Lösungskonzepte für praxisbezogene Instanzen des PSPACE-vollständigen Stacking-Problems

Nach F.G. König, M.E. Lübbecke, R.H. Möhring, G. Schäfer, I. Spenke

Seminar Algorithm Engineering, Sommersemester 2008

Tobias Langner

Albert-Ludwigs-Universität Freiburg
langneto@informatik.uni-freiburg.de

1 Einleitung

Die zugrundeliegende Veröffentlichung von König *et al.* [1] beschäftigt sich mit dem Stacking-Problem, einem Problem aus der Stahlindustrie. Im Kern geht es um die Optimierung der Umlagerungsprozesse, die dadurch entstehen, dass die Erzeugungs- und Weiterverarbeitungsreihenfolgen der Stahlrohlinge sich häufig unterscheiden und somit eine Zwischenlagerung dieser unumgänglich macht. Durch den in natürlicher Weise beschränkt vorhandenen Platz kann es notwendig sein, die Stahlobjekte innerhalb der Zwischenlager umzusortieren, um die gewünschte Weiterverarbeitungsreihenfolge zu ermöglichen. Diese Umlagerungen kosten Zeit und Geld, weshalb sich König *et al.* damit beschäftigen, wie man ihre Anzahl möglichst gering halten kann. Ein grundlegendes Problem stellt hierbei allerdings die Tatsache dar, dass das Stacking-Problem PSPACE-vollständig ist und man damit aller Voraussicht nach keinen effizienten Algorithmus finden wird, der eine optimale Lösung liefert. Aus diesem Grund präsentieren die Autoren einen Greedy-Algorithmus, der mittels geführter Graphensuche im Zustandsraum in kurzer Zeit für praktische Anwendungen hinreichend gute Lösungen berechnet.

2 Problemdefinition

Das Stacking-Problem besteht, ignoriert man einige weniger wichtige Details, aus den folgenden Komponenten:

Eingehende Objekte. Die Menge $I := \{1, \ldots, n\}$ stellt die Menge der eingehenden Objekte dar. Maximal m dieser Objekte können gleichzeitig ankommen.

Pufferstapel. Die Menge $\mathcal{S} := \{S_1, \ldots, S_k\}$ stellt die Menge der k verschiedenen Pufferstapel dar. Die Funktion $h : \mathcal{S} \mapsto \mathbb{N}$ ordnet jedem Pufferstapel S eine Höhe $h(S)$ zu. Eine Zuordnung von Objekten zu Stapeln nennt man *Konfiguration*. Die Startkonfiguration C_0 darf bereits Objekte enthalten. Die Gesamtmenge V der zu bearbeitenden Objekten setzt sich damit zusammen aus der Menge der eingehenden Objekte I und der Menge J der in C_0 bereits zugeordneten Elemente, $V := I \uplus J$.

Stapelbedingungen. Die Objekte aus V dürfen nicht beliebig aufeinander gestapelt werden. Zur Modellierung dieser Restriktion wird ein *Konfliktgraph* verwendet. Sei dazu $G := (V, A)$ ein gerichteter Graph, der genau dann eine Kante $(i, j) \in A$ enthält, wenn

das Objekt i nicht direkt auf j liegen darf. Dies erlaubt uns nun, den Begriff *gültige Konfiguration* zu definieren, als eine Konfiguration, in der ein Objekt i nur genau dann direkt auf j liegt, wenn gilt $(i, j) \notin A$ und weiterhin auf keinem Stapel S mehr als $h(S)$ Objekte liegen.

Zielstapel. Die Menge $\mathcal{T} := \{T_1, \ldots, T_\ell\}$ stellt die Menge der Zielstapel dar. Jedes Objekt $i \in V$ ist genau einem Zielstapel $t(i) \in \mathcal{T}$ zugeordnet, welcher weiterhin eine Reihenfolge definiert, in der die ihm zugeordneten Objekte eingesammelt werden sollen. Damit Objekte für einen bestimmten Zielstapel eingesammelt werden dürfen, muss dieser Zielstapel zunächst aktiv sein. Gleichzeitig dürfen aber nur maximal w Zielstapel *aktiv* sein. Erst wenn für einen aktiven Zielstapel alle zugeordneten Objekte eingesammelt wurden, darf der Zielstapel entfernt werden und stattdessen ein anderer Stapel aktiviert werden. Die Reihenfolge, in der Zielstapel entfernt werden dürfen, ist durch eine partielle Ordnung $\prec_{\mathcal{T}}$ auf $\mathcal{T}$ eingeschränkt: Gilt $T_1 \prec_{\mathcal{T}} T_2$ für zwei Zielstapel, so darf T_2 erst entfernt werden, nachdem T_1 entfernt wurde.

