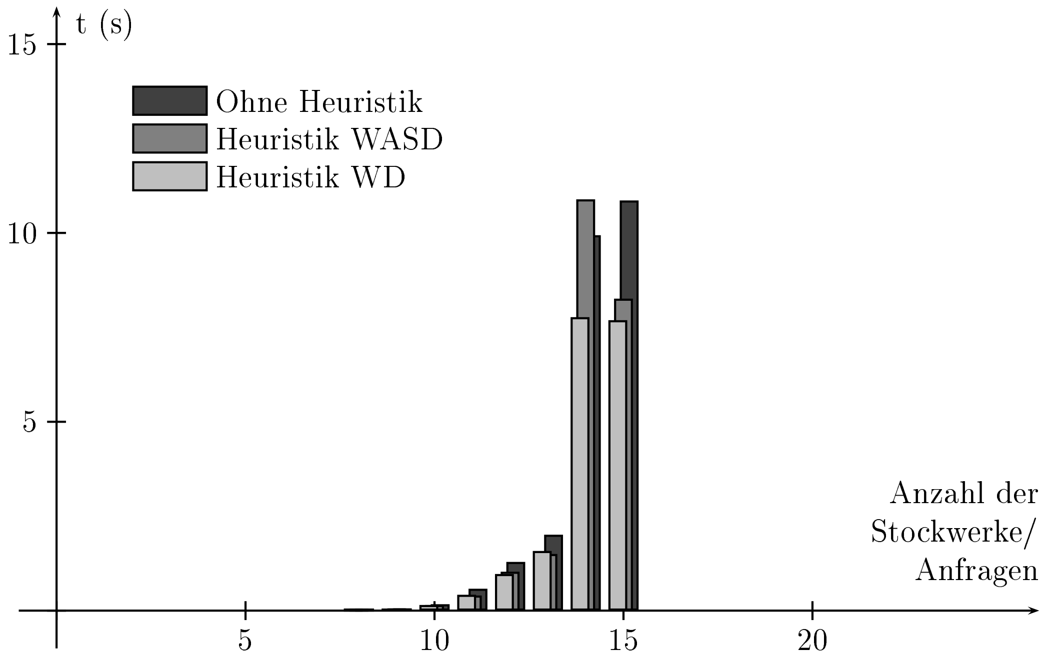


Abbildung 16: Dauer bei Klasse B). Mit WASD hat man eine Reduktion um 10,5%, mit WD um 25,2%.

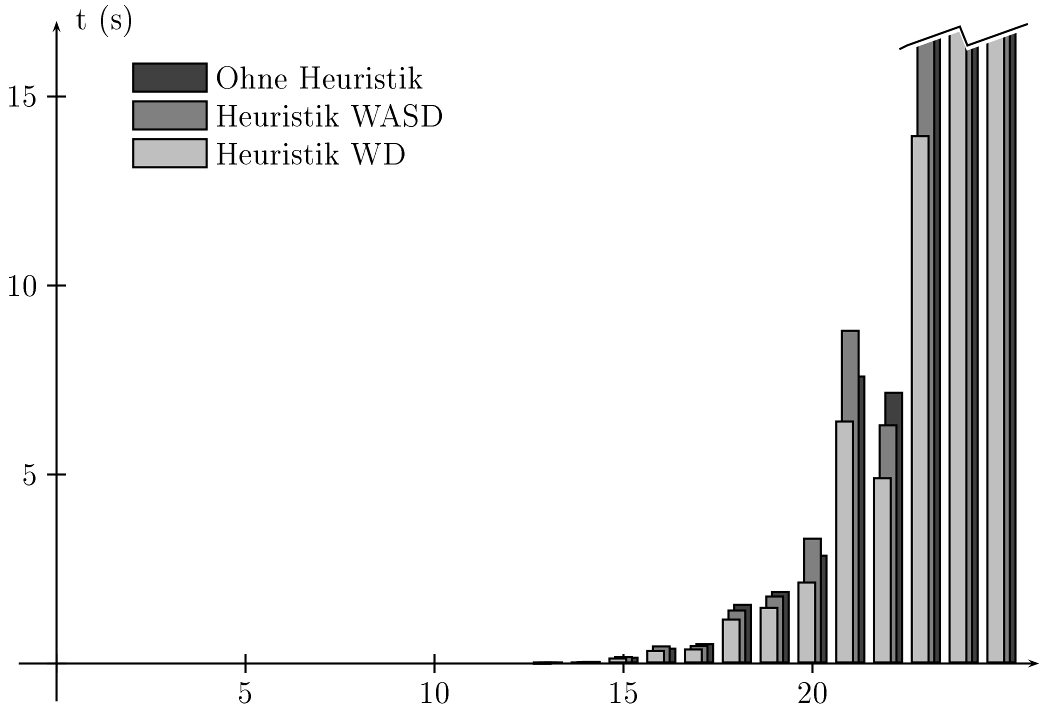


Abbildung 17: Dauer bei Klasse C). Mit WASD hat man eine Reduktion um 12,7%, mit WD um 34,8%.

