Tabusuche für Maschinenbelegungsprobleme

Diplomarbeit

zur Erlangung des akademischen Grades
„Magistra der Naturwissenschaften“
an der Formal- und Naturwissenschaftlichen Fakultät
der Universität Wien

eingereicht von

Ulrike Schneider

Wien, im April 1999.

Inhaltsverzeichnis

Kurzbeschreibung der Arbeit

Diese Arbeit entstand ausgehend vom Wunsch einer Firma, ihre Arbeitsabläufe zu *optimieren*: durch intelligente Zuweisung der Aufträge auf die verschiedenen Maschinen sollen Zeit und Kosten gespart werden.

An die Aufgabenstellung wurde folgendermaßen herangegangen:

Die Arbeitsabläufe wurden in einem Modell erfaßt, das es erlaubt, die Aufgabenstellung als *kombinatorisches Optimierungsproblem* zu formulieren:

Die Menge S enthält alle möglichen Zuweisungen der auszuführenden Aufträge an die Maschinen. Eine Kostenfunktion $f : S \to \mathbb{R}$ kann einen solchen Plan bewerten. Nun soll einfach der billigste Plan aus S gefunden – also die Kostenfunktion f über S minimiert werden.

Ein Computerprogramm wurde geschrieben, das dieses *Maschinenbelegungsproblem* mithilfe der *Tabusuche* – eines sog. *metaheuristischen Verfahrens* – behandelt. Dieser Algorithmus versucht, mit intelligenten Strategien einen Plan (oder *Punkt*) mit möglichst geringem Kostenwert aus der Menge S zu finden.

Dieses Programm kann also nach Eingabe einer Auftragsliste eine kostengünstige Zuteilung dieser Aufträge auf die Maschinen finden.

Die Arbeit ist folgendermaßen aufgebaut:

Kapitel 1 ist der mathematischen Modellierung des Problems gewidmet.

In Kapitel 2 wird allgemeiner auf Maschinenbelegungsprobleme eingegangen und ihre in der Literatur übliche Klassifikation gebracht. Die Komplexität solcher Probleme wird mit den bekannten Klassen $\mathcal{P}$ und $\mathcal{NP}$ dargestellt.

Schließlich werden in Kapitel 3 die heuristischen und metaheuristischen Suchmethoden beschrieben. Diese Verfahren dienen dazu, kombinatorische Optimierungsprobleme zu behandeln, für die es zu rechenaufwendig ist, mit Sicherheit ein globales Optimum zu finden. (Meta-) Heuristiken sind im allgemeinen Beispiele von *lokalen Nachbarschaftssuchen*: Ein solches Verfahren sucht – ausgehend von einem Anfangspunkt – den Suchraum S iterativ ab, indem jeweils ein Element s' aus $N(s) \subseteq S$, der *Nachbarschaft* des aktuellen Punktes s als neuer Punkt gewählt wird. Diese Methoden lösen zumindest ein

entsprechendes lokales Suchproblem, das man durch Einführen einer solchen Nachbarschaftumgebung aus dem ursprünglichen globalen Problem erhält.

Die einfachen, heuristischen Methoden finden im allgemeinen irgendein lokales Optimum – die komplexeren, metaheuristischen Verfahren (wie z.B. die Tabusuche) sollen *gute* lokale Optima aufspüren, die möglicherweise nahe am globalen liegen.

Beispiele solcher einfachen und erweiterten Nachbarschaftsmethoden werden umrissen. Diese unterscheiden sich jeweils in den Strategien, von einem Punkt zu einem benachbarten zu „ziehen“. Auf das Beispiel der Tabusuche, dem zentralen Thema dieser Arbeit, wird im speziellen eingegangen.

In Kapitel 4 wird die Komplexität von lokalen Suchproblemen erläutert und mit den Komplexitätsklassen aus Kapitel 2 in Bezug gebracht.

Kapitel 5 kommt auf das konkrete Problem zurück: Die Implementierung des Modells aus Kapitel 1 in ein C-Programm wird beschrieben und erklärt, wie ein solches Programm zu benützen ist.

In Kapitel 6 werden verschiedene Programmversionen und ihre Ergebnisse nach Eingabe einer (typischen) Auftragsliste präsentiert. Viele Strategien der Tabusuche wurden an die Modellierung angepaßt, implementiert, getestet und verändert. Es wird beschrieben, wie mit diesen Methoden die Ergebnisse verbessert werden konnten. Außerdem wird genau erläutert, wie der Rechenaufwand im Laufe der Entstehung der einzelnen Versionen drastisch verringert wurde.

Im letzten Kapitel sind mögliche Verbesserungsvorschläge sowohl für die Modellierung als auch den implementierten Suchalgorithmus dargelegt.

Die in der Arbeit beschriebenen Programme wurden in der höheren Programmiersprache C geschrieben und sind auf dem Rechnersystem der Gruppe für Computer Mathematik an der Universität Wien unter dem Verzeichnis `home/cma/uli/Ccodes` zu finden.

Kapitel 1

Problemstellung und Modellierung

1.1 Überblick

In diesem Kapitel wird die mathematische Formulierung des Problems der Kabelfirma vorgenommen.

Dazu wird erst auf die Aufgabenstellung – nämlich herauszufinden, auf welchen Maschinen und in welcher Reihenfolge die Kabelaufträge produziert werden sollen – genauer eingegangen und die Aufträge mit ihren unterschiedlichen Merkmalen charakterisiert.

Danach wird erläutert, wie man eine solche Zuweisung der Aufträge auf die Maschinen darstellen kann und damit der *Suchraum* definiert. Es wird genau untersucht, auf welche Weise die Reihenfolge der Aufträge auf den Maschinen Kosten verursacht und damit die *Zielfunktion* konstruiert.

Zuguterletzt wird das Problem als *kombinatorisches Optimierungsproblem* formuliert.

1.2 Problemstellung

Eine Firma will die Arbeitsabläufe bei der Herstellung von Kabeln optimieren: durch intelligente Zuweisung der Aufträge an die Maschinen sollen Kosten und Zeit gespart werden.

Den Hauptvorgang der Produktion bildet die Ummantelung der Drähte mit einer Isolationsschicht aus Kunststoff. Diese kann entweder ein- oder zweifärbig (*Kennstreifen*) angebracht werden. Außerdem werden manche Kabeln mit einer *Ringsignierung* versehen.
Zur Verfügung stehen m Maschinen $M_1, \ldots, M_m$. Ringsignierungen können

nur auf den Maschinen $M_{q_1}, \ldots, M_{q_s}$ ausgeführt werden, Kennstreifen prinzipiell auf allen Maschinen, wobei auf M_1 und M_3 gewisse Vorrichtungen (UCCS) es ermöglichen, ohne Produktionsstopp eine neue sog. Kennfarbe einzustellen.

Die Firma erhält laufend neue Aufträge, formal wird aber von einer *Auftragsliste* mit durchnumerierten Aufträgen ausgegangen.

1.3 Auftragscharakteristik

Eine Auftragsbestellung hat folgende relevante Charakteristika:

- Eine *Laufnummer* l (also eine Identifikationsnummer in der Bestellung),
- eine *Sachnummer* S_l, die bestimmt wird durch die *Auftragskomponenten*:
  - die Farbe der Ringsignierung r_l (*Ringfarbe*),
  - die Kunststoffart der Isolation m_l (*Massetyp*),
  - den Querschnitt der Litze q_l (*Querschnitt*),
  - die Farbe der Isolierung g_l (*Grundfarbe*) und
  - die Farbe des Kennstreifen b_l (*Kennfarbe*).

  (Zwei Aufträge l_1, l_2 haben also die gleiche Sachnummer S, falls gilt, daß $r_{l_1} = r_{l_2}$, $m_{l_1} = m_{l_2}$, $q_{l_1} = q_{l_2}$, $g_{l_1} = g_{l_2}$ und $b_{l_1} = b_{l_2}$ ist.)

