Technische Universität Ilmenau
Fakultät für Wirtschaftswissenschaften
Fachgebiet Wirtschaftsinformatik 1
Prof. Dr. P. Gmilkowsky

Hauptseminar zur Wirtschaftsinformatik
Sommersemester 2003

Shifting-Bottleneck-Heuristik zur Lösung von Schedulingproblemen mit terminorientierter Zielfunktion

Bearbeiter: René Drießel
Termin: 08.04.2003

Inhaltsverzeichnis

1 Einleitung und Zielstellung

Durch den rasanten technischen Fortschritt im letzten Jahrhundert ist die Komplexität der industriellen Fertigung immer weiter angestiegen. Die daraus resultierende höhere Produktivität führte zu einem Wandel von einem Verkäufer- zu einem Käufermarkt. Dieser Prozess begann ungefähr in den sechziger Jahren des vorigen Jahrhunderts und setzt sich bis heute fort.

Speziell die Fertigung von Halbleiterprodukten ist von enormer Komplexität geprägt. Aus einem einzigen Wafer können mehrere tausend verschiedene Chips entstehen. Für jeden Wafer kann eine andere Maschinen- und Arbeitsgangfolge notwendig sein.

Um diesen neuen Anforderungen gerecht zu werden, benötigt man ein gutes und verlässliches Scheduling der Jobs. Leider sind die meisten Planungsaufgaben innerhalb der industriellen Fertigung zu komplex. Aus diesem Grund werden häufig Heuristiken angewandt, da exakte Verfahren nur eingeschränkt einsetzbar sind.

Die Shifting-Bottleneck-Heuristik (vgl. [Pinedo 2002]) bietet einige interessante Ansätze um sowohl Laufzeit-Effizienz als auch eine hohe Ergebnisqualität zu erreichen.

Ziel dieser Arbeit ist das Vorstellen der originalen Shifting-Bottleneck-Heuristik und die Anpassung dieser auf das Leistungsmaßder gewichteten Verspätung. Desweiteren soll auf verschiedene Probleme der Heuristik hingewiesen werden.

2 Notation

Im folgenden wird angenommen, dass die Anzahl der Fertigungsaufträge und die Anzahl der Maschinen endlich sind. Die Anzahl der Fertigungsaufträge wird durch n repräsentiert und die Maschinenanzahl durch m. Das Paar (i, j) stellt eine einzelne Operation des Jobs j auf der Maschine i dar. Die folgenden Variablen dienen zur Beschreibung von einem Job j:

Prozesszeit $\mathbf{p_{i,j}}$: Die Prozesszeit $p_{i,j}$ repräsentiert die Bearbeitungszeit der Operation (i, j) des Jobs j auf der Maschine i.

Freigabezeitpunk $\mathbf{r_j}$: Dies ist die Ankunftszeit des Jobs im System. Der Job kann frühestens zu diesem Zeitpunkt bearbeitet werden.

Fälligkeitszeitpunkt $\mathbf{d_j}$: Dies ist der Zeitpunkt an dem der Job fertig abgearbeitet sein sollte. Eine Verfrühung oder Verspätung ist erlaubt.

Priorität $\mathbf{w_j}$: Die Priorität des Jobs.

Fertigstellungszeitpunkt $\mathbf{C_j}$: Der Fertigstellungszeitpunkt des Jobs j.

Abweichung $\mathbf{L_j}$: Die Abweichung des Fertigstellungszeitpunktes vom Fälligkeitszeitpunkt ist definiert als $L_j := C_j - d_j$

maximale Abweichung $\mathbf{L_{max}}$: L_{max} ist die maximale Abweichung (L_j) der Jobs. $L_{max} := max(L_j)$ $\forall j \in (1 \ldots n)$

Verspätung $\mathbf{T_j}$: T_j ist definiert als $max(C_j - d_j, 0) = max(L_j, 0)$ und repräsentiert die Verspätung des Jobs j im Bezug auf seinen Fälligkeitszeitpunkt.

Ein Einplanungsproblem kann durch ein Tripel $\alpha|\beta|\gamma$ beschrieben werden (vgl. [Pinedo 2002]). Das α-Feld beschreibt die Maschinenumgebung durch einen einzelnen Eintrag. Das β-Feld enthält die Prozesscharakteristik sowie verschiedene Einschränkungen. Es kann aus einem oder mehreren Einträgen bestehen. Im γ-Feld steht welche Prozessmerkmale optimiert, das heißt minimiert oder maximiert werden sollen. Häufig enthält das γ-Feld nur einen einzigen Eintrag.

In dieser Arbeit werden nur Job-Shop-Probleme betrachtet. Bei diesen ist das α-Feld immer *Jm*. Bei einem Job-Shop-Problem existieren m Maschinen und n verschiedene Jobs. Jeder Job hat seine eigene Arbeitsfolge. Wenn ein einzelner Job eine Maschine mehrmals in einer Arbeitsfolge belegen kann, können Schleifen auftreten. Wenn dies der Fall ist, so wird in das β-Feld *recrc* (für *recirculation*) eingetragen. Die Verwendung von *Batch*-Maschinen wird entsprechend mit einem Eintrage von *batch* in das β-Feld gekennzeichnet. Hierbei werden mehrere Jobs zu einem Batch zusammengefasst und von der Maschine gleichzeitig bearbeitet. Ist der Batch einmal in Bearbeitung, kann er nicht mehr unterbrochen noch können zusätzliche Jobs dem Batch hinzugefügt werden (vgl. [Mason u. a. 2002]). Werden bei einem Unterproblem *parallele* Maschinen verwendet so wird das α-Feld auf *Pm* gesetzt.

Innerhalb dieser Arbeit werden zwei Zielkriterien behandelt. Zum einen die Minimierung des spätesten Fertigstellungstermins ($\gamma = C_{max}$), welche als $max(C_1, \ldots, C_n)$ definiert ist , und zum anderen die Minimierung der gewichteten Verspätung ($\gamma = \sum w_j T_j$). Eine kurze Durchlaufzeit kann eine hohe Auslastung der Maschinen implizieren. Die gewichtete Verspätung dagegen bezieht sich nur auf die zusätzlichen Kosten, die durch den Plan entstehen (im Vergleich zu den gewünschten Kosten). Im Unterschied zu dem Kriterium des spätesten Fertigstellungstermins wird dabei nur die Einhaltung des Fälligkeitszeitpunktes d_j betrachtet und nicht darüber hinaus optimiert.

3 Shifting-Bottleneck-Heuristik für das Leistungsmaß „Spätester Fertigstellungstermin“

Die Shifting Bottleneck Heuristik wurde für Probleme des Typs $J_m||C_{max}$ von Adams et al. (vgl. [Adams u. a. 1988]) entwickelt. Diese Heuristik ist eine Dekompositionsheuristik. Ergebnis der Dekomposition sind verschiedene Belegungsprobleme für die Einzelmaschinen. Nach der Lösung der einzelnen Probleme wird die am meisten kritische Maschine (in Bezug auf das Zielkriterium) zuerst eingeplant. Es wird zunächst die ursprüngliche Heuristik von Adams et al. (vgl. [Adams u. a. 1988]) vorgestellt. Im nächsten Kapitel erfolgt dann die Vorstellung einiger Erweiterungen dieses Ansatzes.