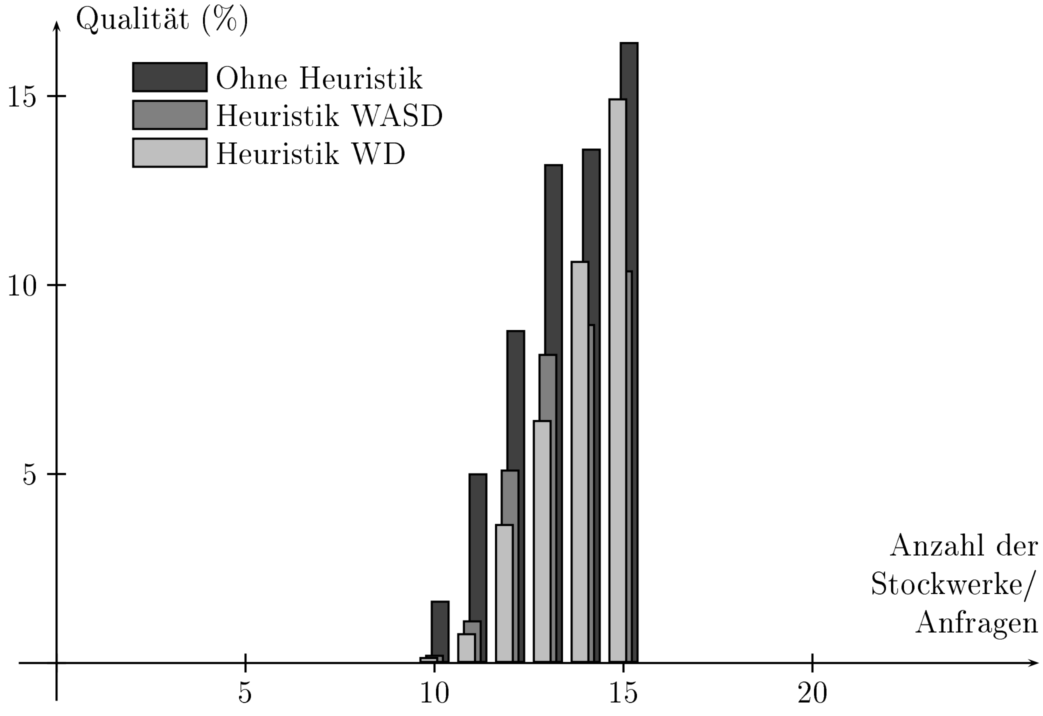


Abbildung 18: Qualität der Lösungen nach einer Sekunde bei Klasse A). Mit WASD erreicht man eine Reduktion um 42,1%, mit WD um 37,6%.

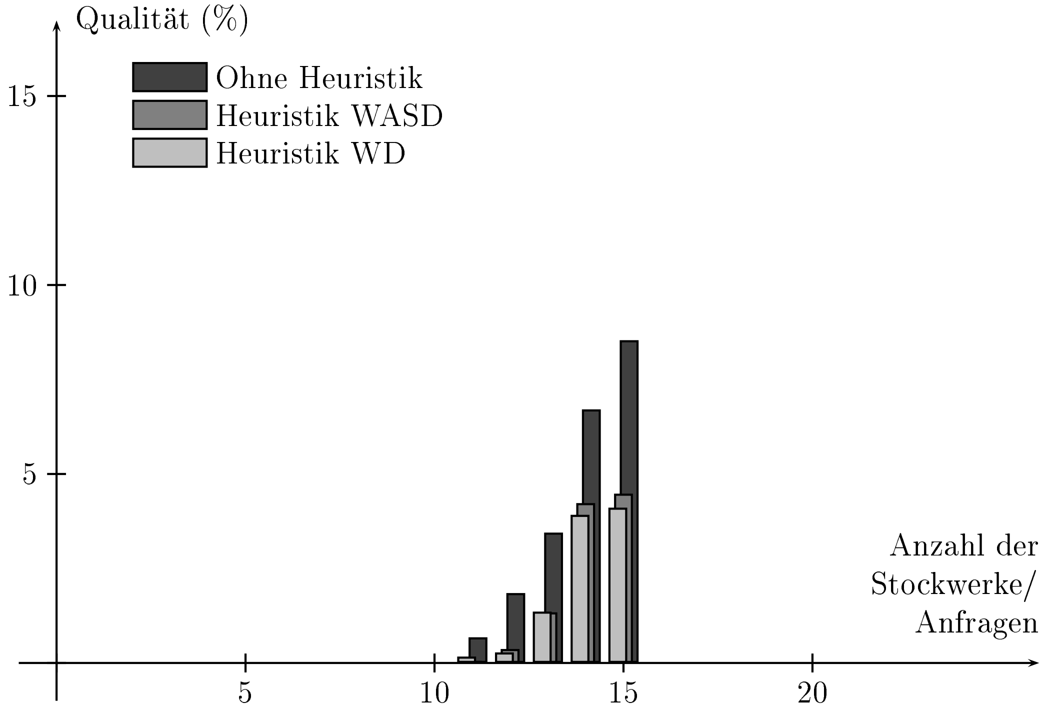


Abbildung 19: Qualität der Lösungen nach einer Sekunde bei Klasse B). Mit WASD erreicht man eine Reduktion um 50,7%, mit WD um 53,7%.