  Darüberhinaus gibt es die
- *farbneutrale Sachnummer* f_l, die festgelegt wird durch den Querschnitt, den Massetyp und dadurch, ob der Auftrag eine Ringsignierung und (oder) einen Kennstreifen beinhaltet. Sie ist aber unabhängig von der eventuellen Ring- bzw. Kennfarbe. Zuletzt ist noch
- die *Länge* L_l des bestellten Kabels von Bedeutung.

Falls bei einem Auftrag l kein Kennstreifen (keine Ringsignierung) vorgesehen ist, so wird $b_l = 0$ ($r_l = 0$) gesetzt. Für zwei Aufträge l_1 und l_2 gilt also $f_{l_1} = f_{l_2}$, wenn $m_{l_1} = m_{l_2}$, $q_{l_1} = q_{l_2}$ ist. Außerdem müssen l_1 und l_2 jeweils entweder beide oder keiner einen Kennstreifen, bzw. eine Ringsignierung enthalten.

Weiters gibt es für jeden Auftrag l eine Rezeptnummer $R(i, f_l)$, die durch die farbneutrale Sachnummer und die Maschine M_i, auf der der Auftrag ausgeführt werden soll, festgelegt wird. Die Rezeptnummer eines Auftrags, zusammen mit der Maschine, bestimmt Daten wie Produktionsgeschwindigkeit, Anfahrzeit, usw. (Sie ist also erst dann bekannt, wenn der Auftrag l bereits auf eine bestimmte Maschine zugewiesen wurde.) $R(i, f)$ ist tabelliert.

Nach Beenden des Arbeitsvorgangs für einen bestimmten Auftrag muß für den darauffolgenden möglicherweise eine der 5 Auftragskomponenten Grundfarbe, Querschnitt, Massetyp, Kennfarbe oder Ringfarbe geändert werden (*Umrüsten*). Dies bedeutet Maschinenstopp und daher Zeitverlust (und in weiterer Folge höhere Maschinen- und Lohnkosten) und vor allem Abfall (höhere Materialkosten). Bis zu einem gewissen Maß können große Umrüstzeiten verhindert werden, indem man Aufträge, die in bestimmen Auftragskomponenten übereinstimmen, hintereinander ausführt.

Gefragt ist also, auf welchen Maschinen und in welcher Reihenfolge die Bestellungen produziert werden sollen. Wie eine solche gültige Zuweisung aussieht, wird im folgenden Abschnitt festgelegt.

1.4 Suchraum

Bei gegebener Auftragsliste mit r Aufträgen ist eine *zulässige Belegung* eine Belegung von Variablen $job(i, k)$ mit den Laufnummern aus der Auftragsliste, wobei $i = 1, \ldots, m$, $k = 1, \ldots, n_i$ und $\sum_{i=1}^{m} n_i = r$. $job(i, k)$ bezeichnet den $k$ten Auftrag auf der $i$ten Maschine. Jeder Auftrag muß genau einmal vorkommen, außerdem dürfen Aufträge mit Ringsignierung ($r_{q_t} \neq 0$ für $t = 1, \ldots, s$) nur auf $job(q_1, .), \ldots, job(q_s, .)$ zugewiesen werden.
Die Maschinen werden also mit den vorhandenen Aufträgen „angefüllt“. (Siehe Abb. 1.1)

Die Menge S aller zulässigen Belegungen für eine bestimmte Auftragsliste heißt *Suchraum*.

Die Idee dieser Darstellung wurde teilweise aus Schaerf [14] entnommen. In diesem Artikel wird ein Stundenplanproblem eines Gymnasiums beschrieben, das ebenfalls mithilfe einer Tabusuche gelöst wird. Bei einem Stundenplan entsprechen den Maschinen die Lehrer, die mit den entsprechenden Aufträgen (Stunden, die sie in verschiedenen Klassen halten müssen) angefüllt werden.

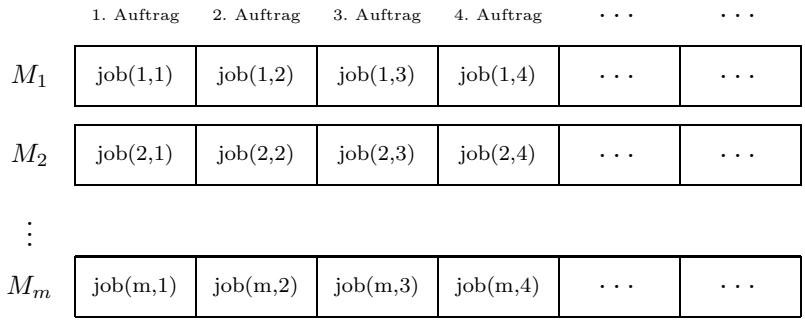


Abbildung 1.1: Eine Maschinenbelegung

1.5 Zielfunktion

Die Zielfunktion bewertet die „Güte“ einer zulässigen Belegung, indem sie die Kosten der Aufträge berechnet, ausgeführt in der von der Belegung festgesetzten Reihenfolge. In groben Zügen sieht sie folgendermaßen aus:

$$\sum_{\text{Aufträge}} \begin{array}{l} [\text{Abfallkosten} \quad + \\ (\text{Anfahrkosten} \times \text{Anfahrzeit}) \quad + \\ (\text{Umrüstkosten} \times \text{Umrüstzeit}) \quad + \\ (\text{Produktionskosten} \times \text{Produktionszeit})] \end{array}$$

1.5.1 Zeiten

Die Dauer p_i^k eines Auftrags $job(i,k)$ setzt sich zusammen aus:
Umrüstzeit + Anfahrzeit + reine Produktionszeit.

Umrüstzeit

Während der *Umrüstzeit* werden eventuelle Wechsel bei den Auftragskomponenten vorgenommen. Bis auf eine Ausnahme (die Änderung der Grundfarbe) werden diese Änderungen hintereinander ausgeführt. Die Maschine muß angehalten werden.

- *Umrüsten auf neue Ringfarbe*
  $u_1(i,k) = T_{u_1}(i, r_{job(i,k-1)}, r_{job(i,k)}) \ldots$ bezeichnet die benötigte Zeit, um auf Maschine M_i von der Ringfarbe des Auftrags $job(i,k-1)$ (dem

vorangegangenen Auftrag) auf die Ringfarbe des Auftrags $job(i,k)$ umzurüsten.
$T_{u_1}(i,r,r')$ ist tabelliert.

- *Umrüsten auf neuen Massetyp*
  $u_2(i,k) = T_{u_2}(i, m_{job(i,k-1)}, m_{job(i,k)}) \ldots$ bezeichnet die Zeit, um auf Maschine M_i vom Massetyp des vorangegangenen Auftrags auf den aktuellen Massetyp umzurüsten.
  $T_{u_2}(i,m,m')$ ist tabelliert.

- *Umrüsten auf neuen Querschnitt*
  $u_3(i,k) = T_{u_3}(i, q_{job(i,k-1)}, q_{job(i,k)}) \ldots$ bezeichnet die Zeit, um auf Maschine M_i vom Querschnitt des vorangegangenen Auftrags auf den aktuellen Querschnitt umzurüsten.
  $T_{u_3}(i,q,q')$ ist tabelliert.

- *Umrüsten auf neue Grundfarbe*
  $T_{u_4}(i, g_{job(i,k-1)}, g_{job(i,k)}) \ldots$ bezeichnet die Zeit, um auf Maschine M_i von der Grundfarbe des vorangegangenen Auftrags auf die aktuelle Grundfarbe umzurüsten. Wenn allerdings bereits auf einen neuen Massetyp gewechselt werden muß, kann diese Zeit vernachlässigt werden. (d.h. es ist

  $$u_4(i,k) = \begin{cases} T_{u_4}(i, g_{job(i,k-1)}, g_{job(i,k)}) & \text{falls } m_{job(i,k-1)} = m_{job(i,k)}, \\ 0 & \text{sonst.} \end{cases}$$

  $T_{u_4}(i,g,g')$ ist tabelliert.