Bewegungen. Um Objekte zwischen den einzelnen Orten zu bewegen, gibt es vier Möglichkeiten. Ein Objekt kann (a) von einem Eingang auf einen Pufferstapel, (b) von einem Eingang direkt zu einem Zielstapel, (c) von einem Pufferstapel zu einem anderen Pufferstapel und (d) von einem Pufferstapel zu einem Zielstapel bewegt werden. Eine Bewegung ist *gültig*, wenn beim Ausführen alle der zuvor definierten Bedingungen erfüllt bleiben.

Ziel. Eine Instanz der Erfüllbarkeitsvariante des Stacking-Problems zu lösen bedeutet, eine Sequenz von gültigen Bewegungen zu finden, so dass alle Zielstapel in der korrekten Reihenfolge erzeugt werden können. In der Optimierungsvariante geht es darum, eine Sequenz mit möglichst geringer Zahl an Bewegungen zu finden.

Die wesentlichen Elemente der Problemdefinition sind zur Veranschaulichung in Abbildung 1 grafisch dargestellt.

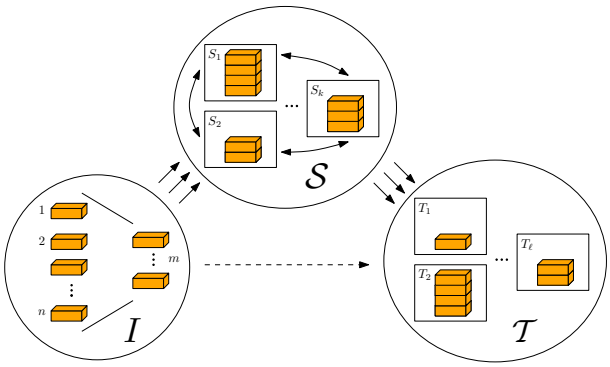

Abbildung 1: Illustration der Definition des Stacking-Problems.

3 Berechnungskomplexität

Nachdem das Stacking-Problem auf ein formales Fundament gestellt wurde, wird der Fokus in diesem Abschnitt auf die Berechnungskomplexität gerichtet. Die dabei

hauptsächlich betrachtete Komplexitätsklasse ist die Klasse der Probleme, die auf einer deterministischen Turingmaschine mit polynomiellem Speicher entschieden werden können, genannt PSPACE. Die entsprechende Klasse für nichtdeterministische Turingmaschinen heißt NPSPACE. Savitch [2] wies 1970 nach, dass ein auf einer nichtdeterministischen Turingmaschine mit einer bestimmten Speicherkomplexität $f(n) \geq \log(n)$ lösbares Problem auf einer deterministischen Turingmaschine nur eine quadratisch höhere Platzkomplexität hat. Eine Konsequenz dieses Satzes ist die Äquivalenz der Klassen PSPACE und NPSPACE. Diese Tatsache wird im Beweis des ersten Thereoms verwendet.

Theorem 1. *Die Entscheidbarkeitsvariante des* Stacking-Problems *gehört zur Klasse PSPACE.*

Beweis. Jede Konfiguration lässt sich mit polynomiellem Speicherbedarf repräsentieren. Alle möglichen Bewegungen können in polynomieller Zeit (und damit auch mit polynomiellem Speicherbedarf) bestimmt werden. Zur Lösung des Problems kann man nun also eine nichtdeterministische Suche mit polynomiellem Speicherbedarf durchführen, in dem nichtdeterministisch eine der möglichen Bewegungen gewählt wird und man jeweils nur den aktuellen Zustand vorhält. Nach Savitchs Theorem ist dies auch auf einer *deterministischen* Turingmaschine mit polynomiellem Speicherbedarf möglich. □

Das soeben bewiesene Theorem gibt uns nun also eine obere Schranke für die Komplexität des Problems. Im Weiteren werden wir auch eine untere Schranke für das Problem ableiten, indem wir zeigen, dass es zu den schwierigsten Problemen in PSPACE gehört und damit PSPACE-vollständig ist. Der Beweis verläuft analog zu den üblichen Vollständigkeitsbeweisen für Probleme in NP, nämlich mittels polynomieller Reduktion.