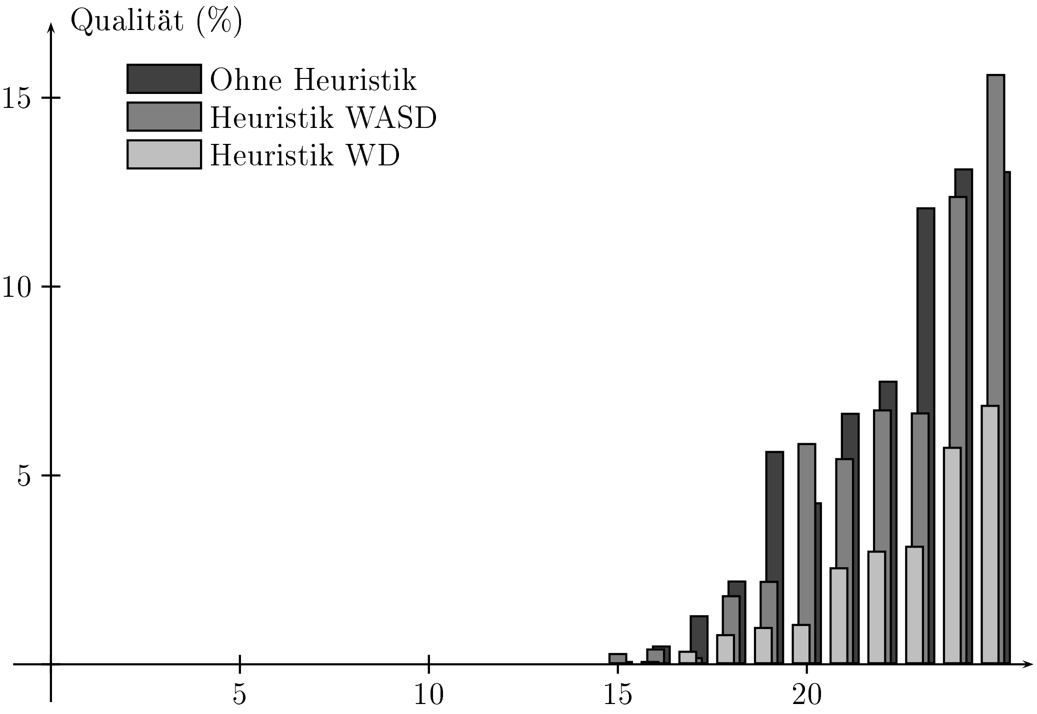


Abbildung 20: Qualität der Lösungen nach einer Sekunde bei Klasse C). Mit WASD erreicht man eine Reduktion um 13,2%, mit WD um 62,9%.

7 Verhalten der Vorwärtssuche im System

7.1 Ein Problem, welches durch Neuplanen entsteht

Dadurch, daß bei jeder neuen Anfrage die Planungseinheit vollständig neu plant, kann es passieren, daß einzelne Passagiere nie ihr Ziel erreichen. Wir sehen uns dazu folgendes Szenario an:

Die Kabine steht in Position 2, gleichzeitig gehen nun zwei Anfragen ein: $9 \to 1$ und $1 \to 3$. Jeweils nach $7 + 18k$ Sekunden, $k \in \mathbb{N}$, erfolgt eine Anfrage $4 \to 2$, jeweils nach $16 + 18k$ Sekunden, $k \in \mathbb{N}$, erfolgt eine Anfrage $1 \to 3$. Es gibt also nach jeweils 9 Sekunden eine neue Anfrage, und diese ist abwechselnd $4 \to 2$ und $1 \to 3$.

In Abbildung 21a) sieht man das Verhalten des Aufzugs mit linearer Optimierung. Der Algorithmus berechnet $L_1 = (2, 1, 3, 9, 1)$ mit $T_\Sigma(L_1) = 38$.[18] Die Lösung $L_1' = (2, 9, 1, 3)$ schneidet mit $T_\Sigma(L_1') = 48$ schlechter ab (siehe Abb. 21c①). Während der Fahrt von Position 1 nach Position 3 wird die neue Anfrage ($4 \to 2$) generiert. Nun lautet der optimale Plan: $L_2 = (3, 4, 2, 9, 1)$ mit $T_\Sigma(L_2) = 50$. Wiederum schneidet die alternative Lösung $L_2' = (3, 9, 4, 2, 1)$ mit $T_\Sigma(L_2') = 59$ schlechter ab (Abb. 21c②). Dies setzt sich so fort, der Passagier auf Stockwerk 9 wird nie abgeholt.

Genauso kann es passieren, daß ein Passagier, der in der Kabine ist, sein Ziel nie erreichen wird. Dieses zweite Problem läßt sich allerdings sehr leicht in den Griff bekommen, indem man fordert, daß alle Anfragen umwegefrei sind. Das ist sowieso besser, da Fahrten in die falsche Richtung die Passagiere verunsichern und Umwegefreiheit auch noch die Berechnung beschleunigt, wie wir im letzten Kapitel gesehen haben.

Das erste Problem läßt sich in den Griff bekommen, indem man zu T_{Σ^2} als Optimierungskriterium übergeht. Um dies zu verstehen, sehen wir uns an, wie zwei Lösungen verglichen werden:

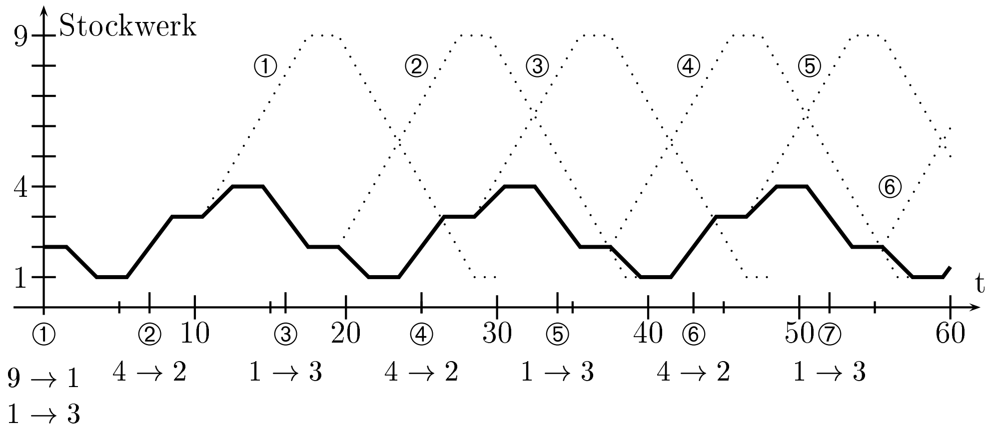
Seien $A_1, \ldots, A_n$ Anfragen mit den Anmeldezeitpunkten $T_1^{\odot} = 0, \ldots, T_{n-1}^{\odot} = 0$ und $T_n^{\odot} = -x$. Der Passagier der n-ten Anfrage wartet also schon seit x Sekunden, die anderen Passagiere haben soeben den Aufzug gerufen. Weiter seien P und Q zwei zulässige Lösungen mit den Ausstiegszeitpunkten $T_{i,P}^{\ominus}$ bzw. $T_{i,Q}^{\ominus}$, $i \in \{1, \ldots, n\}$.

Wir haben

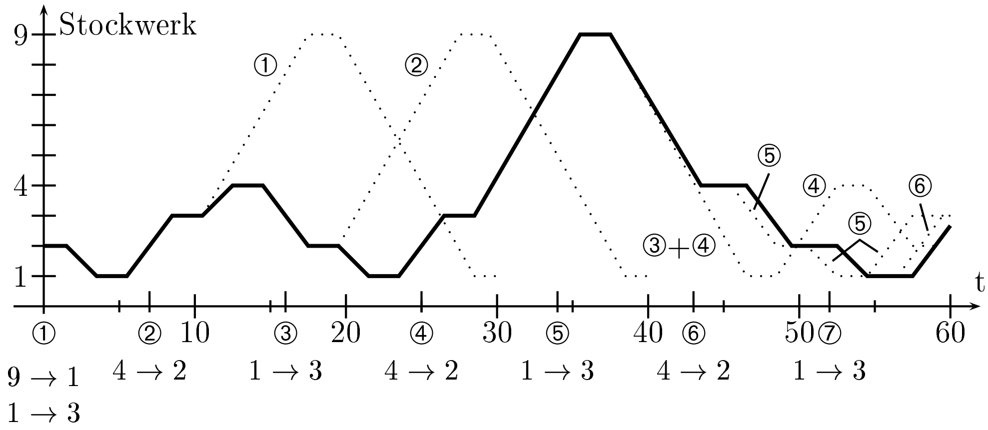
$$T_\Sigma(P) = \sum_{k=1}^{n-1} T_{k,P}^{\ominus} + \left(T_{n,P}^{\ominus} + x\right) \tag{4}$$

[18] Wir setzen $\alpha = \beta = \eta = \kappa = 1$.

a)



b)



c) Die Pläne zu den mit ①–⑥ bezeichneten Zeitpunkten:

①	T_Σ	T_{Σ^2}
(2,1,3,9,1)	38	922
(2,9,1,3)	48	1170
(2,1,9,3,1)	53	1417
(2,1,9,1,3)	57	1637

②	T_Σ	T_{Σ^2}
(3,4,2,9,1)	50	1642
(3,4,9,2,1)	59	1801
(3,9,4,2,1)	59	1801
(3,4,9,1,2)	61	1865
(3,9,4,1,2)	61	1865
(3,9,1,4,2)	62	1930

③	T_Σ	T_{Σ^2}
(2,1,3,9,1)	58	2330
(2,9,1,3)	68	2362
(2,1,9,3,1)	73	2885
(2,1,9,1,3)	77	3025

④	T_Σ	T_{Σ^2}
(3,4,2,9,1)	68	3370
(3,4,9,2,1)	77	3385
(3,9,4,2,1)	77	3385
(3,4,9,1,2)	79	3341
(3,9,4,1,2)	79	3341
(3,9,1,4,2)	80	**3298**

⑤ b)	T_{Σ^2}
(9,4,2,1,3)	1152
(9,4,1,2,3)	1417
(9,4,1,3,2)	1597
(9,1,3,4,2)	1620
(9,1,4,3,2)	1825
(9,1,4,2,3)	1873

⑥ b)	T_{Σ^2}
(4,2,1,3)	1469
(4,1,2,3)	1859
(4,1,3,2)	2149

Abbildung 21: Verhalten des Aufzugs mit a) linearer und b) quadratischer Optimierung. Die Bewegung des Aufzugs ist der dicke Linienzug, die Bewegung, die den einzelnen Plänen entspricht, ist punktiert wiedergegeben. c) Die Pläne samt Bewertung T_Σ bzw. T_{Σ^2}.

$$T_\Sigma(Q) = \sum_{k=1}^{n-1} T^\ominus_{k,Q} + \left(T^\ominus_{n,Q} + x\right) \tag{5}$$

Wollen wir nun wissen, welche der beiden Lösungen besser ist, so müssen wir $T_\Sigma(P)$ und $T_\Sigma(Q)$ vergleichen. Wir erhalten:

$$\begin{aligned}
& T_\Sigma(P) \lesseqqgtr T_\Sigma(Q) \\
\overset{(4),(5)}{\Longleftrightarrow} \quad & \sum_{k=1}^{n-1} T^\ominus_{k,P} + \left(T^\ominus_{n,P} + x\right) \lesseqqgtr \sum_{k=1}^{n-1} T^\ominus_{k,Q} + \left(T^\ominus_{n,Q} + x\right) \\
\overset{-x}{\Longleftrightarrow} \quad & \sum_{k=1}^{n} T^\ominus_{k,P} \lesseqqgtr \sum_{k=1}^{n} T^\ominus_{k,Q}
\end{aligned}$$

Hier sieht man, daß die Wartezeit x, des n-ten Passagiers überhaupt nicht in den Vergleich eingeht.