- *Umrüsten auf neue Kennfarbe*
  $u_5(i,k) = T_{u_5}(i, b_{job(i,k-1)}, b_{job(i,k)}) \ldots$ bezeichnet die Zeit, um auf Maschine M_i von der Kennfarbe des vorangegangenen Auftrags auf die aktuelle umzurüsten. Bei Maschinen mit UCCS ist dies ohne Stopp möglich, deshalb gilt $u_5(1,k) = u_5(3,k) = 0$ für alle k.
  $T_{u_5}(i,b,b')$ ist tabelliert.

Die einzelnen u_j $(j = 1, \ldots, 5)$ sind also abhängig von der Maschine M_i, auf der der Auftrag produziert wird, von der jeweiligen Auftragskomponente des aktuellen Auftrags und von derselben des vorangegangenen Auftrags.

Die Umrüstzeit des Auftrags $job(i,k)$ läßt sich also schreiben als

$$t_u(i,k) := \sum_{j=1}^{5} u_j(i,k).$$

Anfahrzeit

Die *Anfahrzeit* beschreibt die Zeit, in der die Maschine bereits läuft, aber noch kein brauchbares Kabel (noch keine *Gutware*) produziert wird. (So muß beispielsweise gewartet werden, bis die Isolationsschicht die gewünschte Farbe hat, wenn davor ein Kabel mit anderer Grundfarbe produziert wurde). Die Anfahrzeit von $job(i,k)$ ist tabelliert in $T_{t_a}(i, R, R')$, wobei i die Maschine M_i bezeichnet, R die Rezeptnummer des aktuellen und R' die des vorangegangenen Auftrags ist:

$$t_a(i,k) := T_{t_a}(i, R(i, f_{job(i,k-1)}), R(i, f_{job(i,k)})).$$

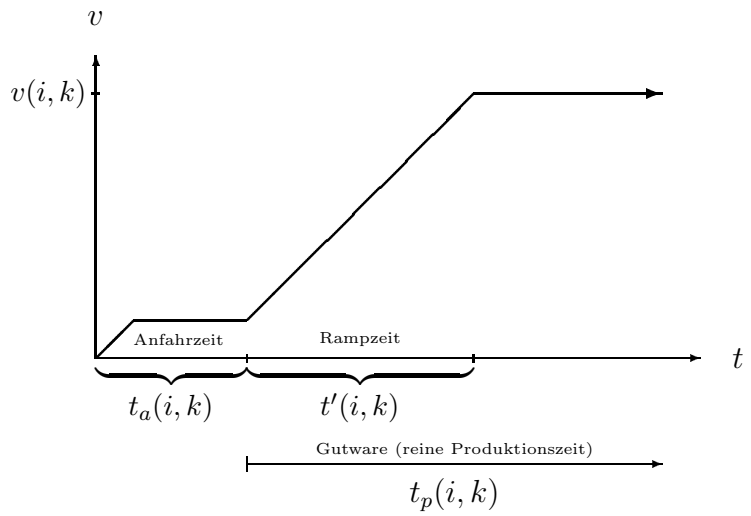
Reine Produktionszeit

In der *reinen Produktionszeit* eines Auftrags wird das eigentliche Kabel, die Gutware produziert. Diese Zeit ist abhängig von der Maschine M_i, der Rezeptnummer R und der Länge L des bestellten Kabels.

Während der Anfahrzeit ist die Geschwindigkeit der Maschine relativ gering. Sobald aber Gutware produziert wird, erhöht man auf volle Produktionsgeschwindigkeit.

In den Tabellen $T_l(i,R)$, $T_{t'}(i,R)$ und $T_v(i,R)$ stehen die benötigten Daten zur Bestimmung der reinen Produktionszeit. $l(i,k) := T_l\,(i, R(i, f_{job(i,k)}))$ bezeichnet die Länge des Kabels, das während des Hochfahrens auf volle Produktionsgeschwindigkeit erzeugt wird (die sog. *Rampläng*e). Der Wert $t'(i,k) := T_{t'}(i, R(i, f_{job(i,k)}))$ ist die Zeit, in der bei geringerer Geschwindigkeit Gutware produziert wird (*Rampzeit*), und $v(i,k) := T_v\,(i, R(i, f_{job(i,k)}))$ bezeichnet die volle Produktionsgeschwindigkeit.

Die folgende Skizze soll den oben beschriebenen Vorgang veranschaulichen (auf die x-Achse ist die Zeit t, auf der y-Achse die Maschinengeschwindigkeit v aufgetragen):



Vorerst kann die *Abrampzeit* (in der die Maschine wieder zum Stillstand gebracht wird) vernachlässigt werden. Sie fällt, verglichen mit der relativ langen reinen Produktionszeit, kaum ins Gewicht.

Als reine Produktionszeit des Auftrags $job(i,k)$ ergibt sich also:

$$t_p(i,k) := \frac{L_{job(i,k)} - l(i,k)}{v(i,k)} \; + \; t'(i,k).$$

Die Dauer eines Auftrags

Mit diesen Bezeichnungen kann man die gesamte Dauer des Auftrags $job(i,k)$ schreiben als:

$$p_k^i := t_u(i,k) + t_a(i,k) + t_p(i,k).$$

1.5.2 Kosten

Die jeweiligen Kosten sind unabhängig von der Art des Kabels, das produziert wird.

Abfallkosten

Die *Abfallkosten* c sind Fixkosten, die pro Auftrag anfallen.

Umrüstkosten

Unter den *Umrüstkosten* $c_u(i) := T_{c_u}(i)$ versteht man die Kosten, die während der Umrüstzeit anfallen. Sie beinhalten vor allem Lohnkosten, da die Umrüstvorgänge bei angehaltenen Maschinen vom Personal selbst vorgenommen werden müssen.

Anfahrkosten

Die *Anfahrkosten* $c_a(i) := T_{c_a}(i)$ spiegeln hauptsächlich den Materialverlust während der Anfahrzeit wieder (in der ja bereits Rohmaterial verbraucht, aber noch keine Gutware hergestellt wird).

Produktionskosten

Mit den *Produktionskosten* $c_p(i) := T_{c_p}(i)$ sind die Maschinenkosten gemeint, die in der reinen Produktionszeit anfallen.

c_u, c_a und c_p sind als Kosten pro Minute zu verstehen.

1.5.3 Zielfunktion

Die Zielfunktion $f : S \to \mathbb{R}$ sieht nun folgendermaßen aus:

$$f(\text{Belegung}) = cr + \sum_{i=1}^{m} \sum_{k=1}^{n_i} \left[c_a(i)\, t_a(i,k) + c_u(i)\, t_u(i,k) + c_p(i)\, t_p(i,k) \right]$$

Damit ist das Problem jetzt ein *kombinatorisches Optimierungsproblem* (mit diskretem Suchraum) und läßt sich schreiben als:

$$\text{finde } s_0 \in S, \text{ sodaß } f(s_0) = \min_{s \in S} \{f(s)\}.$$

Kapitel 2

Klassifikation und Komplexität

2.1 Überblick

In diesem Kapitel werden Maschinenbelegungsprobleme als Optimierungsfragen in einem allgemeineren Rahmen beschrieben und die in der Literatur übliche Klassifikation umrissen. Eine Einordnung des konkreten Problems aus Kapitel 1 wird angedeutet.

Weiters werden die bekannten Klassen $\mathcal{P}$ und $\mathcal{NP}$ eingeführt und ihre Komplexität beschrieben. Die Komplexität von Optimierungsproblemen wird untersucht, indem die zugehörigen Entscheidungprobleme in $\mathcal{P}$ und $\mathcal{NP}$ eingeordet werden.

2.2 Scheduling

Die Behandlung von Maschinenbelegungsproblemen wird im Englischen *scheduling* genannt. Die folgende Beschreibung stammt aus Pinedo [12].