3.1 Algorithmus

Die Shifting-Bottleneck-Heuristik lässt sich durch die folgenden 5 Schritte grob beschreiben (M repräsentiert die Menge aller Maschinen). Der Arbeitsplan wird durch einen gerichteten Graphen beschrieben. Die Knoten repräsentieren die einzelnen Arbeitsgänge. Die Kanten repräsentieren die Abhängigkeiten zwischen den Arbeitsgängen. An den Kanten ist die Bearbeitungszeit für die Arbeitsgänge angetragen.

Schritt 1 (Initialbedingungen)
Setze $M_0 = \emptyset$;
der Graph G enthält alle konjunktive Kanten (Auftragsrestiktionen) und keine disjunktiven Kanten;
$C_{max}(M_0)$ ist gleich dem längsten Pfad im Graphen G

Schritt 2 (Analyse der Maschinen, welche eingeplant werden sollen)
$\forall i \in \{M - M_0\}$ tue folgendes:
erzeuge eine Instanz $1|r_j|L_{max}$ mit
$r_{i,j}$ = der längste Pfad von der Quelle bis zum Knoten (i, j);
$d_{i,j}$ = der längste Pfad vom Knoten (i, j) zur Senke reduziert um $p_{i,j}$;
$L_{max}(i)$ = minimales L_{max} für die Maschine i

Schritt 3 (Enpass Auswahl und Einplanung)
Setze $L_{max}(k) = max(L_{max}(i)) \; \forall i \in \{M - M_0\}$;
Einplanung der Maschine k in der Reihenfolge, wie Schritt 2 sie ergeben hat;
Setze $M_0 = M_0 \cup \{k\}$

Schritt 4 (Erneutes Optimieren und Einplanen aller schon eingeplanten Maschinen)

Schritt 5 (Abbruchkriterium)
Wenn $M_0 = M$ dann Stopp, sonst gehe zu Schritt 2

Zum Schluss enthält der Graph Maschinen und Auftragsrestiktionen, aus denen die Reihenfolge der Maschinenbelegung ablesbar ist. Die einzelnen Phasen werden im Folgenden näher besprochen.

3.1.1 Initialisierung und Abbruchkriterium

Die Menge M enthält alle Maschinen, welche eingeplant werden sollen. Die Menge M_0 enthält alle Maschinen, welche schon eingeplant sind. Am Anfang ist diese Menge leer. Wenn alle Maschinen eingeplant sind, also $M_0 = M$, dann terminiert der Algorithmus.

3.1.2 Lösen der Unterprobleme

Innerhalb dieser Phase wird untersucht, welche Maschine als nächstes eingeplant werden soll. Dabei wird davon ausgegangen, dass die am meisten kritischse Maschine, ein sogenannter Engpass (engl. Bottleneck), zuerst eingeplant werden soll. Da die schon eingeplanten Maschinen in der Menge M_0 nicht mehr berücksichtigt werden, verschiebt sich der Engpass von Iteration zu Iteration. Dies ist auch der Grund für den Namen der Heuristik.

Für jede Maschine aus $(M - M_0)$, welche noch nicht eingeplant wurde, wird ein Unterproblem $1|r_j|L_{max}$ gelöst. Die nötigen Eingangsdaten sind $r_{i,j}$ und $d_{i,j}$, welche aus dem Graphen ermittelt werden und $p_{i,j}$ welche für jede Operation gegeben sind.

Für die eigentliche Lösung der Unterprobleme kann man verschiedene Algorithmen verwenden. Im folgenden Beispiel wird das Problem durch vollständige Enumeration gelöst, welche eine optimale Lösung liefert. Bei diesem Verfahren werden einfach alle Möglichkeiten der Maschinenbelegung ausprobiert. Die Maschinenbelegung, welche die geringste maximale Verspätung (L_{max}) liefert, wird genommen. Es ist jedoch auch möglich, einen anderen Algorithmus zu verwenden. Dieser Algorithmus muss nur in der Lage sein, die am meisten kritische Maschine und die einzufügenden Kanten zu liefern.

3.1.3 Engpassauswahl

Bei jedem gelösten Unterproblem ergibt sich eine lokal optimale Einplanreihenfolge, welche zu einer bestimmten Verspätung (L_{max}) führt. Es wird nun die Maschine als Engpass (k) genommen, bei der die Verspätung maximal ist.

Die Einplanreihenfolge des Engpasses wird in den Graphen als disjunktive Kanten eingefügt. Gleichzeitig wird der Engpass der Menge M_0 hinzugefügt.

3.1.4 Reoptimierung

Normalerweise wäre eine Iteration jetzt beendet. Es hat sich jedoch als vorteilhaft erwiesen, alle bisher eingeplanten Maschinen nochmals zu optimieren und wieder einzuplanen.

Für alle Maschinen in M_0, außer k, werden die disjunktiven Kanten gelöscht. Danach werden die Unterprobleme erneut gelöst (mit anderen $r_{i,j}$ und $d_{i,j}$). Die minimalen $L_{max}(i)$ werden wieder in den Graphen eingefügt.

3.2 Beispiel

In diesem Abschnitt wird der Shifting-Bottleneck-Algorithmus, wie weiter oben beschrieben, anhand eines kleinen Beispiels näher erläutert.
Gegeben ist eine Fertigungsanlage mit vier Maschinen ($i \in \{1,2,3,4\}$) und drei Fertigungsaufträgen ($j \in \{1,2,3\}$). Die Arbeitspläne der einzelnen Fertigungsaufträge sind ebenfalls gegeben (Tabelle 1).

Fertigungsauftrag	Arbeitsplan	Prozesszeiten
1	1,2,3	$p_{1,1} = 10,\ p_{2,1} = 8,\ p_{3,1} = 4$
2	2,1,4,3	$p_{2,2} = 8,\ p_{1,2} = 3,\ p_{4,2} = 5,\ p_{3,2} = 6$
3	1,2,4	$p_{1,3} = 4,\ p_{2,3} = 7,\ p_{4,3} = 3$

Tabelle 1: Arbeitspläne

3.2.1 Ausgangssituation

Aus den gegebenen Daten erzeugt man zunächst einen Ausgangsgraphen (Abbildung 3/1), welcher vorerst nur die Abhängigkeiten durch den Arbeitsplan widerspiegelt. Die Menge M_0 ist leer ($M_0 = \emptyset$). Das $C_{max}(0)$ beträgt 22.

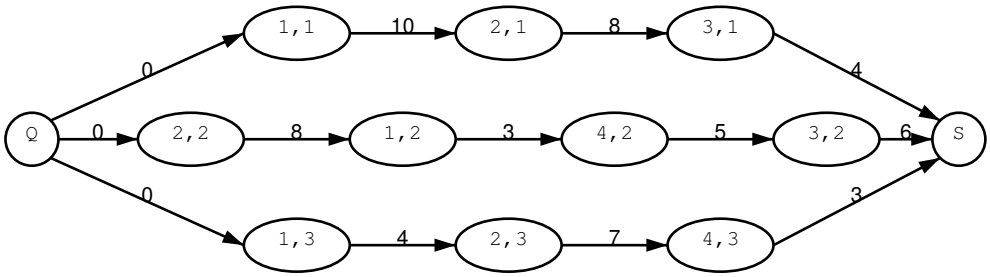

Abb. 3/1: Ausgangsgraph

3.2.2 Erste Iteration

In der ersten Iteration werden zunächst für alle drei Maschinen das Unterproblem $(1|r_j|L_{max})$ formuliert. Die $r_{i,j}$ werden dabei über eine Vorwärtsterminierung errechnet und die $d_{i,j}$ durch eine Rückwärtsterminierung. Daraus ergeben sich die Eingangsdaten für die Unterprobleme wie in Tabelle 2 angegeben. Aus diesen Eingangsdaten ergeben sich die Lösungen in Tabelle 3.