Das Stacking-Problem wird auf das Problem Configuration-To-Edge reduziert, welches wie folgt definiert ist. Gegeben ist ein ungerichteter Graph $\hat{G} = (\hat{N}, \hat{E})$ mit nicht-negativen Kantengenwichten und ganzzahligen Eingangsflussgrenzwerten für jeden Knoten. Eine *gültige Konfiguration* von $\hat{G}$ ist eine Orientierung der Kanten, bei der die Summe der Gewichte der eingehenden Kanten eines jeden Knoten mindestens dessen Eingangsflussgrenzwert entspricht. Man gelangt von einer Konfiguration zu einer anderen, indem man genau eine Kante umkehrt, und dabei sicherstellt, dass alle Eingangsflussgrenzwerte erfüllt bleiben. Das Configuration-To-Edge-Problem fragt nun, ob eine Sequenz von Kantenumkehrungen existiert, so dass man eine bestimmte Kante umkehren kann. Es ist selbst dann PSPACE-vollständig, wenn $\hat{G}$ planar ist und nur aus den beiden Knotentypen in Abbildung 2 besteht.

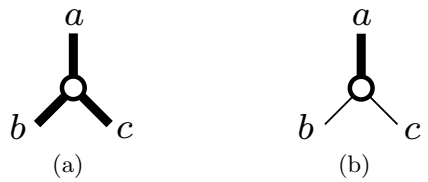

Abbildung 2: Illustration eines OR- (a) und eines AND-Knotens (b). Fett dargestellte Kanten haben das Gewicht 2, normal dargestelle das Gewicht 1, die Eingangsflussgrenzwerte betragen für beide Knotentypen 2.

Theorem 2. *Die Entscheidbarkeitsvariante des* Stacking-Problems *ist auch in einer vereinfachten Form PSPACE-vollständig, wenn es keine eingehenden Objekte gibt, jeder Pufferstapel höchstens die Höhe drei hat und nur ein einziges Objekt an einem Zielstapel angefordert ist.*

Beweis. Wie schon bereits angedeutet verläuft der Beweis per polynomieller Reduktion von Configuration-To-Edge mit einem AND/OR-Graphen $\hat{G} = (\hat{N}, \hat{E})$ auf das Stacking-Problem. Hierfür erzeugen wir zunächst für jeden Knoten $n \in \hat{N}$ zwei Stapel S_n^1, S_n^2 und für jede Kante $e \in \hat{E}$ ein Objekt. Die Kante e ist genau dann ausgehend von einem Knoten n, wenn das Objekt e auf S_n^1 oder S_n^2 liegt.

Die Stapel haben, abhängig ob sie zu einem AND- oder einem OR-Knoten gehören unterschiedliche „Eigenschaften“. Betrachten wir zunächst einen OR-Knoten $n \in \hat{N}$. Durch das Platzieren von Dummy-Objekten auf S_n^1 und S_n^2 können wir mittels des Konfliktgraphen modellieren, dass nur die Objekte a, b, c auf S_n^1 und S_n^2 gelegt werden können. Legt man eine Höhe von zwei für beide Stapel fest, so können höchstens zwei der Objekte a, b, c auf die Stapel gelegt werden. Mindestens eine Kante muss also eingehend sein, d. h. für alle Stapelkonfigurationen ist die zugehörige Kantenkonfiguration gültig.

Für einen AND-Knoten n gibt es zusätzlich zu den drei Objekten a, b, c noch zwei Hilfsobjekte h_b und h_c (die jeweils doppelt für beide durch b und c verbundenen Knoten existieren). Erneut modellieren wir mit Hilfe von Dummy-Objekten die folgenden Bedingungen. Die Hilfsobjekte h_b und h_c können nur direkt aufeinander und auf dem Grund von S_n^1 und S_n^2 platziert werden. Objekt a kann nur auf dem Grund der beiden Stapel, b nur auf h_b und c nur auf h_c platziert werden. Mit einer Höhe von drei für beide Stapel wird sichergestellt, dass a nur ausgehend von n sein kann, wenn b und c gleichzeitig eingehend sind. Auch alle anderen Stapelkonfigurationen entsprechen gültigen Kantenkonfigurationen.