Definition 1 *"Scheduling" befaßt sich mit der Zuordnung von begrenzten Ressourcen zu verschiedenen Aufgaben während bestimmten Zeitspannen. Scheduling ist ein Entscheidungsprozeß, der die Optimierung einer oder mehrerer Funktionen zum Ziel hat.*

Da die Ressourcen meist Maschinen sind, spricht man von *Maschinenbelegungsproblemen.*

2.3 Klassifikation

Die weitgehend übliche Klassifikation von Maschinenbelegungsproblemen aus Brucker [11] soll nun umrissen werden. Diese deckt sich weitgehend mit der

Klassifikation in [12].

2.3.1 Das Problem

Ein Maschinenbelegungsproblem ist ein Problem der folgenden Art: Auf m Maschinen $M_1, \ldots, M_m$ sollen n Aufträge $J_1, \ldots, J_n$ produziert werden. Eine Belegung (*schedule*) ist eine Zuweisung der Aufträge auf die Maschinen. Zusätzlich gibt es eine Zielfunktion f, die die Güte einer Belegung bewertet.

Wie schon am konkreten Problem in 1.4 und 1.5 beschrieben, ist ein Maschinenbelegungsproblem ein kombinatorisches Optimierungsproblem.

2.3.2 Auftragscharakteristik

Unterschieden werden sechs Auftragsmerkmale, die hier mit ihrer üblichen englischen Bezeichnung angeführt werden.

1. *preemption*:
   Ist es erlaubt, Aufträge zu unterbrechen und zu einem späteren Zeitpunkt – möglicherweise auf einer anderen Maschine – fertigzuproduzieren?

2. *precedence relations*:
   Haben manche Aufträge höhere Priorität als andere, d.h. gibt es Aufträge, die erst in Angriff genommen werden dürfen, wenn andere fertiggestellt sind?
   Solche Relationen werden durch einen Graph $G = (V, A)$ dargestellt, wobei die Knoten $V = \{1, \ldots, n\}$ die Aufträge repräsentieren und $(i, k) \in A$ ist, falls J_i fertig ausgeführt sein muß, bevor die Produktion von J_k beginnt ($J_i \rightarrow J_k$). Hat der Graph spezielle Gestalt, so werden noch feinere Unterscheidungen getroffen.

3. *release dates*:
   Können manche Aufträge erst ab einem gewissen Zeitpunkt ausgeführt werden?

4. *processing times*:
   Haben die Aufträge einheitliche oder unterschiedliche Länge?

5. *deadlines*:
   Müssen die Aufträge zu bestimmten Zeiten fertiggestellt sein?

6. *batching*:
   Gibt es Aufträge, die hintereinander auf der derselben Maschine (in

einer Charge oder engl. *batch*) produziert werden sollen?
Üblicherweise gibt es zwei Typen von Chargen: beim ersten ist die Länge einer Charge die Summe der Länge der enthaltenen Aufträge (*s-batching*). Die Länge des zweiten Typs ist die Länge des „längsten " enthaltenen Auftrags (*p-batching*).

2.3.3 Maschinenumgebung

Bei den Maschinen werden drei Arten unterschieden:

1. parallele Maschinen:

   Jeder Auftrag kann prinzipiell auf jeder Maschine produziert werden. Bei identischen Maschinen (*identical parallel machines*) gibt es eine einheitliche Produktionsgeschwindigkeit. Liegen gleichmäßige Maschinen (*uniform parallel machines*) vor, so ist die Produktionsgeschwindigkeit maschinenabhängig. Bei unabhängigen Maschinen (*unrelated parallel machines*) ist für jeden Auftrag und jede Maschine eine eigene Produktionsgeschwindigkeit angegeben.

2. Mehrzweckmaschinen (*multi-purpose-machines*):

   Gibt es für jeden Auftrag bestimmte Maschinen, auf denen er ausgeführt werden muß, so spricht man von Mehrzweckmaschinen. Analog zu 1 werden identische und gleichmäßige Mehrzweckmaschinen unterschieden.

3. Maschinen für einzelne Operationen (*multi-operation model*):

   Bei diesem Modell besteht jeder Auftrag J_i aus k_i Operationen $O_{i1}, \ldots,$ O_{i,k_i}. Die Operation O_{ij} muß auf der Maschine $M_{\mu_{ij}}$ ausgeführt werden. Dieser allgemeine Fall wird *general shop* genannt. Folgende Spezialfälle werden unterschieden:

   Beim *job shop* gibt es spezielle Prioritätsvorschriften der Form $O_{i1} \rightarrow O_{i2} \rightarrow \cdots \rightarrow O_{i,k_i}$. Außerdem wird angenommen, daß $\mu_{ij} \neq \mu_{i,j+1}$ ist. Falls $\mu_{ij} = \mu_{i,j+1}$ vorkommt, wird dies *job shop mit Maschinenwiederholung* (*job shop with machine repetition*) genannt.

   Der *flow shop* ist ein Spezialfall des job shop, wo $k_i = m$ und $\mu_{ij} = j$ für $i = 1, \ldots, m$ und $j = 1, \ldots, k_i$ ist.

   Der *open shop* ist wie der flow shop definiert, allerdings gibt es keine Prioritätsvorschriften zwischen den einzelnen Operationen.

   Liegt eine Kombination zwischen dem job shop und dem open shop vor, so spricht man von einem *mixed shop*.

Ein *permutation flow shop* ist ein flow shop, bei dem die Aufträge auf jeder Maschine in der gleichen Reihenfolge produziert werden müssen. D.h. auf jeder Maschine M_i gelten die Prioritätsvorschriften $O_{1i} \rightarrow O_{2i} \rightarrow \cdots \rightarrow O_{k_i,i}$.

2.3.4 Optimalitätskriterien

Bei den allgemein behandelten Problemen in der Literatur ist die Zielfunktion auf folgende Weise eingeschränkt:

Sei C_i der Zeitpunkt der Fertigstellung von J_i und $f_i(C_i)$ die entsprechende Kostenfunktion von J_i. Die einzelnen Funktionen können auf zwei Arten in der Zielfunktion vorkommen:

1. *bottleneck objective*

$$f_{max}(C) := \max\{f_i(C_i)|i = 1, \ldots, n\}$$

2. *sum objective*

$$\sum f_i(C_i) := \sum_{i=1}^{n} f_i(C_i)$$

Die am häufigsten verwendeten Zielfunktionen sind die folgenden:

- *makespan*: $\max\{C_i|i = 1 \ldots, n\}$,
- *total flow time*: $\sum_{i=1}^{n} C_i$ und
- *weighted flow time*: $\sum_{i=1}^{n} \omega_i C_i$.

2.3.5 Theorie und Praxis

Probleme in der Praxis treten natürlich kaum in genau einem der oben beschriebenen Fälle auf. Auch die in Kapitel 1 ausgeführte Modellierung ist auf den ersten Blick damit kaum zu klassifizieren. Manches soll trotzdem dazu gesagt werden.

Das konkrete Problem

Der Maschinenumgebung entsprächen am ehesten gleichmäßige Mehrzweckmaschinen oder – wenn man vernachlässigt, daß manche Aufträge nur auf bestimmten Maschinen produziert werden können – parallele, unabhängige Maschinen.

Für jeden Auftrag ist eine eigene Auftragslänge angegeben.

Die Aufträge, die sich nur in der Länge unterscheiden, sollten in einer Charge produziert werden (s-batching).

Von den vorgegebenen Optimalitätskriterien kann keines die in 1.5 konstruierte Zielfunktion annähernd charakterisieren. Hier wäre mehr Arbeit nötig, um vielleicht manche Kriterien aus der Zielfunktion in andere, in der Klassifikation erlaubte Bedingungen zu verpacken.