i	$p_{i,1}$	$r_{i,1}$	$d_{i,1}$	$p_{i,2}$	$r_{i,2}$	$d_{i,2}$	$p_{i,3}$	$r_{i,3}$	$d_{i,3}$
1	10	0	10	3	8	11	4	0	12
2	8	10	18	8	0	8	7	4	19
3	4	18	22	6	16	22	-	-	-
4	-	-	-	5	11	16	3	11	22

Tabelle 2: 1. Iteration – Eingangsdaten der Teilprobleme

i	Einplanreihenfolge	$L_{max}(i)$
1	1, 2, 3	5
2	2, 3, 1	5
3		4
4		0

Tabelle 3: 1. Iteration – Ergebnisse der Teilprobleme

Da bei Maschine 1 die maximale Verspätung am höchsten ist, wird diese Maschine eingeplant. Es werden also zwei Kanten in den Ausgangsgraphen (Abbildung 3/1) eingefügt ($(1,1) \Rightarrow (1,2)$ und $(1,2) \Rightarrow (1,3)$). Daraus ergibt sich ein neuer Graph (Abbildung 3/2). Gleichzeitig wird die Maschine 1 der Menge M_0 hinzugefügt ($M_0 = \{1\}$).
Innerhalb des neuen Graphen (Abbildung 3/2) ergibt sich nun ein neues C_{max} von 27.

$$C_{max}(\{1\}) = C_{max}(\emptyset) + L_{max}(1) = 22 + 5 = 27.$$

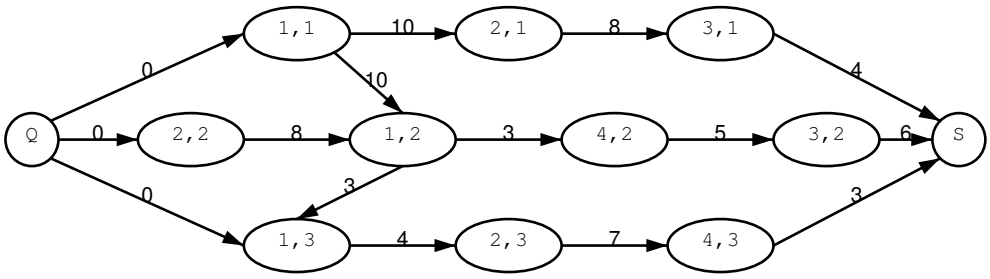

Abb. 3/2: Graph nach der 1. Iteration

3.2.3 Zweite Iteration

In der zweiten Iteration werden nur noch die Maschinen 2, 3 und 4 berücksichtigt. Die Eingangsdaten für die Teilprobleme sind in Tabelle 4 gegeben.

i	$p_{i,1}$	$r_{i,1}$	$d_{i,1}$	$p_{i,2}$	$r_{i,2}$	$d_{i,2}$	$p_{i,3}$	$r_{i,3}$	$d_{i,3}$
2	8	10	23	8	0	10	7	17	24
3	4	18	27	6	18	27	-	-	-
4	-	-	-	5	13	21	3	24	27

Tabelle 4: 2. Iteration – Eingangsdaten der Teilprobleme

Daraus ergeben sich wiederum die Ausgangsdaten, wie in Tabelle 5. Es wird Maschine 2 gewählt. Die Menge M_0 ist also $M_0 = \{1,2\}$. Der resultierende Graph ist in Abbildung 3/3 gegeben. C_{max} ist demzufolge:

$$C_{max}(\{1,2\}) = C_{max}(\{1\}) + L_{max}(2) = 27 + 1 = 28.$$

i	Einplanreihenfolge	$L_{max}(i)$
2	2, 1, 3	1
3		1
4		0

Tabelle 5: 2. Iteration – Ergebnisse der Teilprobleme

An dieser Stelle sollte man nun versuchen, die Maschine 1 zu reoptimieren. Hierzu entfernt man die eingefügten Kanten aus der ersten Iteration (Abbildung 3/2). Daraus ergeben sich die entsprechenden Eingangsdaten (Tabelle 6). Das Ergebnis der Reoptimierung ist allerdings dasselbe. Die vorher entfernten Kanten ($(1,1) \Rightarrow (1,2)$ und $(1,2) \Rightarrow (1,3)$) werden wieder eingefügt.

3.2.4 Dritte Iteration

In der dritten Iteration sind nur noch die Maschinen 3 und 4 einzuplanen. Aus den Eingangsdaten (Tabelle 7) ergibt sich eine Einplanreihenfolge von $1,2$ für Maschine 3 und $2,3$ für Maschine 4. In beiden Fällen erhöht sich C_{max} ($L_{max}(3) = L_{max}(4) = 0$) nicht. Für die nächste Einplanung wird Maschine 3 gewählt (Abbildung 3/4).
Die Menge M_0 ist dann $M_0 = \{1,2,3\}$. Bei der Reoptimierung entfernt man zunächst die Kanten für die Maschine 1 und erhält somit $r_{1,1} = 0$, $d_{1,1} = 10$, $r_{1,2} = 8$, $d_{1,2} = 11$, $r_{1,3} = 0$ und $d_{1,3} =$

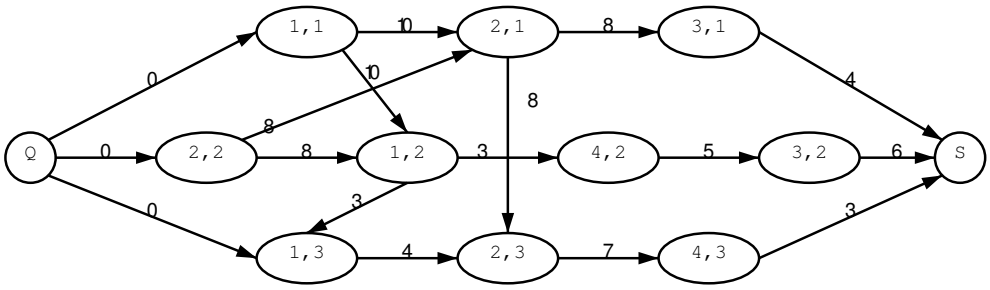

Abb. 3/3: Graph nach der 2. Iteration

j	1	2	3
$p_{1,j}$	10	3	4
$r_{1,j}$	0	8	0
$d_{1,j}$	10	11	13

Tabelle 6: 2. Iteration – Reoptimierung

13. Da die Eingangsdaten dieselben sind wie bei der Reoptimierung in der zweiten Iteration, ändert sich an den Kanten nichts (sie werden wieder eingefügt). Danach erfolgt dasselbe mit Maschine 2. Hier werden ebenfalls die Kanten entfernt ($r_{2,1} = 10$, $d_{2,1} = 24$, $r_{2,2} = 0$, $d_{2,2} = 11$, $r_{2,3} = 18$ und $d_{2,3} = 25$). Da sich auch bei dieser Optimierung nichts ändert, werden die ursprünglichen Kanten wieder eingefügt.

3.2.5 Vierte Iteration

Nun ist nur noch Maschine 4 übrig. Die Einplanreihenfolge $2,3$ wurde in der dritten Iteration schon bestimmt. Die Reoptimierung der Maschinen 1, 2 und 3 bringt diesmal auch keine Verbesserung. Nach der Einplanung ergibt sich folgender Arbeitsplan (Abbildung 3/5):

- Arbeitsgangfolge $1,2,3$ auf Maschine 1,
- Arbeitsgangfolge $2,1,3$ auf Maschine 2,
- Arbeitsgangfolge $2,1$ auf Maschine 3,
- Arbeitsgangfolge $2,3$ auf Maschine 4.