Betrachten wir nun das gleiche Szenario mit T_{Σ^2} anstelle von T_Σ:

$$\begin{aligned}
T_{\Sigma^2}(P) &= \sum_{k=1}^{n-1} \left(T^\ominus_{k,P}\right)^2 + \left(T^\ominus_{n,P} + x\right)^2 \\
&\overset{Bin.Formel}{=} \sum_{k=1}^{n-1} \left(T^\ominus_{k,P}\right)^2 + \left(T^\ominus_{n,P}\right)^2 + 2T^\ominus_{n,P}x + x^2 \\
&= \sum_{k=1}^{n} \left(T^\ominus_{k,P}\right)^2 + 2T^\ominus_{n,P}x + x^2
\end{aligned} \tag{6}$$

Genauso berechnet man

$$T_{\Sigma^2}(Q) = \sum_{k=1}^{n} \left(T^\ominus_{k,Q}\right)^2 + 2T^\ominus_{n,Q}x + x^2 \tag{7}$$

Vergleich ergibt:

$$\begin{aligned}
& T_{\Sigma^2}(P) \lesseqqgtr T_{\Sigma^2}(Q) \\
\overset{(6),(7)}{\Longleftrightarrow} \quad & \sum_{k=1}^{n} \left(T^\ominus_{k,P}\right)^2 + 2T^\ominus_{n,P}x + x^2 \lesseqqgtr \sum_{k=1}^{n} \left(T^\ominus_{k,Q}\right)^2 + 2T^\ominus_{n,Q}x + x^2 \\
\overset{-x^2}{\Longleftrightarrow} \quad & \sum_{k=1}^{n} \left(T^\ominus_{k,P}\right)^2 + 2T^\ominus_{n,P}x \lesseqqgtr \sum_{k=1}^{n} \left(T^\ominus_{k,Q}\right)^2 + 2T^\ominus_{n,Q}x
\end{aligned}$$

Falls $T^{\ominus}_{n,P} \neq T^{\ominus}_{n,Q}$ wirkt sich hier die Wartezeit x auf den Vergleich aus. Die Lösung, bei der der n-te Passagier Δ Sekunden früher am Ziel ist, erhält einen Bonus von $2x \cdot \Delta$. Ist also x groß genug, d. h. wartet der Passagier schon lange genug, dann wird ein Plan bevorzugt, bei dem dieser Passagier früher bedient wird.

In Abbildung 21b) haben wir wieder das Verhalten des Aufzugs im Szenario vom Anfang dieses Abschnitts wiedergegeben. Diesmal wurde aber mit T_{Σ^2} an Stelle von T_Σ geplant. Dadurch erhält zum Zeitpunkt ④ die Lösung, $L_4 = (3,9,1,4,2)$ mit $T_{\Sigma^2}(L_4) = 3298$ den Vorrang vor $L'_4 = (3,4,2,9,1)$ mit $T_{\Sigma^2}(L'_4) = 3370$ (s. Abb. 21c④). Wie man in Abb. 21b) sieht, wird der Passagier auf Stockwerk 9 abgeholt.

Von den anderen Passagieren erfahren nur die beiden Passagiere, die zu den Zeitpunkten ④ und ⑤ eine Anfrage ausgelöst haben, eine Verzögerung gegenüber der mit T_Σ geplanten Fahrt. Diese ist 14 bzw. 16 Zeiteinheiten.[19]

7.2 Simulation mit mehreren Aufzügen

7.2.1 Die Situation

Wir betrachten ein Haus mit 15 Stockwerken und fünf Aufzügen. Diese sind alle von der gleichen Bauart: Sie bieten sechs Passagieren Platz, wobei wir davon ausgehen, daß alle Passagiere den gleichen Platzverbrauch haben. Um die Simulation realistisch zu gestalten, wollen diese *umwegefrei* fahren. Zudem haben wir in den Zeitberechnungsformeln am Ende von Abschnitt 1.3 $\alpha = \beta = \eta = 2$ Sekunden und $\kappa = 4$ Sekunden gesetzt.

In diesem Gebäude simulierten wir den Aufzugsbetrieb. Um die Belastungsgrenze zu finden, haben wir die Messungen für 1–40 Anfragen pro Minute berechnet. Dabei hielten wir jedesmal den Aufzugbetrieb eine halbe Stunde aufrecht.

Um zu entscheiden, welcher Aufzug den Zuschlag erhält, haben wir $\sqrt{\frac{T_{\Sigma^2}(L_i)}{n_i}}$ als Vergleichskriterium benutzt. Dabei ist L_i die beste Lösung, die für den i-ten Aufzug nach einer Sekunde gefunden wurde, und n_i die Anzahl der Anfragen, die der i-te Aufzug bearbeitet (inkl. der neuen). Dieses Vergleichskriterium haben wir ohne weitere Überlegungen gewählt – es versucht, die durchschnittliche Bediendauer einigermaßen gleich zu verteilen.