2.4 Komplexität von Berechnungsproblemen

Es soll nun genauer definiert werden, wann Probleme „leicht“ oder „schwer“ zu lösen sind. Dazu werden die Klassen $\mathcal{P}$ und $\mathcal{NP}$ und die Begriffe $\mathcal{NP}$-hart und $\mathcal{NP}$-vollständig eingeführt und ein Beispiel gebracht. Für die aus Brucker [11] stammende Darstellung müssen vorerst einige Definitionen gebracht werden:

2.4.1 Berechnungsprobleme

Ein Berechnungsproblem P (oder kurz Problem, engl. *computational problem*) hat Definitionsbereich D_P und für jede *Instanz* $x \in D_P$ gibt es eine Wertemenge $O_P(x)$ als mögliche Antworten. Ein Algorithmus löst P, wenn er auf den Input x ein $y \in O_P(x)$ ausgibt. Beim allgemeinen Fall eines Berechnungsproblems, dem *Suchproblem*, kann $O_P(x)$ entweder kein, ein oder mehrere Elemente enthalten. Ist $O_P(x) \leq 1$ für alle $x \in D_P$, so ist das Problem *funktional*. Gilt für alle $x \in D_P$, daß $O_P(x) \neq \emptyset$ ist, so spricht man von einem *totalen Problem*.

Für einen Input x soll $|x|$ die „Länge“ (der Codierung) von x repräsentieren. (Im Modell von Kapitel 1 kann das z.B. die Anzahl der Aufträge sein).

$T(n)$ soll eine obere Schranke (der sog. *worst-case*) für die Anzahl der Operationen sein, die der Algorithmus auf einen Input mit der Länge n durchführt. $T(n)$ wird selten genau angegeben, sondern meist in der bekannten Schreibweise als $O(g(n))$ geschrieben. Es ist $T(n) \in O(g(n))$, falls es eine Konstante $C > 0$ und ein $n_0 \in \mathbb{N}$ gibt, so daß $T(n) \leq Cg(n)$ für alle $n \geq n_0$ ist.

Mit T soll die Komplexität eines Algorithmus und des entsprechenden Problems gemessen werden.

Entscheidungsprobleme

Ein *Entscheidungsproblem* ist ein funktionales und totales Problem, bei dem der Wertebereich entweder gleich {yes} oder {no} ist. P kann dann auch als Funktion $h : D_P \to \{yes, no\}$ aufgefaßt werden. Der entsprechende Algorithmus berechnet also die Funktion h.

Optimierungsprobleme als Berechnungsprobleme

In der Sichtweise von oben ist ein Optimierungsproblem ein totales Berechnungsproblem P, bei dem für jede Instanz x ein Suchraum $S(x)$ und eine Kosten- oder Zielfunktion $f_x : S(x) \to \mathbb{R}$ definiert ist. Die Menge $O(x)$ enthält die globalen Minima von $S(x)$ bezüglich f_x.

Im Maschinenbelegungsproblem von Kapitel 1 ist also eine Instanz x eine Auftragsliste und der Suchraum $S(x)$ die Menge der zulässigen Maschinenbelegungen mit den Aufträgen aus der Liste.

Optimierungsprobleme als Entscheidungsprobleme

Zu einem Optimierungsproblem formuliert man ein entsprechendes Entscheidungsproblem: Für jede Schranke k der Zielfunktion f kann man fragen, ob es eine zulässige Belegung s gibt, so daß $f(s) \leq k$ ist.

Dieses Problem ist natürlich leicht zu lösen, wenn ein s_0 mit $f(s_0) \leq k$ bekannt ist. Eine solche Information nennt man *Zertifikat*.

Gibt es kein solches s_o, so ist dies schon schwieriger zu verifizieren. Brucker [11] spricht deshalb von einer Asymmetrie zwischen dem „yes“ und „no“-Input.

2.4.2 Komplexitätsklassen

Die Klasse $\mathcal{P}$

Ein Problem P heißt *polynomial lösbar*, wenn es für jeden Input x ein $k \in \mathbb{N}$ gibt, so daß $T(|x|) \in O(|x|^k)$ ist – d.h. wenn ein Algorithmus gefunden wurde, der P in *polynomialer Zeit* löst.

Eine wichtige Klasse von polynomial lösbaren Problemen sind z.B. ganzzahlige lineare Programme mit einer fixen Anzahl von Variablen (siehe Lenstra [16]).

Die Klasse $\mathcal{P}$ besteht nun aus allen Entscheidungsproblemen P, die polynomial lösbar sind.

Für viele Probleme wurde (noch?) kein polynomialer Algorithmus gefunden, was nicht notwendigerweise bedeutet, daß es keinen solchen geben kann. Solche Probleme werden mit der folgenden Klasse charakterisiert:

Die Klasse $\mathcal{NP}$

$\mathcal{NP}$ besteht aus allen Entscheidungsproblemen P, für die es für jeden "yes"-Input ein Zertifikat y gibt, so daß $|y|$ durch ein Polynom in $|x|$ beschränkt ist und außerdem ein polynomialer Algorithmus bekannt ist, der verifiziert, daß y ein gültiges Zertifikat ist.

$\mathcal{P} = \mathcal{NP}$?

Da für jedes polynomial lösbare Entscheidungsproblem die Antwort $h(x)$ ein Zertifikat ist, gilt offensichtlich $\mathcal{P} \subseteq \mathcal{NP}$.

Eine zentrale Frage der Komplexitätstheorie ist, ob $\mathcal{P} = \mathcal{NP}$ ist. Vom praktischen Standpunkt aus gilt es als „unwahrscheinlich“ daß es für Probleme aus $\mathcal{NP}$ polynomiale Algorithmen gibt. Trotzdem ist diese Frage noch ungelöst. Mit den folgenden Begriffen kann sie umschrieben werden:

Reduktion

Sind P und Q zwei Entscheidungsprobleme, so ist P *auf* Q *reduzierbar* (oft: $P \propto Q$), wenn es eine polynomial berechenbare Funktion g gibt, die jeden Input von P in einen Input von Q umwandelt, so daß x ein "yes"-Input von P genau dann ist, wenn $g(x)$ ein "yes"-Input von Q ist.

Im Prinzip bedeutet dies, daß ein Algorithmus, der Q löst, (mit höchstens polynomialem Mehraufwand) auch P lösen kann.

$\mathcal{NP}$-vollständig

Ein Entscheidungsproblem $P \in \mathcal{NP}$ heißt $\mathcal{NP}$-*vollständig*, wenn jedes Problem aus $\mathcal{NP}$ auf P reduzierbar ist.

Würde man also einen polynomialen Algorithmus für ein $\mathcal{NP}$-vollständiges Problem finden, so würde das $\mathcal{P} = \mathcal{NP}$ implizieren.

$\mathcal{NP}$-hart

Ein Optimierungsproblem wird $\mathcal{NP}$-hart genannt, wenn das zugehörige Entscheidungsproblem $\mathcal{NP}$-vollständig ist.

Hier gilt ebenfalls, daß die Existenz eines polynomialen Algorithmus für ein $\mathcal{NP}$-hartes Problem $\mathcal{P} = \mathcal{NP}$ bedeuten würde.

Ein Beispiel

Eines der bekanntesten $\mathcal{NP}$-harten Probleme ist das Problem des Handlungsreisenden, das im nächsten Kapitel in 3.2.1 genauer formuliert wird. Siehe auch Brucker [11].

2.5 Komplexität von Maschinenbelegungsproblemen

Da man sich mehr oder weniger mit der Existenz von $\mathcal{NP}$-harten Problemen abfinden muß, ist die Grenze interessant, ab wann ein Problem von einem polynomial lösbaren zu einem $\mathcal{NP}$-harten wird.

In [11] gibt es eine umfangreiche Darstellung vieler Probleme aus der in 2.3 beschriebenen Klassifikation mit Angaben über ihre Komplexität und konkreten Algorithmen.

Eine solche besonders aktuelle Auflistung findet man auch im Internet unter Brucker & Knust [15].

2.5.1 Beispiele

Wann konkrete Probleme die Grenze zu $\mathcal{NP}$-harten überschreiten, läßt sich natürlich nicht allgemein sagen. Oft bringen aber bestimmte Optimalitätskriterien oder Vorschriften über den Ablauf der Aufträge eine höhere Komplexität mit sich.