Das Problem, ob eine bestimmte Kante e im Graphen $\hat{G}$ umgekehrt werden kann, lässt sich nun auf die Frage reduzieren, ob das Dummy-Objekt unterhalb von e in der Startkonfiguration einen Zielstapel erreichen kann. Dazu muss nämlich zunächst e aus dem Weg geräumt werden, was der Umkehrung der zugehörigen Kante entspricht. □

Die entscheidende Eigenschaft, die dem Stacking-Problem diese Komplexität verleiht, ist, dass unter Umständen beliebig komplexe Umstapelungen von Nöten sein können, um ein bestimmtes Objekt zu bewegen. Dies ist dann der Fall, wenn der Konfliktgraph relativ viele Kanten enthält. Die Autoren konnten allerdings zeigen, dass das Problem, selbst wenn es keinerlei Konflikte zwischen Objekten gibt (und weitere Vereinfachungen durchgeführt werden), immer noch sehr komplex, nämlich NP-hart ist [1].

4 Geführte Graphensuche

Im vorherigen Abschnitt wurde nachgewiesen, dass es relativ unwahrscheinlich ist, dass es für das Stacking-Problem einen effizienten und optimalen Algorithmus gibt. Aus diesem Grunde entwickelten König *et al.* einen approximativen Ansatz mittels geführter Suche im Zustandsgraphen der Probleminstanz. Für einen jeden Zustand erzeugt man einen Knoten und verbindet ihn mit allen Knoten, deren Zustand durch eine gültige Bewegung erreichbar ist. Ein Zustand $\Sigma := (C, t, A, \Omega)$ selbst besteht aus der Pufferstapelkonfiguration C, einem Zeitstempel t, der Menge A der Elemente, die bereits aus

dem Eingang entnommen wurden und der Menge Ω der momentan aktiven Zielstapel. Da schon allein die Menge der verschiedenen Pufferkonfigurationen gigantisch ist, ist es unmöglich, den gesamten Zustandsgraphen aufzubauen. Aus diesem Grund verwendet man bei der geführten Graphensuche eine *Zustandsbewertungsfunktion*, die uns – basierend auf möglichst „intelligenten“ Kriterien – sagt, welcher der benachbarten Zustände am vielversprechendsten ist.

Entscheidend für die Güte des Suchergebnisses ist offensichtlich die Wahl der Zustandsbewertungsfunktion, die im Weiteren *val* genannt wird. Aus diesem Grunde widmet sich der Rest dieses Abschnitts der Entwicklung einer Bewertungsfunktion, die speziell auf das STACKING-PROBLEM angepasst ist. Dazu ist es zunächst einmal notwendig, überhaupt ein Kriterium zu finden, mit dem sich verschiedene Zustände miteinander vergleichen lassen. Als intuitives Kriterium schlagen die Autoren die Anzahl der sog. *falschen Positionen* vor. Ein Objekt liegt dann in einer falschen Position, wenn ein anderes Objekt, dass sich auf dem selben Stapel unterhalb befindet, vorher eingesammelt werden muss.

Um diese Idee zu formalisieren, erweitert man die partielle Ordnung $\prec_{\mathcal{T}}$ für die Entfernungsreihenfolge der Zielstapel (unter Berücksichtigung der Zeitfenster der Objekte) zu einer Totalordnung $<_{\mathcal{T}}$. Im Folgenden gelte $T_1 <_{\mathcal{T}} \ldots <_{\mathcal{T}} T_\ell$.

Basierend auf dieser Totalordnung definiert man nun eine partielle Ordnung $\prec$ auf den Objekten, die im Prinzip nach der Einsammelungsreihenfolge ordnet. Für zwei Objekte i, j gilt $i \prec j$ wenn

1. $t(i) <_{\mathcal{T}} t(j)$ und nicht beide Zielstapel sind momentan gleichzeitig aktiv

oder

2. $t(i) = t(j)$ und i wird auf dem Zielstapel vor j angefordert.