[19]Die Anfragen nach dem Zeitpunkt ⑤ werden sogar ein paar Zeiteinheiten schneller bedient, aber dies ist Zufall. Wir hätten auch ein Szenario angeben können, bei dem diese ebenfalls eine leichte Verzögerung in Kauf hätten nehmen müssen.

Wir haben uns nun folgende Fragen gestellt:

- Wie gut sind die Ergebnisse bei quadratischer Optimierung im Vergleich zur linearen Optimierung?
- Benötigt man für jeden Aufzug einen eigenen Computer?

7.2.2 Wie gut sind die Ergebnisse bei quadratischer Optimierung?

In Abbildung 22 haben wir die maximale Bediendauer, die während der 30 Minuten Aufzugsbetrieb auftrat, angegeben. Hier haben wir die Berechnung sowohl mit linearer als auch mit quadratischer Optimierung durchgeführt. Wie man sieht, kann durch quadratische Optimierung die maximale Bedienzeit leicht gesenkt werden. Allerdings ist das Ergebnis dennoch unbefriedigend, da Bedienzeiten von mehr als 5 Minuten inakzeptabel sind.

Für eine Verbesserung könnte etwa das Optimierungskriterium

$$T_{\Sigma\exp}(L) = \sum_{i=1}^{n} \exp(T_i^{\ominus} - T_i^{\odot}) \tag{8}$$

sorgen. Wir sind aber etwas skeptisch, daß sich dieses Problem überhaupt über das Optimierungskriterium in den Griff bekommen läßt. Wenn nämlich

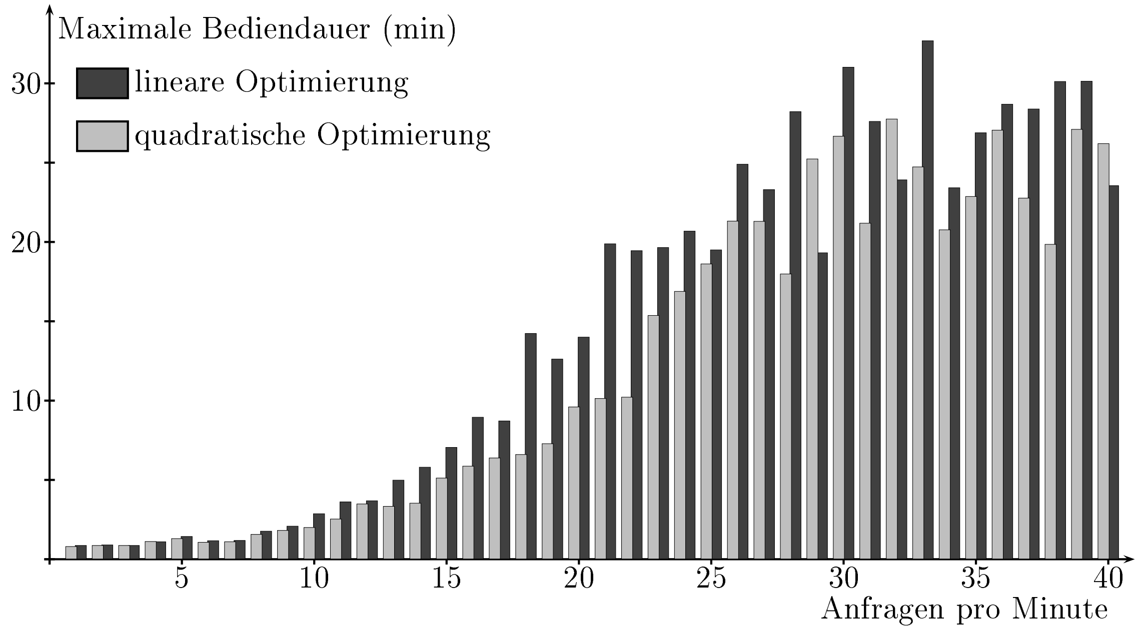


Abbildung 22: Die Maximale Bediendauer im Vergleich bei linearer und quadratischer Optimierung