Für das Problem mit identischen, parallelen Maschinen, möglicher Auftragsteilung (preemption) und der Handelspanne (makespan) als Zielfunktion gibt es einen polynomialen Algorithmus. Sind zusätzlich Prioritätsvorschriften (precendence relations) angegeben, so ist das Problem bereits $\mathcal{NP}$-hart. Ebenso komplex ist es, wenn man im polynomial lösbaren Problem als Optimalitätskriterium $\sum \omega_i C_i$ (total weighted flow time) annimmt. Das Problem bleibt dann sogar ohne Auftragsteilung und Prioritätsvorschriften $\mathcal{NP}$-hart.

2.5.2 $\mathcal{NP}$-hart – Was nun?

$\mathcal{NP}$-harte Probleme exakt zu lösen, verlangt also – zumindest solange nicht $\mathcal{P} = \mathcal{NP}$ bewiesen wurde – in den ungünstigsten Fällen exponentiellen Rechenaufwand. Viele praktische Probleme werden daher mit heuristischen statt exakter Methoden gelöst, welchen Kapitel 3 gewidmet ist.

Kapitel 3

(Meta-)Heuristische Suchmethoden

3.1 Überblick

In diesem Kapitel werden heuristische und metaheuristische Suchmethoden beschrieben, die eine Möglichkeit bieten, kombinatorische Optimierungsprobleme zu behandeln.

Damit soll das zentrale Thema dieser Arbeit, das metaheuristische Verfahren der *Tabusuche*, in einem allgemeinen Kontext motiviert werden.

Zuerst wird der bereits in 1.5.3 erwähnte Begriff eines *kombinatorischen Optimierungsproblems* definiert und ein bekanntes Beispiel gebracht.

Danach wird erklärt, was eine *lokale Nachbarschaftssuche* ist und ein Gerüst eines solchen Algorithmus skizziert.

Es werden Beispiele einfacher heuristischer Verfahren erwähnt, und die metaheuristischen Methoden *simulated annealing* und *genetische Algorithmen* umrissen.

Die *Tabusuche* wird schließlich als weiteres Beispiel solcher Methoden eingeführt. Dazu wird erst eine besonders einfache Version für das Gerüst Verfahrens herangezogen, bevor in 3.5.6 die Grundideen und wichtigsten Elemente genauer erläutert werden.

3.2 Einleitung

3.2.1 Kombinatorische Optimierungsprobleme

Ein kombinatorisches Optimierungsproblem ist ein Problem der folgenden Art:

Definition 2 *Sei S ein diskreter Suchraum, $f : S \to \mathbb{R}$ die Zielfunktion. Gesucht ist*

$$\min \{f(s) \mid s \in S\}.$$

Die $s \in S$ heißen **zulässige Punkte** *oder* **Lösungen**.

Ein Beispiel

Ein bekanntes Beispiel eines kombinatorischen Optimierungsproblems ist das sog. *Problem des Handlungsreisenden* (engl. *travelling salesman problem* oder kurz *TSP*). Es sollen n Städte $\{1, \ldots, n\}$ besucht werden. c_{ij} $(i, j = 1, \ldots, n)$ sind die Kosten, um von Stadt i nach Stadt j zu fahren. (Ist die Matrix $C = c_{ij}$ symmetrisch, so spricht man von einem *symmetrischen TSP*). Gesucht ist eine „billigste“ Route, bei der jede Stadt genau einmal besucht wird und die zum Schluß zum Ausgangspunkt zurückführt.

Der Suchraum $S = \{\Pi = \{\pi_1, \ldots \pi_n\} \mid \Pi \text{ ist Permutation von } \{1, \ldots, n\}\}$ und die Zielfunktion $f : S \to \mathbb{N}$ ist

$$f(\Pi) = \sum_{i=1}^{n-1} c_{\pi_i \pi_{i+1}} + c_{\pi_n \pi_1}.$$

3.2.2 Heuristische Methoden

Da S meist sogar endlich ist, ist das Problem theoretisch einfach durch Aufzählen und Auswerten aller $s \in S$ zu lösen (engl. *complete enumeration*). Praktisch bedeutet das aber oft exponentiellen Rechenaufwand. Gefragt sind also Methoden, die in angemessenem Zeitaufwand möglichst gute Lösungen finden, die zumindest nahe am Optimum liegen. Mit heuristischen Strategien hofft man, den Suchraum weitgehend zu erkunden, ohne jeden einzelnen Punkt auswerten zu müssen. Diese stehen im Gegensatz zu *exakten Methoden*, die mit Sicherheit ein globales Optimum finden.

Definition 3 *Eine Heuristik ist eine Suchmethode, die mit vernünftigem Rechenaufwand gute Lösungen sucht, ohne garantieren zu können, das Optimum zu erreichen oder Aussagen darüber machen zu können, wie weit man vom Optimum entfernt ist.* (Reeves & Beasley [9])

3.2.3 Heuristiken vs. exakte Methoden

Inwiefern Heuristiken eine Alternative zu exakten Verfahren bieten können, soll die Behandlung folgender Fragen erläutern:

Wie optimal ist das Optimum?

Kombinatorische Optimierungsprobleme sind oft Modelle von Problemstellungen in der Industrie, Physik, usw.

Die Verwendung von exakten Methoden ist meist nur bei einfacheren Problemmodellierungen möglich, da solche Methoden z.B. die Linearität der Zielfunktion oder ähnliches voraussetzen oder nur für spezifische Modelle bekannt sind (vgl. 2.3.5). Die unten beschriebenen Heuristiken hingegen fordern keinerlei Einschränkungen zusätzlich zu Definition 2.
Nun kann es erstrebenswert sein, anstatt optimalen Punkten in sehr einfachen oder angenäherten Modellen relativ gute Punkte in komplexeren Darstellungen zu bekommen – solche Punkte kommen den tatsächlich gesuchten Lösungen oft näher als exakte Lösungen eines eher grob angepaßten Modells.

Wie teuer ist das Optimum?

Es kann fragwürdig sein, ob man „um jeden Preis“ (mit hohen Rechenkosten) das globale Optimum finden will oder ob man nicht eine gutes lokales Optimum mit niedrigerem Rechenaufwand bevorzugt – dazu siehe auch 2.5.2.

3.3 Nachbarschaftssuche

Die meisten heuristischen Methoden sind spezielle Formen der sog. *Nachbarschaftssuche*.

Die Idee besteht darin, ausgehend von einem zulässigen Punkt (dem *aktuellen Punkt*), andere Punkte (*Nachbarschaftspunkte*) zu betrachten, die in gewissem Sinn nahe bei dem gegebenen liegen. Man wählt den jeweils „attraktivsten“ als nächsten und hofft, sich so einen Weg in Richtung Optimum zu bahnen.

3.3.1 Nachbarn

Um von einem Punkt des Suchraums zu einem benachbarten zu kommen, sondert man eine Klasse M von Abbildungen $m : S \to S$ ab, die zulässige Punkte in leicht veränderte zulässige Punkte umformen. Die $m \in M$ heißen *Züge*, die *Nachbarschaft* $N(s)$ einer Lösung $s \in S$ ist die Menge aller Punkte, die man durch Anwenden der ganzen Klasse M von Zügen erhält: $N(s) = \{m(s) \,|\, m \in M\}$.

3.3.2 Lokale Minima

Bezüglich Nachbarschaften läßt sich nun auch der Begriff *lokales Minimum* definieren:

Definition 4 *Ein $s' \in S$ heißt lokales Minimum, wenn für alle $s \in N(s')$ gilt, daß $f(s') \leq f(s)$ ist.*

3.3.3 Algorithmus

Das formale Gerüst für eine Nachbarschaftssuche sieht folgendermaßen aus:

1. *Initialisieren*
   Wähle eine Anfangslösung $s_0 \in S$ und setze $s^{now} = s_0$, $s^{best} = s_0$ und $f^{best} = f(s_0)$.

2. *Iterationsschritt*, bzw. *Abbruch*
   Wähle ein $s^{next} \in N(s^{now})$, breche gegebenenfalls ab, wenn ein Abbruchkriterium zutrifft.