Die Gesamtdurchlaufzeit ist 28.

3.3 Grenzen der Heuristik

In der Praxis sind die meisten Problemstellungen allerdings komplizierter als in der vorigen Lösung beschrieben. Die Unterprobleme sind meistens komplexer als das, welches hier verwendet wurde. In einer Halbleiterfabrik existieren parallele und Batch-Maschinen (vgl. [Mason u. a. 2002]). Diese Schedulingprobleme können nicht mehr mit vollständiger Enumeration gelöst werden. Außerdem können Schleifen innerhalb des Graphen auftreten. Desweiteren werden Umrüstzeiten innerhalb der ursprünglichen Heuristik nicht berücksichtigt.

Das nächste Beispiel soll das Schleifenproblem näher erläutern (Tabelle 8).

i	$p_{i,1}$	$r_{i,1}$	$d_{i,1}$	$p_{i,2}$	$r_{i,2}$	$d_{i,2}$	$p_{i,3}$	$r_{i,3}$	$d_{i,3}$
3	4	18	28	6	18	28	-	-	-
4	-	-	-	5	13	22	3	24	28

Tabelle 7: 3. Iteration – Eingangsdaten der Teilprobleme

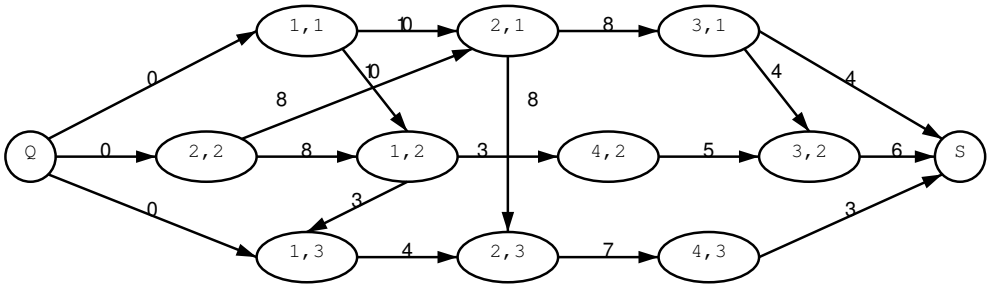

Abb. 3/4: Graph nach der 3. Iteration

3.3.1 1. Iteration

Die optimale Lösung für die Maschinen 1 und 2 hat ein $L_{max} \leq 0$. Für Maschine 3 ergibt sich ein L_{max} von 4. Die Maschine 3 wird demzufolge eingeplant und die Kante $(3,4) \rightarrow (3,3)$ wird eingefügt.

3.3.2 2. Iteration

Die optimale Lösung für die Maschinen 1 und 2 haben beide bei $L_{max} = -6$. Die Maschine 1 wird gewählt und die Kante $(1,2) \rightarrow (1,1)$ eingefügt. Wenn nun Maschine 2 eingeplant wird ergibt sich immer ein $L_{max} \leq 0$. Wenn die Kante $(2,1) \rightarrow (2,2)$ eingefügt würde, dann würde es zu einer Schleife innerhalb des Graphen kommen (Abbildung 3/6). Um dies zu verhindern, muss nach der 2. Iteration eine Kante $(2,2) \rightarrow (2,1)$ mit der Länge 3 eingefügt werden (3 ist die Summe der Kanten zwischen $(2,2)$ und $(2,1)$). Mit Hilfe dieser Bedingung erzeugt die dritte Iteration keine Schleife.

Wie in obigen Beispiel zu sehen ist, kann die Schleifenproblematik sehr leicht auftreten. Besonders problematisch wird dieses Problem bei massiv parallelen Maschinen. In Mason et al. hat sich in [Oey und Mason 2001] näher mir der Schleifenproblematik für *Batch*.Maschinen auseinandergesetzt.

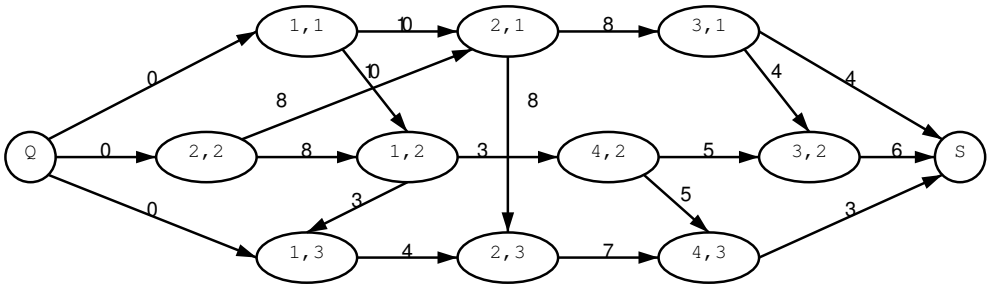

Abb. 3/5: Graph nach der 4. Iteration

Auftrag	Maschinenfolge	Bearbeitungszeiten
1	$1,2$	$p_{11} = 1, p_{21} = 1$
2	$2,1$	$p_{22} = 1, p_{12} = 1$
3	3	$p_{33} = 4$
4	3	$p_{34} = 4$

Tabelle 8: Arbeitspläne

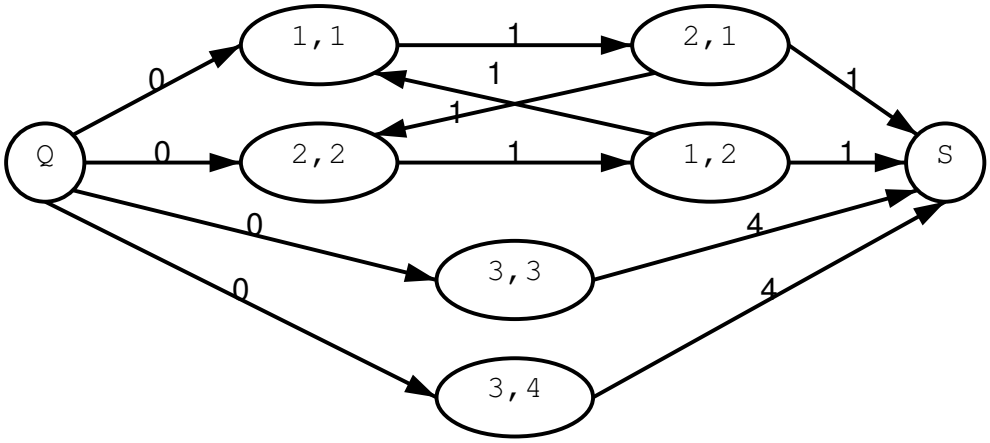

Abb. 3/6: Endgraph mit Schleife

4 Shifting-Bottleneck-Heuristik für das Leistungsmaß „Gewichtete Verspätung“

Durch den modularen Aufbau der Heuristik ist es relativ einfach, die verschiedenen Bestandteile auszutauschen. Da die ursprüngliche Shifting-Bottleneck-Heuristik für die meisten praktischen Probleme nicht ausreicht, wurde sie in der Literatur bereits an vielen Stellen erweitert (vgl. [Mason u. a. 2002], [Mason und Fowler 2000] und [Pinedo 2002]).

Es existieren prinzipiell zwei Möglichkeiten die Heuristik anzupassen:

- Anpassung der Zielfunktion und
- Anpassung der Unteralgorithmen.