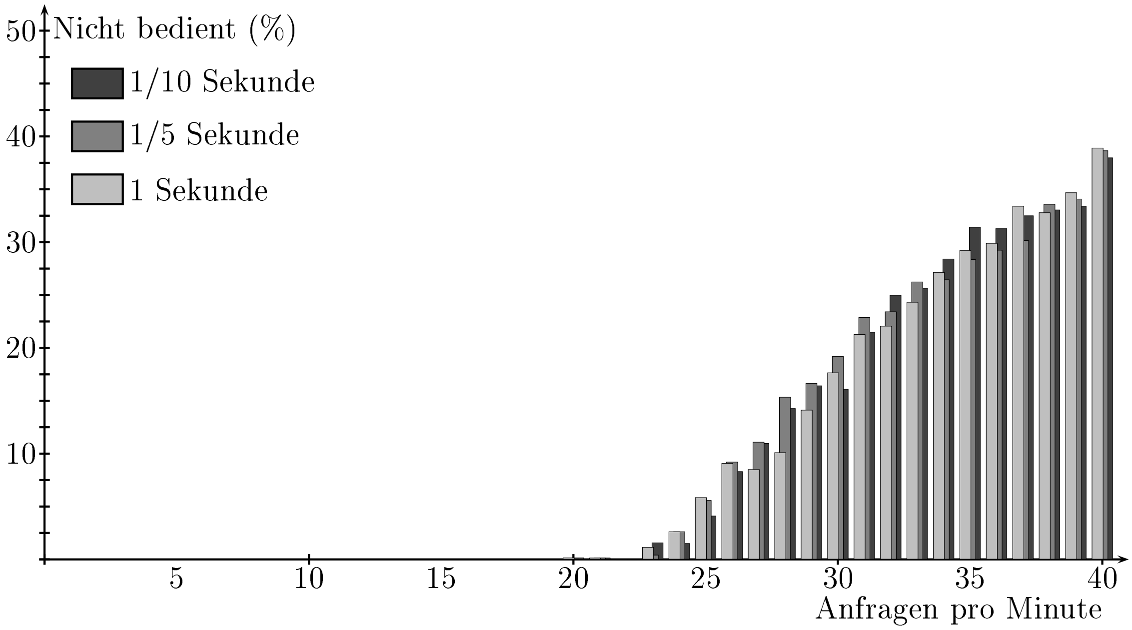


Abbildung 23: Prozentualer Anteil der Anfragen, die durch die Aufzüge nicht bedient werden können, jewails mit 1/10, 1/5 und 1 Sekunde Berechnungszeit pro Auftrag und Aufzug.

zu einem Auftrag nur eine einzige zulässige Lösung existiert, dann hilft das beste Optimierungskriterium nichts.

Wir empfehlen deshalb, entweder durch Variation des Vergleichskriteriums aus Abschnitt 7.2.1 eine Verbesserung zu erreichen (derjenige Aufzug erhält den Zuschlag, bei dem die schlechteste Bediendauer minimal ist). Oder aber man beschränkt künstlich die Bediendauer auf beispielsweise 5 Minuten und fordert, daß zulässige Lösungen zusätzlich die Eigenschaft haben müssen, daß diese Zeit für keinen Passagier überschritten wird.

7.2.3 Benötigt man für jeden Aufzug einen eigenen Computer?

Wir gingen bisher immer davon aus, daß die Planung für alle Aufzüge gleichzeitig vonstatten geht. Dies setzt jedoch voraus, daß für jeden Aufzug ein eigener Computer (oder zumindest ein eigener Prozessor) vorhanden ist. Möchte man mit einem Computer auskommen, so reduziert sich dadurch die Rechenzeit. Wir haben untersucht, welche Auswirkungen dies auf den Aufzugsbetrieb hat.

In Abbildung 23 haben wir angegeben, wieviele der Anfragen abgelehnt wurden, da kein Aufzug diese übernehmen konnte.[20]

[20] Diese Anfragen wurden danach in der Simulation nicht weiter berücksichtigt.

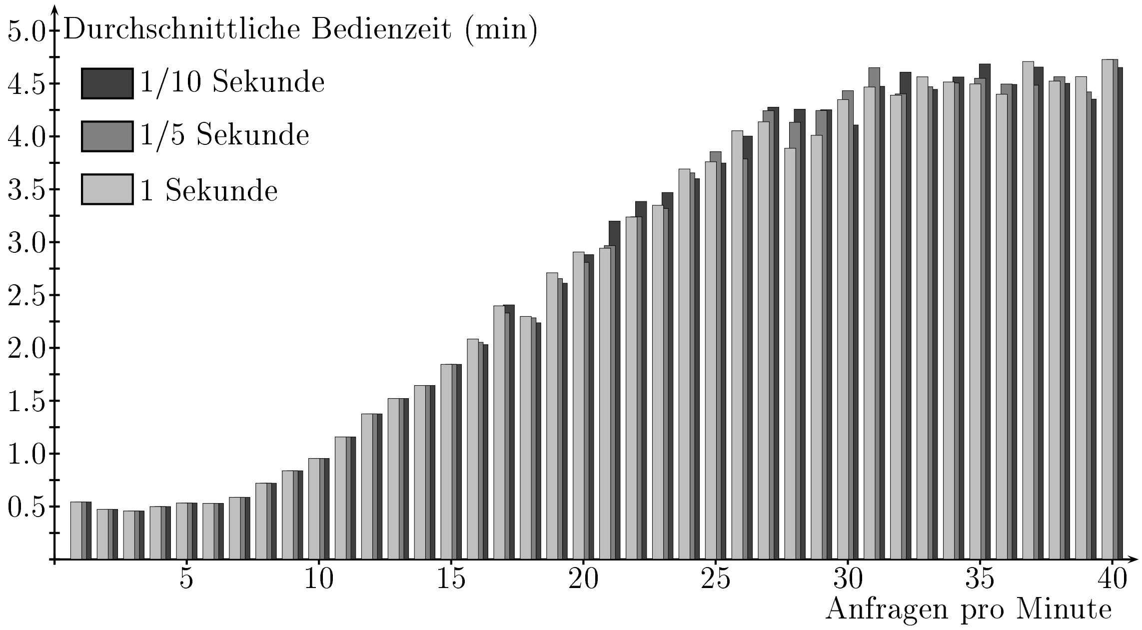


Abbildung 24: Die durchschnittliche Bedienzeit, wiederum jeweils mit 1/10, 1/5 und 1 Sekunde Berechnungszeit pro Auftrag und Aufzug.

Man erkennt deutlich, daß es für die hier untersuchte Aufzugsgruppe keinen wesentlichen Unterschied macht, ob 1/10, 1/5 oder 1 Sekunde für die Planung zur Verfügung gestellt wird.