3. *Update*
   Setze $s^{now} = s^{next}$ und falls $f(s^{now}) < f^{best}$ (*echte Verbesserung*), ersetzte $x^{best} = s^{now}$ und $f^{best} = f(s^{now})$. Gehe zu Schritt 2

3.4 Einfache Beispiele

Einfache Beispiele von Nachbarschaftssuchen sind die folgenden:

3.4.1 Abstiegsverfahren

Ein typischer, sehr einfacher lokaler Suchalgorithmus ist das sog. *Abstiegsverfahren* (engl. *descent method*): Aus der Nachbarschaft wird ein Punkt gewählt, der näher am Optimum liegt als der akutelle.

Iterationschritt:
Wähle ein $s^{next} \in N(s^{now})$, so daß $f(s^{next}) < f(s^{now})$, Abbruch, wenn kein solches $s \in S$ gefunden werden kann.

3.4.2 Verfahren des steilsten Abstiegs

Ein weiteres Beispiel für Nachbarschaftssuche ist das *Verfahren des steilsten Abstiegs* (engl. *steepest descent method*). Im Iterationsschritt wird der beste Punkt aus der Nachbarschaft gewählt, dieser wird allerdings nur dann akzeptiert, wenn er besser als der aktuelle ist:

Iterationsschritt:
Wähle ein $s^{next} \in N(s^{now})$, so daß $f(s^{next}) \leq f(s)$ für alle $s \in N(s^{now})$ und $f(s^{next}) < f^{now}$, Abbruch wenn kein solches $s \in S$ gefunden werden kann.

3.5 Fortgeschrittene Methoden

3.5.1 Definition

Die einfachen lokalen Suchalgorithmen (wie z.B. die beiden Verfahren aus 3.4) terminieren beim ersten lokalen Minimum, auf das sie stoßen. Bei den folgenden Beispielen von Nachbarschaftssuchen soll das verhindert werden und mit möglichst wenig Aufwand große Teile des Suchraums abgetastet werden. Solche Methoden werden auch *metaheurisitsche Methoden* genannt, die in Glover & Laguna [1] folgendermaßen charakterisiert werden:

Definition 5 *Eine Metaheuristik ist eine übergeordnete Strategie, die andere Heuristiken führen und modifizieren soll, um bessere Lösungen zu produzieren als solche, die normalerweise mit dem Anspruch auf lokale Optimalität gefunden werden. Die zugrundeliegende Heuristik kann bereits ein weiterentwickelter Algorithmus sein, oder einfach nur eine Beschreibung einer Menge von Zügen (die einen zulässigen Punkt in einen anderen modifizieren) zusammen mit einer Auswertungsregel beinhalten.*

Zwischen lokal und global

In diesem Sinn sind die Metaheuristiken also ein Kompromiß zwischen dem lokalen und globalen Optimalitätsanspruch – diese Methoden sollen bessere Ergebnisse bringen als ein beliebiges lokales Minimum, allerdings ohne garantieren zu können, daß die Resultate nahe dem globalen Optimum sind.

3.5.2 Die wichtigsten Beispiele

Im folgenden wird auf die drei bekanntesten metaheuristischen Verfahren eingegangen. In der Literatur gibt es viele allgemeine Kurzbeschreibungen dieser Methoden. Einige davon sind zum Beispiel Dowsland [2], Aarts et al. [3] für *simulated annealing*, Reeves [7], Mühlenbein [8] für *genetische Algorithmen* und Glover & Laguna [4], Hertz et al. [5] über die *Tabusuche*.

Ein besonders umfangreiches Werk über die *Tabusuche* ist Glover & Laguna [1].

Auch im Internet findet man Informationen, wie einen groben Überblick über metaheuristische Methoden unter Thiel [10].

3.5.3 Simulated Annealing

Diese Art der Nachbarschaftssuche soll „den physikalischen Prozeß des Auskühlens von Materialien in einem Hitzebad simulieren“ [2].

Nachbarn werden zufällig erzeugt und sicherlich gewählt, wenn sie die Zielfunktion verringern. Falls sie eine Verschlechterung der Zielfunktion mit sich bringen, werden sie mit einer gewissen Wahrscheinlichkeit, die im Laufe der Suche immer kleiner wird, akzeptiert.

Erhöht der potentielle Nachbarschaftspunkt die Zielfunktion um δ, so wird er trotzdem als neuer Nachbar genommen, falls

$$x < e^{-\frac{\delta}{t}}$$

ist, wobei x gleichmäßig verteilt aus $(0, 1)$ gewählt wird.

t ist die Temperatur, die sich während der Suche verringert und das System „auskühlt“ bis es in einem (zumindest lokalen Minimum) zum Stehen kommen soll.

Der Algorithmus sieht also folgendermaßen aus:

1. *Initialisieren*

   Wähle eine Anfangslösung $s_0 \in S$
   $s^{now} = s_0$
   $s^{best} = s_0$
   $f^{best} = f(s_0)$.

2. *Iterationsschritt*, bzw. *Abbruch*

   Wähle ein zufälliges $s^{next} \in N(s^{now})$
   $\delta = f(s^{next}) - f(s^{now})$
   ist $\delta < 0$,
   dann $s^{now} = s^{next}$
   sonst
   generiere ein gleichmäßig verteiltes $x \in (0, 1)$
   wenn $x < e^{-\frac{\delta}{t}}$,
   dann $s^{now} = s^{next}$
   t_count = t_count + 1
   Gegebenenfalls Stopp bei Abbruchkriterium.

3. *Update*

   Falls $f(s^{now}) < f^{best}$
   $x^{best} = s^{now}$
   $f^{best} = f(s^{now})$.
   falls t_count = new_t
   $t = \alpha(t)$
   t_count = 0

   Gehe zu Schritt 2

Gesetzt werden müssen die *Anfangstemperatur*, das *Kühlungsschema* (die Zahl *n_rep* der Iterationen bei gleicher Temperatur t und die Funktion α, die den Verlauf des Temperaturabfalls bestimmt) und das Abbruchkriterium.

Anfangstemperatur

Um Unabhängigkeit von der Anfangslösung s_0 zu gewährleisten, sollte das System „heiß“ genug sein, so daß zu Beginn jeder Nachbar akzeptiert wird.

Ist etwa die größtmögliche Differenz zwischen zwei Nachbarn bekannt, so kann die Anfangstemperatur t_0 entsprechend gewählt werden.

Oft ist dies nicht der Fall, dann kann beispielsweise t_0 erst beliebig gewählt und dann sukzessive erhöht werden, bis ein vorher festgesetzter Anteil an Nachbarn akzeptiert wird. (Auch dies unterstreicht die Verbindung zum physikalischen Analogon, wo Substanzen rasch bis zum flüssigen Zustand erhitzt und dann langsam zum Auskühlen gebracht werden).

Kühlungsschema

Die Reduktionsfunktion von t wird oft als $\alpha(t) = at$ (mit $a \in (0.8, 0.99)$) oder als $\alpha(t) = \frac{t}{1+\beta t}$ angenommen.

Theoretisch sollten bei jeder Temperatur soviele Iterationen durchgeführt werden, bis keine Nachbarn mehr akzeptiert werden können, praktisch bedeutet das meist zu hohen Rechenaufwand.
Zumindest aber sollte der Parameter *n_rep* bei fallender Temperatur vergrößert werden. Dazu sind drei verschiedene Arten üblich:
n_rep wird geometrisch (*n_rep* = *b*n_rep*, wobei $b > 1$) oder arithmetisch (*n_rep* = *n_rep*+*c*, wobei $c > 0$) verändert. Die dritte Möglichkeit ist, den Verlauf der Suche mitbestimmen zu lassen. Setzt man eine Mindestanzahl von akzeptierten Nachbarn fest, wird *n_rep* automatisch größer – da die Temperatur sinkt, müssen im Laufe der Suche immer mehr Iterationen pro t gemacht werden, um genügend Nachbarn zu finden.