Im Folgenden wird eine Anpassung der Heuristik an das Maß der gewichteten Verspätung ($Jm||\sum w_j T_j$) besprochen (vgl. [Pinedo 2002]). Der größte Unterschied zu dem vorigen Algorithmus ist, dass der Graph nun mehrere Endknoten besitzt (Abbildung 4/1). Dies liegt daran, dass die Fertigstellungszeit eines jeden Jobs berücksichtigt wird.

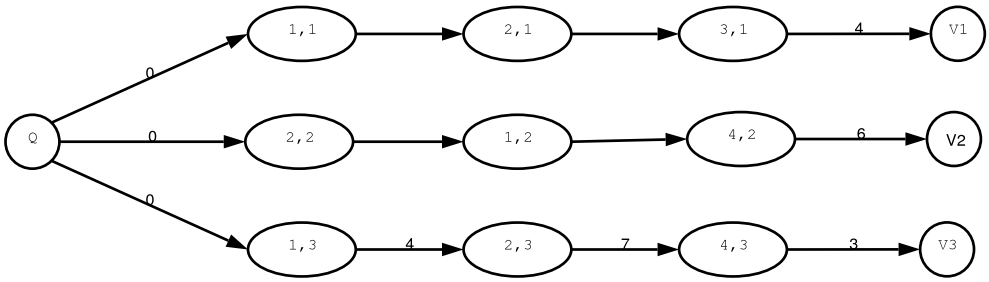

Abb. 4/1: Graph für gewichtete Verspätung

4.1 Algorithmus

Der Algorithmus ändert sich nur leicht:

Schritt 1 (Initialbedingungen)
Setze $M_0 = \emptyset$;
der Graph G enthält alle konjunktiven Kanten und keine disjunktiven Kanten

Schritt 2 (Analyse der Maschinen, welche eingeplant werden sollen)
$\forall i \in \{M - M_0\}$ tue folgendes:
erzeuge eine Instanz $1||\sum w_j T_j$;
Minimiere $\sum w_j T_j$ für die Maschine i

Schritt 3 (Enpass Auswahl und Einplanung)
Wähle die kritischste Maschine k aus $\{M - M_0\}$;
Einplanung der Maschine k in der Reihenfolge, wie sie Schritt 2 ergeben hat;
Setze $M_0 = M_0 \cup \{k\}$

Schritt 4 (Erneutes Optimieren und Einplanen aller schon eingeplanten Maschinen)

Schritt 5 (Abbruchkriterium)
Wenn $M_0 = M$ dann Stopp, sonst gehe zu Schritt 2

4.1.1 Initialisierung und Abbruchkriterium

Wie in der originalen Heuristik existieren hier zwei Mengen (M und M_0). Die Menge M_0 ist am Anfang leer. Wenn alle Maschinen eingeplant sind, also $M_0 = M$ gilt, dann terminiert der Algorithmus.

4.1.2 Lösen der Unterprobleme

Wie beim ursprünglichen Shifting-Bottleneck-Algorithmus kann man verschiedene Algorithmen verwenden, welche das Unterproblem lösen. Da das Problem $1||\sum w_j T_j$ NP-vollständig ist (vgl. [JK u. a. 1977]), ist die Verwendung einer Heuristik anzuraten. Pinedo und Mason et al. (vgl. [Pinedo 2002] und [Mason u. a. 2002]) verwenden die ATC-Prioritätsregel (Apparent Tardiness Cost), welche eine Kombination aus der WSPT-Regel (Weighted Shortest Processing Time First) und der MS-Regel (Minimum Slack First) ist. An dieser Stelle soll die Variante von Pinedo [Pinedo 2002] näher erläutert werden.

Bei dieser Regel wird für jeden Job j, welcher auf der Maschine i abgearbeitet werden soll, ein Ranking-Index ermittelt. Der Job mit dem höchsten Ranking-Index wird zuerst eingeplant. Dieser Ranking-Index ist eine Funktion der Zeit t, zu der die Maschine frei wird. Für die Operation (i, j) kann der Ranking-Index mittels

$$I_{ij}(t) = \underbrace{\sum_{k=1}^{n} \frac{w_k}{p_{ij}}}_{WSPT-Regel} \underbrace{exp\left(-\frac{max(d_{ij}^k - p_{ij} + (r_{ij} - t), 0)}{K\overline{p}}\right)}_{MS-Regel}$$

ermittelt werden. Hierbei ist w_k die Priorität des Jobs k, $\overline{p}$ ist die durchschnittliche Dauer der Operationen auf Maschine i und d_{ij}^k ist der lokale Fälligkeitszeitpunkt der Operation (i, j), des Jobs k. Wenn es keinen Weg von der Operation (i, j) zur Senke des Jobs k gibt, dann ist d_{ij}^k unendlich. Der Summand des Ranking-Index würde demzufolge 0. Der Faktor K ist ein empirischer Parameter. Lee und Pinedo (vgl. [Lee und Pinedo 1997]) haben in verschiedenen Untersuchungen einige Berechnungsvorschriften für den Parameter K entwickelt.

4.1.3 Engpassauswahl

Der Engpass ist auch hier wieder die kritischste Maschine, die auch zuerst eingeplant wird. Mit folgender Funktion lässt sich bestimmen, wie kritisch eine Maschine ist:

$$\sum_{k=1}^{n} w_k(C_k'' - C_k')exp\left(-\frac{(d_k - C_k'')^+}{K}\right)$$

C_k'' ist hierbei der Fertigstellungszeitpunkt des Jobs k, wenn die Maschine eingeplant wird. C_k' ist der Fertigstellungszeitpunkt des Jobs k ohne das die Maschine eingeplant wird. $C_k'' - C_k'$ ist also ein Maß dafür, wie sich der Job k verspätet, wenn die Maschine eingeplant wird. Die Variable d_k steht für den Fääligkeitszeitpunkt des Jobs k. K ist an dieser Stelle wieder ein empirischer Parameter, mit dem das Verhalten dieser Regel beeinflusst werden kann.

4.1.4 Reoptimierung

Auch bei dieser Version des Algorithmus ist eine Wiedereinplanung möglich und angeraten. Das Vorgehen ist wieder dasselbe wie in der ursprünglichen Heuristik.

4.2 Weitere Modifikationen in der Literatur

Innerhalb der oben vorgestellten Heuristik wird allerdings das Schleifenproblem auch noch nicht berücksichtigt. Desweiteren fehlen parallele und *Batch*-Maschinen.

Mason et al. (vgl. [Mason u. a. 2002]) hat die oben vorgestellte Heuristik auf $Jm|r_j, s_{jk}, batch, recrc|\sum w_j T_j$ erweitert. Das Unterproblem wurde ebenfalls entsprechend erweitert ($Pm|r_j, s_{jk}, recrc|\sum w_j T_j$). Es wurden also noch Rüstzeiten (s_{jk}), Batching- ($batch$) und parallele (Pm) Maschinen sowie die Möglichkeit von Schleifen ($recrc$) berücksichtigt. Die Unterschiede zu der im vorigen Abschnitt vorgestellten Heuristik sind in den folgenden Abschnitten erläutert.

4.2.1 Rüstzeiten

Innerhalb des Shifting-Bottleneck-Graphen gibt es keinen Hinweis auf Rüstzeiten. Diese werden erst bei der Lösung der Unterprobleme bekannt. Die Rüstzeiten werden zu den normalen Bearbeitungszeiten addiert und als disjunktive Kanten in den Graphen eingefügt.