Zu dem gleichen Resultat kommt man, wenn man die durchschnittliche Bedienzeit betrachtet (s. Abb. 24). Hier sieht man sogar deutlich, daß bei bis zu 15 Anfragen pro Minute die optimale Lösung schon nach 1/10 Sekunde gefunden wurde.

8 Zusammenfassung und Ausblick

8.1 Zusammenfassung

Wir haben gezeigt, daß das Finden einer minimalen Lösung für des Aufzugsproblems NP-vollständig ist. Bei konkreten Lösungsansätzen hat sich die Betrachtung als Schedulingproblem als ungeeignet erwiesen. Ein deklaratives Verfahren mit Constraintprogrammierung hat sich als unzureichend herausgestellt, da sich einige Constraints nicht beschreiben lassen und der resultierende Lösungsprozeß zu lange dauert.

Wir haben deshalb einen spezifischen Suchalgorithmus entwickelt, der gute Resultate liefert, wie wir durch Messung von mehreren 10.000 Beispielen belegen konnten. Zuletzt haben wir noch untersucht, welchen Einfluß das Optimierungskriterium und die für die Suche erlaubte Zeit auf das Ergebnis hat.

8.2 Ausblick

Weitere Untersuchungen sollten sich in erster Linie darauf konzentrieren, welchen Einfluß das Kriterium hat, mit dem entschieden wird, auf welchen Aufzug ein Passagier gebucht wird. Ausgehend von diesen Untersuchungen kann man sich *dann* die Frage stellen, welches Optimierungskriterium das geeignetste ist.

Auch eine weitergehende Untersuchung der Heuristiken sehen wir als lohnend an. Hier kann es sinnvoll sein, diese jeweils an die betrachte Problemklasse anzupassen.

Desweiteren dürfte es interessant sein, die Komplexität von verschiedenen Problemklassen zu bestimmen. Insbesondere sei hier auf die Klasse verwiesen, in der alle Passagiere *umwegefrei* fahren und die Kapazität der Kabine beschränkt ist. Wir können uns vorstellen, daß für diese Klasse (evtl. mit einigen zusätzlichen Einschränkungen) zur Berechnung der optimalen Lösung (oder guter Approximationen) polynomiale Algorithmen existieren, da durch die Umwegefreiheit viele Teile des resultierenden Plans schon festgelegt bzw. auf wenige Fälle beschränkt sind.

A Literaturverzeichnis

[AB93] A. Aggoun und N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Journal of Mathematical and Computer Modelling*, 17(7):57–73, 1993.

[BESW94] J. Blazewicz, K.H. Ecker, G. Schmidt, und J. Węglarz. *Scheduling in Computer and Manufacturing Systems*. Springer-Verlag, Berlin, Heidelberg, 2. Auflage, 1994.

[ENSS95] Guy Even, Joseph Narr, Barud Schieber, und Madka Sudan. Approximating minimum feedback sets and multi-cuts in directed graphs. In Egon Balas (Herausgeber), *4th International Conference on Integer Programming and Combinatorial Optimization, Copenhagen*, Band 920 der Reihe *Lecture Notes in Computer Science*, 14–28. Springer-Verlag, Berlin, Heidelberg, 1995.

[Har98] Seif Haridi. Tutorial of Oz 2. DFKI Oz Documentation Series, September 1998. `http://www.ps.uni-sb.de/ns3/oz2/documentation`.

[Kan92] Viggo Kann. *On the Approximability of NP-complete Optimization Problems*. Doktorarbeit, Department of Numerical Analysis and Computer Science, Royal Institute of Technology, Stockholm, 1992.

[Kar72] R.M. Karp. Reducibility among combinatorial problems. In R.E. Miller und J.W. Thatcher (Herausgeber), *Complexity of Computer Computations*, 85–103. Plenum Press, New York, 1972.

[OW90] Thomas Ottman und Peter Widmayer. *Algorithmen und Datenstrukturen*. BI-Wissenschaftsverlag, Mannheim, 1990.

[Pug98] Jean-Francois Puget. Constraint programming: A great AI success. In *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI'98)*, 1998.

[Sey95] P.D. Seymour. Packing directed circuits fractionally. *Combinatorica*, 15(2):281–288, 1995.

[Sha79] Adi Shamir. A linear time algorithm for finding minimum cutsets in reducible graphs. *Siam Journal on Computing*, 8(4):645–655, November 1979. MIT-Report No. MIT/LCS/TM-85.

[SK99] Bernhard Seckinger und Jana Köhler. Online-Synthese von Aufzugssteuerungen als Planungsproblem. In Frank Puppe (Herausgeber), *XPS-99 5th German Conference on Knowledge-Based Systems*, 127–134, Würzburg, März 1999.

[Smo95] Gerd Smolka. The Oz programming model. In J. van Leeuwen (Herausgeber), *Computer Science Today*, Band 1000 der Reihe *Lecture Notes in Computer Science*, 324–343. Springer-Verlag, Berlin, Heidelberg, 1995.

[SSW98] Christian Schulte, Gert Smolka, und Jörg Würtz. Finite domain constraint programming in Oz. DFKI Oz Documentation Series, Februar 1998. `http://www.ps.uni-sb.de/ns3/oz2/documentation`.

[Wür98] Jörg Würtz. *Lösen kombinatorischer Probleme mit Constraintprogrammierung in Oz*. Doktorarbeit, Technische Fakultät der Universität des Saarlandes, Saarbrücken, Januar 1998.