Die Anzahl der falschen Positionen für Objekt i auf dem Pufferstapel S ist dann

$$\mathit{false}(i, S) := |\{j \in S : i \prec j \wedge (j \text{ liegt in } S \text{ oberhalb von } i)\}|$$

Zustände mit wenigen falschen Positionen sind offensichtlich erstrebenswert, die Bewertungsfunktion für die Elemente eines Zielstapels T_t ergibt sich daher als

$$\mathit{val}_t(\Sigma) := \sum_{\ell=1}^{k} \sum_{i \in S_\ell \cap T_t} \mathit{false}(i, S_\ell) \ .$$

Die Bewertungsfunktion für den gesamten Zustand unter Betrachtung aller noch zu komplettierenden Zielstapel $T_\alpha, T_{\alpha+1}, \ldots$ ist dann

$$\mathit{val}(\Sigma) := \sum_{t=\alpha}^{\ell} d^{t-\alpha} \cdot \mathit{val}_t(\Sigma) \ ,$$

wobei $d \in (0, 1]$ eine Konstante ist, die angibt, wie schnell das Gewicht der Bewertung für Zielstapel abnimmt, die erst später komplettiert werden müssen.

Fügt man das Kriterium hinzu, dass Zustände, in denen ein Objekt zu einem Zielstapel bewegt wird, immer zu bevorzugen sind, kann man nun mit dieser Bewertungsfunktion von einem Zustand aus den „interessantesten“ Nachbarzustand finden und dort rekursiv verfahren. Dieser Ansatz klingt relativ simpel, dennoch produziert er für praktisch relevante Instanzen erstaunlich gute Ergebnisse, wie im nächsten Abschnitt gezeigt wird.

5 Experimentelle Ergebnisse

Aus praktischer Sicht gibt es für die Bewertung des soeben vorgestellten Verfahrens zwei Szenarien, die von besonderer Bedeutung sind. Dies sind einerseits sog. *Hot-Buffer*-Instanzen mit wenigen Pufferstapeln und engen Zeitfenstern, andererseits *Cold-Buffer*-Instanzen mit vielen Pufferstapeln, aber keinen eingehenden Objekten. Den Autoren standen jeweils Originaldaten aus der Industrie für die Durchführung der Tests zur Verfügung.

Zur Beurteilung der Qualität der Ergebnisse ist es hilfreich, absolute untere Schranken für die Anzahl der Bewegungen zu haben. Für die *Cold-Buffer*-Instanzen erfüllt die Anzahl der falschen Positionen der Anfangskonfiguration diesen Zweck recht gut, für die *Hot-Buffer*-Instanzen ist sie aber wenig aussagekräftig. Hierfür schlagen König *et al.* vor, das ursprüngliche Problem einer Lineare-Programmierung-Relaxierung zu unterwerfen, bei welcher man die Anzahl der Pufferstapel größer als die Anzahl der Elemente wählt (und somit jeglicher Art von Konflikt aus dem Weg geht) und als Ziel versucht, die Anzahl der direkten Bewegungen zu maximieren. Für das entstehende MIP lässt sich dann – mit immer noch beträchtlichem Aufwand – mittels LP-Solvern eine ganzzahlige Lösung bestimmen. Trotz der erheblichen Vereinfachung stellt die so errechnete Lösung doch eine brauchbare untere Schranke für das ursprüngliche Problem dar.

Für die *Hot-Buffer*-Instanzen beträgt die betrachtete Zeitspanne 12 Stunden. Es gibt 3 parallele Eingänge, maximal 3 gleichzeitig aktive Zielstapel und 14 bis 16 Pufferstapel. Die Laufzeit des Algorithmus beträgt weniger als 0,1 Sekunden pro berechnete Bewegung. Die bestimmte Anzahl der Bewegungen wird in Abbildung 3 mit der unteren Schranke aus der Lösung des LPs verglichen und liegt maximal 12% darüber.

Auf der Seite der *Cold-Buffer*-Instanzen gibt es pro Testlauf 50 Pufferstapel sowie maximal 5 gleichzeitig aktive Zielstapel. Die Ergebnisse in Abbildung 4 zeigen, dass die berechnete Lösung maximal 25% oberhalb der unteren Schranke – gegeben durch die falschen Positionen – liegt.