4.2.2 Parallele Maschinen

Pinedo (vgl. [Pinedo 2002]) unterscheidet drei verschiedene Arten von parallelen Maschinen.

- Maschinen mit gleicher Geschwindigkeit
- Maschinen mit unterschiedlicher Geschwindigkeit
- die Geschwindigkeit der Maschinen hängt von den Jobs ab

An dieser Stelle soll nur der erste Fall behandelt werden. Statt einer Maschine wird bei der Lösung des Unterproblems von einer Maschinengruppe ausgegangen. Bis zur Lösung des Unterproblems gibt es auch in diesem Fall kein Indiz für parallele Maschinen innerhalb des Graphen. Nach der Lösung und Einplanung der Unterprobleme enthält der Graph die Informationen, in welcher Reihenfolge die Jobs bearbeitet werden.

4.2.3 Batching-Maschinen

Eine Darstellung, wie man eine Batching-Maschine innerhalb des Shifting-Bottleneck-Graphen darstellen kann, findet sich in Abbildung 4/2. In diesem Beispiel ist Maschine 1 eine Batching Maschine mit einer Batchgröße von $b = 2$. Die Bearbeitungszeiten für den jeweiligen Batch auch hier erst nach der Lösung des Unterproblems bekannt. Aus diesem Grund erfolgt für die Operationen, die auf einer Batching Maschine bearbeitet werden, eine Verbindung mit einem besonderen Batch-Knoten.
Für die Lösung der Unterprobleme von anderen ist es wichtig, dass die lokalen Fälligkeitszeitpunkte der einzelnen Jobs berechnet werden können. Wenn die Batching-Maschine noch nicht eingeplant ist, dann existieren allerdings noch keine Kanten. Daraus ergibt sich das Problem, dass einige Knoten nicht mit den Senken verbunden sind. Um dieses Problem zu lösen, schlägt Mason (vgl. [Mason u. a. 2002]) die anfängliche Einfügung von konjunktiven Kanten vor. Abbildung 4/3 zeigt die konjunktiven Kanten für zwei Jobs bei einer Batchgröße von $b = 2$. Wenn die Batching-Maschine eingeplant wird, dann werden die überschüssigen Kanten entfernt.
Eine andere Möglichkeit wäre sicherlich die Batching-Maschinen zuerst einzuplanen, bevor die eigentliche Shifting-Bottleneck-Heuristik angewendet wird.

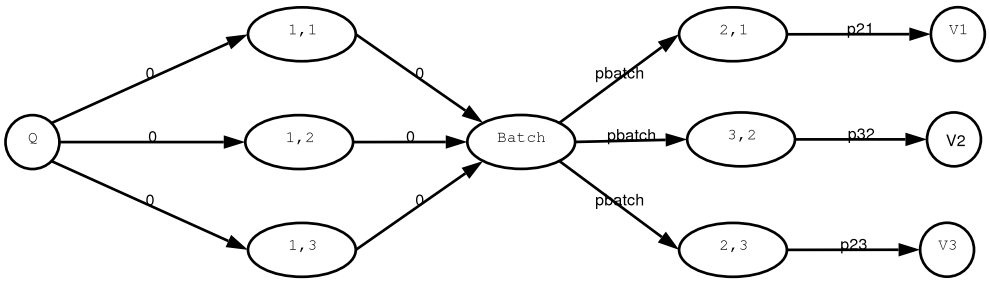

Abb. 4/2: Graph mit Batching-Maschinen

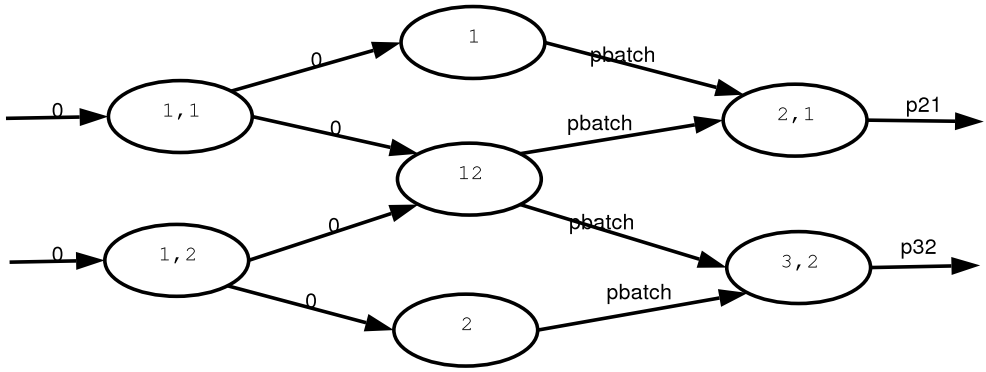

Abb. 4/3: Anfängliche konjunktive Kanten bei einer Batchgröße von $b = 2$

4.2.4 Lösen der Unterprobleme

Die einzelnen Lösungen für die Unterprobleme hängen von der Art der Maschinen und Probleme ab. Für die Lösung eines $Pm|r_j, s_{jk}|\sum w_j T_j$-Problems entwickelten Lee und Pinedo (vgl. [Lee und Pinedo 1997]) eine Lösung, die ebenfalls auf der ATC-Regel aufbaut. Bei dieser Regel werden allerdings die Umrüstzeiten mit berücksichtigt. Sobald eine Maschine frei geworden ist, wird ein Ranking-Index I_j für jeden der verbleibenden Jobs berechnet. Der Job mit dem höchsten Ranking-Index wird eingeplant.

$$I_j(t,l) = \underbrace{\sum_{k=1}^{n} \frac{w_k}{p_j}}_{WSPT-Regel} \underbrace{exp\left(-\frac{(d_j^k - p_j + (r_j - t)^+)}{K_1 \overline{p}}\right)}_{MS-Regel} \underbrace{exp\left(-\frac{s_{lj}}{K_2 \overline{s}}\right)}_{SST-Regel}$$

Bei diesem Index erfolgt die Berücksichtigung der Rüstzeiten durch die SST-Regel (Shortest Setup Time First). Inwieweit diese in die Gesamtregel einfließen, bestimmt der empirische Faktor K_2. Äquivalent zu der MS-Regel steht $\overline{s}$ für die durchschnittlichen Rüstzeiten und s_{lj} für die Rüstzeiten zwischen dem fertiggestellten Job l und dem Job j. Die Variable t beinhaltet die Zeit, zu der die Maschine freigeworden ist.

Mason et al. (vgl. [Mason u. a. 2002]) hat die ATC-Regel von Lee und Pinedo für das Problem der Batching-Maschinen erweitert. Für eine Maschine i kann mit Hilfe der folgenden Formel ein Ranking-Index für jeden potentiellen Batch berechnen.

$$\begin{aligned} I_{bj}(t,l) &= \sum_{k=1}^{n} \frac{w_{bj}}{p_{bj}} exp\left(-\frac{(d_{bj}^k - p_{bj} + (r_{bj} - t)^+)}{K_1 \overline{p}}\right) exp\left(-\frac{s_{bl,bj}}{K_2 \overline{s}}\right) \\ & \quad min\left(\frac{LotsAvailToBatch_{bj}}{MaxBatchSize_i}, 1\right) \end{aligned}$$

Hierbei steht w_{bj} für die mittlere Gewichtung der Jobs in dem Batch bj und p_{bj} ist die Bearbeitungszeit des Batches bj. Ebenso ist d_{bj}^{k} der lokale Fälligkeitszeitpunkt des Batches bj des Jobs k. Die Rüstzeiten, die bei dem Umrüsten von Batch bl auf bj auftreten, sind durch $s_{bl,bj}$ repsäsentiert. Die Variablen $\overline{s}$ und $\overline{p}$ stehen für die durchschnittlichen Rüstzeiten beziehungsweise die durchschnittlichen Bearbeitungszeiten. Die restlichen Variablen sollten selbsterklärend sein.

4.2.5 Engpassauswahl

Die Engpassauswahl bei Mason et al. erfolgt wie bei Pinedo (vgl. [Pinedo 2002]).

$$\sum_{k=1}^{n} w_k(C_k{}'' - C_k{}')exp\left(-\frac{(d_k - C_k{}'')^+}{K}\right)$$

$C_k'' - C_k'$ stellt die Verspätung des Jobs k dar, wenn die Maschine eingeplant wird. K ist ein empirischer Parameter, mit dem das Verhalten dieser Regel beeinflusst werden kann.

5 Ausblick

Verschiedene Untersuchungen haben ergeben, dass die Shifting-Bottleneck-Heuristik sehr effizient ist. Sie lieferte für ein Standardproblem mit 10 Maschinen und 10 Aufträgen sehr schnell eine gute Lösung. Im Vergleich dazu benötigt ein Branch-And-Bound-Ansatz mehrere Stunden an CPU-Zeit (vgl. [Pinedo 2002]). Durch den dekompositionellen Ansatz der Heuristik ist außerdem eine leichte Parallelisierbarkeit gegeben. Die einzelnen Maschinenprobleme können auf einfache Weise verteilt gelöst werden.

Bei einer Untersuchung der Ergebnisse der Shifting-Bottleneck-Heuristik im Vergleich zu anderen Heuristiken und Prioritätsregeln von Mason et al. (vgl. [Mason und Fowler 2000]) ergab in neun von zehn Fällen die Shifting-Bottleneck-Heuristik die beste Lösung. In diesen Fällen war es sogar die optimale Lösung.

Die guten Ergebnisse haben jedoch auch ihren Preis. Während die einfachen Prioritätsregeln laut Mason et al. [Mason und Fowler 2000] nur 5 Sekunden benötigten, benötigte die Shifting-Bottleneck-Heuristik 4,5 Minuten um ihre Ergebnisse zu liefern. Ein weiterer Nachteil ist sicherlich, dass die Optimalität der Lösung nicht garantiert ist.

Die Heuristik scheint allerdings ein guter Kompromiss zwischen Laufzeit und Ergebnisqualität zu sein.

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[Adams u. a. 1988] ADAMS, J. ; BALAS, E. ; ZAWACK, D.: The shifting bottleneck procedure for job shop scheduling. In: *Managment Science* 34 (1988), S. 391–401

[JK u. a. 1977] JK, Lenstra ; AHG, Rinnooy K. ; P., Brucker: Complexity of machine scheduling problems. In: *Annals of Discrete Mathematics*, 1977, S. 1:343–362

[Lee und Pinedo 1997] LEE, YH ; PINEDO, ML: Scheduling jobs on parallel machines with sequence-dependent setup times. In: *European Journal of Operational Research* 100 (1997), S. 464–474

[Mason und Fowler 2000] MASON, Scott J. ; FOWLER, John W.: Maximizing delivery performance in semiconductor wafer fabrication facilities. In: *Proceedings of the 2000 Winter Simulation Conference*, 2000, S. 1458–1463

[Mason u. a. 2002] MASON, Scott J. ; FOWLER, John W. ; CARLYLE, W. M.: A modified shifting bottleneck heuristic for minimizing total weighted tardiness in complex job shops. In: *Journal of Scheduling* 5 (2002), S. 247–262

[Oey und Mason 2001] OEY, Kasin ; MASON, Scott J.: Scheduling batch processing machines in complex job shops. In: *Proceedings of the 2001 Winter Simulation Conference*, 2001, S. 1200–1207

[Pinedo 2002] PINEDO, M.: *Scheduling: Theory, Algorithm, and Systems*. Second Edition. Prentice Hall, 2002

A Implementierungsbeispiel

Im folgenden ist das Listing zu einer Beispielimplementierung der Shifting-Bottleneck-Heuristik abgedruckt. Sie ist in der Sprache C++ geschrieben und wurde unter Debian GNU/Linux mit Hilfe des gcc in der Version 3.2 getestet. Es wurde nur die Originalheuristik nach Adams et al. (vgl. [Adams u. a. 1988]) implementiert, um die Übersichtlichkeit zu waren.

Die Implementation kann mittels eines make Programms übersetzt werden. Der Aufbau des Graphen erfolgt in der Datei sbh.cc.

Listing des Makefiles:

```
OBJECTS = sbh.o node.o machine.o job.o

CC = g++

sbh: $(OBJECTS)
        $(CC) -o sbh $(OBJECTS)

sbh.cc: datamodell.hh
job.cc: datamodell.hh
machine.cc: datamodell.hh
node.cc: datamodell.hh
```

Listing von datamodell.hh:

```
#if !defined(DATAMODELL_HH)
#define DATAMODELL_HH

#include <string>
#include <set>
#include <vector>
#include <iostream>
#include <iterator>

class Machine;
class Job;
class Node;

typedef std::set<Machine*> MachineSet;
typedef std::set<Job*>     JobSet;
typedef std::set<Node*>    NodeSet;
typedef std::vector<Node*> ScheduleVector;

// Ein Knoten
class Node {
public:
  Node();
  Node(Machine& i, Job& j, double time);
  ~Node();
```

```
  Machine* i();
  Job*     j();

  double   get_completion_date();
  double   get_release_date();
  double   get_process_time();

  void     insert_arc_to(Node& node);
  void     delete_arc_to(Node& node);

  void     set_recalculate_flag();
  void     recalculate();

  void     print();
private:
  Machine* m_machine;
  Job*     m_job;

  double   m_release_date;
  double   m_processtime;
  bool     m_must_recalculate;

  NodeSet  m_nextarcs;
  NodeSet  m_prevarcs;
};

// Eine Maschine
class Machine {
public:
  Machine(std::string name);
  ~Machine();

  std::string get_name();

  void        add_node(Node* node);
  void        schedule_machine();
  double      get_criticaly();

  void        insert_arcs(ScheduleVector schedule);
  void        remove_arcs(ScheduleVector schedule);

  ScheduleVector get_schedule();

  void        print();
private:
  std::string m_name;
  NodeSet nodes;
```

```
  ScheduleVector schedule_arcs;
  double         bestScheduleValue;
  void           schedule_nodes(NodeSet s,
                                ScheduleVector schedule);
  double         get_schedule_value(ScheduleVector schedule);
};

// Ein Auftrag
class Job {
public:
  Job(std::string name);
  ~Job();

  std::string get_name();

  void add_node(Node* node);
private:
  std::string m_name;
  NodeSet nodes;
};

double calculate_l_max();
void   print_schedule(ScheduleVector schedule);
#endif
```