6 Diskussion

Der von den Autoren vorgestellte Ansatz für die Lösung des Stacking-Problems erzielt, angewendet auf praktische Instanzen, Ergebnisse, die nur um maximal 25% schlechter sind als theoretische berechnete untere Schranken vorgeben. In Anbetracht der Tatsache, dass das Problem PSPACE-vollständig ist, ist dies ein durchaus respektables Ergebnis.

Der Vergleich mit empirischen Daten aus der Industrie lässt vermuten, dass durch die Anwendung des präsentierten Algorithmus viel Zeit und Geld gespart werden kann. Mittels einem Prototyp des Algorithmus, der in zwei Stahlbetrieben getestet wurde, konnte diese Annahme auch schon in der Praxis verifiziert werden.

Literatur

1. König, F.G., Lübbecke, M.E., Möhring, R.H., Schäfer, G., Spenke, I.: Solutions to real-world instances of PSPACE-complete stacking. In Arge, L., Hoffmann, M., Welzl, E., eds.: ESA. Volume 4698 of Lecture Notes in Computer Science., Springer (2007) 729–740
2. Savitch, W.J.: Relationships between nondeterministic and deterministic tape complexities. Journal of Computer and System Sciences **4**(2) (1970) 177–192

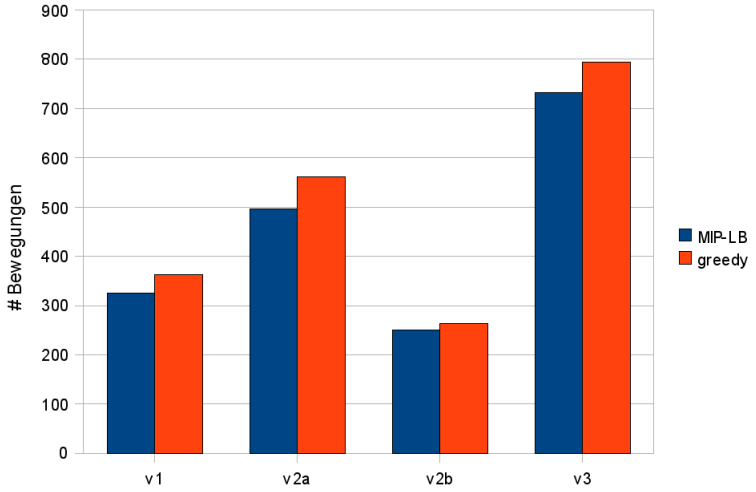

Abbildung 3: Vergleich der Anzahl der Bewegungen der Lösung des Greedy-Algorithmus mit unterer Schranke aus der LP-Relaxierung für verschiedene *Hot-Buffer*-Instanzen.

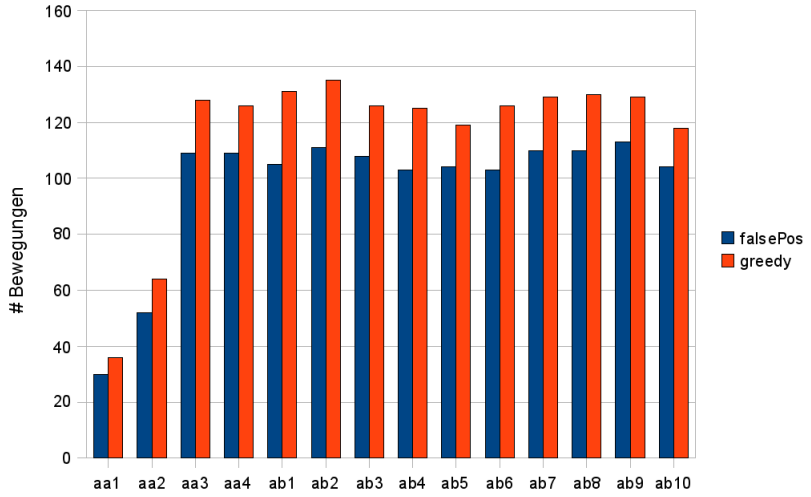

Abbildung 4: Vergleich der Anzahl der Bewegungen der Lösung des Greedy-Algorithmus mit unterer Schranke der falschen Positionen für verschiedene *Cold-Buffer*-Instanzen.