Listing von sbh.cc

```
#include "datamodell.hh"

MachineSet machines;
JobSet jobs;
NodeSet nodes;
Node   senke;
double due_date;

main() {
  std::cout << "Erzeuge Maschinen... " << std::endl;
  Machine m1("1"), m2("2"), m3("3"), m4("4");
  machines.insert(&m1);
  machines.insert(&m2);
  machines.insert(&m3);
  machines.insert(&m4);

  std::cout << "Erzeuge Auftr\"age... " << std::endl;
  Job     j1("1"), j2("2"), j3("3");
  jobs.insert(&j1);
  jobs.insert(&j2);
  jobs.insert(&j3);

  std::cout << "Erzeuge Knoten... " << std::endl;
  Node
    n11(m1,j1,10), n21(m2,j1,8), n31(m3,j1,4),
    n22(m2,j2, 8), n12(m1,j2,3), n42(m4,j2,5), n32(m3,j2,6),
    n13(m1,j3, 4), n23(m2,j3,7), n43(m4,j3,3);
  nodes.insert(&n11); nodes.insert(&n21);  nodes.insert(&n31);

  nodes.insert(&n22); nodes.insert(&n12);  nodes.insert(&n42);
  nodes.insert(&n32);

  nodes.insert(&n13); nodes.insert(&n23);  nodes.insert(&n43);

  std::cout << "Baue Graph auf... " << std::endl;

  // erster Auftrag
  n11.insert_arc_to(n21); n21.insert_arc_to(n31);
  n31.insert_arc_to(senke);

  // zweiter Auftrag
  n22.insert_arc_to(n12); n12.insert_arc_to(n42);
  n42.insert_arc_to(n32); n32.insert_arc_to(senke);

  // dritter Auftrag
  n13.insert_arc_to(n23); n23.insert_arc_to(n43);
  n43.insert_arc_to(senke);
```

```
due_date = senke.get_completion_date();
std::cout << "Due Date ist " << due_date << std::endl;

MachineSet M_0;
int num_iteration = 0;

do {
  num_iteration++;
  std::cout << std::endl;
  std::cout << num_iteration << ". Iteration" << std::endl;
  std::cout <<                  "============" << std::endl;

  MachineSet s; s.clear();
  set_difference(machines.begin(), machines.end(),
                 M_0.begin(), M_0.end(),
                 std::insert_iterator<MachineSet>
                   (s, s.begin())
                 );

  MachineSet::const_iterator m = s.begin();
  while (m != s.end()) {
    (*m)->schedule_machine();
    (*m)->print();
    m++;
  }

  Machine* k = NULL;
  MachineSet::const_reverse_iterator i = s.rbegin();
  while (i != s.rend()) {
    if (k == NULL) k = *i;
    else {
      if (k->get_criticaly() < (*i)->get_criticaly()) {
        k = *i;
      }
    }
    i++;
  }

  std::cout << "choosing machine " << k->get_name()
            << std::endl;
  k->insert_arcs(k->get_schedule());
  M_0.insert(k);

  due_date = senke.get_completion_date();
  std::cout << "New Completion Date: " << due_date
            << std::endl;
} while (M_0 != machines);
```

```
}

double calculate_l_max() {
  return (senke.get_completion_date() - due_date);
}

void print_schedule(ScheduleVector schedule) {
  ScheduleVector::const_iterator i = schedule.begin();

  while (i != schedule.end()) {
    std::cout << (*i)->j()->get_name() << ",";
    i++;
  }

  std::cout << std::endl;
}
```

Listing von node.cc:

```
#include "datamodell.hh"

Node::Node()
  :m_machine(NULL),
   m_job(NULL),
   m_processtime(0)
{
}

Node::Node(Machine& i, Job& j, double time)
  :m_machine(&i),
   m_job(&j),
   m_processtime(time)
{
  m_machine->add_node(this);
  m_job->add_node(this);
}

Node::~Node() {
}

Machine* Node::i() {
  return m_machine;
}

Job* Node::j() {
  return m_job;
}

double Node::get_completion_date() {
  double ret = get_release_date() + get_process_time();
  return ret;
}

double Node::get_release_date() {
  recalculate();
  return m_release_date;
}

double Node::get_process_time() {
  return m_processtime;
}

void Node::insert_arc_to(Node& node) {
  m_nextarcs.insert(&node);
  node.m_prevarcs.insert(this);
```

```
  node.set_recalculate_flag();
}

void Node::delete_arc_to(Node& node) {
  m_nextarcs.erase(&node);
  node.m_prevarcs.erase(this);

  node.set_recalculate_flag();
}

void Node::set_recalculate_flag() {
  if (!m_must_recalculate) {
    m_must_recalculate = true;

    NodeSet::const_iterator i = m_nextarcs.begin();
    while (i != m_nextarcs.end()) {
      (*i)->set_recalculate_flag();
      i++;
    }
  }
}

void Node::recalculate() {
  if (m_must_recalculate) {
    m_must_recalculate = false;
    m_release_date = 0;

    NodeSet::const_iterator i = m_prevarcs.begin();
    while (i != m_prevarcs.end()) {
      if (m_release_date < (*i)->get_completion_date()) {
        m_release_date = (*i)->get_completion_date();
      }

      i++;
    }
  }
}

void Node::print() {
  std::cout << "Node " << i()->get_name() << ", "
            << j()->get_name()
            << ":"
            << "Release Date = " << get_release_date()
            << "; "
            << "Processing Time = " << get_process_time()
            << "; "
            << "Completion Date = " << get_completion_date()
            << std::endl;
```

```
}
```

Listing von machine.cc:

```
#include "datamodell.hh"

Machine::Machine(std::string name)
  : m_name(name)
{
}

Machine::~Machine() {
}

std::string Machine::get_name() {
  return m_name;
}

void Machine::add_node(Node* node) {
  nodes.insert(node);
}


void Machine::schedule_machine() {
  bestScheduleValue = 60000;
  schedule_arcs.clear();

  ScheduleVector tempSchedule;
  schedule_nodes(nodes, tempSchedule);
}

double Machine::get_criticaly() {
  return bestScheduleValue;
}

void Machine::insert_arcs(ScheduleVector schedule) {
  if (schedule.empty()) return;

  ScheduleVector::const_iterator i = schedule.begin();
  Node* node = (*i); i++;
  while (i != schedule.end()) {
    node->insert_arc_to((**i));
    node = (*i);
    i++;
  }
}

void Machine::remove_arcs(ScheduleVector schedule) {
  if (schedule.empty()) return;

  ScheduleVector::const_iterator i = schedule.begin();
```

```
  Node* node = (*i); i++;
  while (i != schedule.end()) {
    node->delete_arc_to((**i));
    node = *i;
    i++;
  }
}

ScheduleVector Machine::get_schedule() {
  return schedule_arcs;
}

void Machine::print() {
  std::cout << "Machine " << get_name() << ": "
            << "Criticaly = " << get_criticaly() << "; "
            << "Schedule List = ";
  print_schedule(schedule_arcs);
}

void Machine::schedule_nodes(NodeSet s, ScheduleVector schedule)
{
  if (s.empty()) {
    double value = get_schedule_value(schedule);
    if (value < bestScheduleValue) {
      bestScheduleValue = value;
      schedule_arcs = schedule;
    }
  } else {
    NodeSet::const_iterator i = s.begin();
    while (i != s.end()) {
      schedule.push_back(*i);
      NodeSet t = s; t.erase(*i);
      schedule_nodes(t, schedule);
      schedule.pop_back();
      i++;
    }
  }
}

double Machine::get_schedule_value(ScheduleVector schedule)
{
  insert_arcs(schedule);
  double ret = calculate_l_max();
  remove_arcs(schedule);

  return ret;
}
```

Listing von job.cc:

```
#include "datamodell.hh"

Job::Job(std::string name)
  : m_name(name)
{
}

Job::~Job() {
}

std::string Job::get_name() {
  return m_name;
}

void Job::add_node(Node* node) {
  nodes.insert(node);
}
```