Abbruch

Bei praktischen Anwendungen hat sich gezeigt, daß eine vorgegebene Anzahl von Iterationen kein sinnvolles Kriterium darstellt. Stattdessen wird oft eine untere Schranke für t angegeben (die nahe bei Null liegen soll), oder ein Mindestanteil von akzeptierten Nachbarn pro Temperatur festgelegt, der (bei einer oberen Schranke für *n_rep*) nicht überschritten werden darf.

3.5.4 Genetische Algorithmen

Genetische Algorithmen sollen eine „intelligente Ausnutzung einer zufälligen Suche“ (Reeves [7]) sein. Der Name weist auf die Analogie zwischen der Darstellung eines zulässigen Punktes und der genetischen Struktur eines Chromosoms hin. Wie der Algorithmus funktioniert, soll nun kurz umrissen werden.

Terminologie

Für die hier beschriebene Anwendung von genetischen Algorithmen müssen die zulässigen Punkte als Vektoren (von fixer Länge n) mit Einträgen von 0 und 1 dargestellt werden. Ein solcher Vektor $(a_1, \ldots, a_n)$ heißt *Chromosom*, die Variablen a_i werden als *Gene* und ihre Positionen i als *Loci* bezeichnet.

Genetische Algorithmen sind genaugenommen kein Beispiel der in 3.3 skizzierten Nachbarschaftssuche. Der Hauptunterschied besteht darin, daß (statt eines einzelnen Nachbars) jeweils eine Menge von M Punkten (eine *Population*) betrachtet wird. Auf diese wird mit *genetischen Operatoren* eingewirkt, die die Rolle der Züge übernehmen. Die beiden wichtigsten Klassen dieser Operatoren sind die folgenden:

Kreuzung

Eine Kreuzung (engl. *crossover*) ist eine Abbildung $m : S^2 \to S^2$. Dabei werden aus zwei Chromosomen (den *Eltern* oder *parents*) P_1 und P_2 zwei neue Chromosomen (die *Kinder* oder *offspring*) O_1 und O_2 geschaffen, indem ihre Gene ab dem *Kreuzungspunkt* k getauscht werden. Wird zum Beispiel bei folgenden Eltern die Kreuzung mit $k = 4$ angewandt, so sehen O_1 und O_2 folgendermaßen aus:

$$\begin{array}{ll} P_1\ (1,0,1,0,0,1,0) & \\ P_2\ (0,1,1,1,0,0,1) & \end{array} \quad \longmapsto \quad \begin{array}{ll} O_1\ (1,0,1,0,0,0,1) & \\ O_2\ (0,1,1,1,0,1,0) & \end{array}$$

Mutation

Eine Mutation wirkt auf einzelne Chromosome, in dem sie ein Gen an einem bestimmten Locus l verändert. (Sie kann also als Funktion $m : S \to S$ betrachtet werden.) Folgendes Chromosom macht eine Mutation bei $l = 2$ mit:

$$(1, 0, 0, 1, 0, 1, 1) \mapsto (1, 1, 0, 1, 0, 1, 1)$$

Fitnesswert

Bei vielen Suchstrategien wird den Punkten einer Population ein Fitnesswert zugeschrieben. Er gibt an, wie überlebensfähig das Chromosom sein soll (je größer der Wert, desto „fitter“ ist der entsprechende Punkt). Bei Maximierungsproblemen kann das beispielsweise die Zielfunktion selbst sein.

Selektion

Erst müssen aus einer bestehenden Generation Eltern ausgesucht werden. Diese können zufällig gewählt oder die Wahl mit einer bestimmten Wahrscheinlichkeit (meist abhängig von ihrem Fitnesswert) getroffen werden. Liegt ein Maximierungsproblem vor, so können die Chromosome der aktuellen Population $(c_1, \ldots, c_M)$ jeweils mit Wahrscheinlichkeit

$$P_{select}(c_j) = \frac{f(c_j)}{f_{total}},$$

als Elternteil ausgesucht werden. (f_{total} ist die Summe aller $f(c_j)$. P_{select} kann als Anteil von c_j am Fitnesswert der gesamten Population aufgefaßt werden.) Bei Minimierungsproblemen werden die Chromosome einer Population oft nach absteigendem Wert der Zielfunktion angeordnet und das $j$te Chromosom mit Wahrscheinlichkeit

$$P_{select}(c_j) = \frac{2j}{M(M+1)}$$

gewählt.

Reproduktion

Zuerst werden durch Kreuzung der Eltern (an einem zufällig bestimmten Punkt) die Kinder erzeugt. Danach wird an jedem Gen mit der Wahrscheinlichkeit $\frac{1}{n}$ eine Mutation durchgeführt. Es können entweder beide oder nur ein neues Chromosom (etwa das „bessere“) verwendet werden.

Austausch

Beim Ersetzung von Chromosomen unterscheidet Reeves & Beasly [9] zwei Arten:

1. Generationswechsel (*generational replacement*)
   Es werden M neue Chromosomen aus einer bestehenden Population erzeugt, die danach gänzlich ersetzt wird.

2. zunehmender Austausch (*incremental replacement*)
   Jeweils ein oder mehrere zulässige Punkte (etwa die schlechtesten) der Population werden durch neue ersetzt. Die Methode führt meist dazu, daß erst viele bessere Punkte erzeugt werden, die Suche aber relativ bald in einem lokalen Minimum (bezüglich dem Reproduktionsschema) stagniert.

3.5.5 Eine einfache Tabusuche

Anhand einer besonders einfachen Tabusuche sollen nun die Grundzüge dieses Algorithmus erläutert werden. Eine direkte Anwendung an einem sehr einfachen und anschaulichem Beispiel findet man im Internet unter Domschke et al. [6].

Die Wahl des Nachbarn

Um das Terminieren beim ersten lokalen Minimum zu verhindern, wird in jeder Iteration der beste Punkt aus der Nachbarschaft gewählt – unabhängig davon, ob dieser besser oder schlechter ist als der aktuelle.

Die Tabuliste

Um die Suche in weite Teile des Suchraums zu lenken und um zu verhindern, einen bereits besuchten Punkt wieder zu wählen (es ist ja im allgemeinen nicht möglich, alle bereits ausgewerteten Punkte zu speichern!) gibt es eine sogenannte *Tabuliste* T: dort werden die Umkehrzüge der letzten k Züge gespeichert, diese sind *tabu*: es dürfen keine Punkte aus der Nachbarschaft gewählt werden, die man mit tabu-gesetzten Zügen erreichen würde. (Ein solcher Punkt wird ebenfalls tabu-gesetzt bezeichnet.)

Die Länge der Tabuliste

Vorsicht ist geboten bei der Wahl der Länge der Tabuliste: eine zu kurze Tabuliste bedeutet Gefahr, immer wieder gleiche Punkte zu wählen, mit einer

zu langen Tabuliste hingegen verbietet man möglicherweise vielversprechende Punkte.

Aspiration

Um das Problem mit der Länge der Tabuliste besser handhaben zu können, wird eine sogenannte *Aspirationsfunktion* $h : Z \to Z$ eingeführt, die den Tabustatus eines Zugs folgendermaßen überschreiben kann: sei s der aktuelle Punkt, s' ein Punkt, den man mit einem tabu-gesetzen Zug erreichen würde. Dann wird s' trotzdem gewählt, falls $f(s') \leq h[f(s)]$. Meist wird ein solcher Punkt s' dann zugelassen, wenn er besser ist als der beste bisher besuchte, (d.h. zum Beispiel $h(s) = f^{best} - 1$). In jedem Iterationsschritt muß daher die Menge $N^*(s^{now}) \subseteq N(s^{now})$ untersucht werden, die folgendermaßen aussieht:

Die Nachbarschaft

$N^*(s^{now})$ besteht aus den Elementen von $N(s^{now})$, allerdings ohne den tabu-gesetzten Punkten $\{m(s^{now}) \mid m \in T\}$, jedoch wiederum mit jenen, deren Tabustatus durch die Aspirationsfunktion überschrieben wurde – nämlich den Elementen $\{m(s^{now}) \mid f(m(s^{now})) \leq h(f(s^{now})), m \in T\}$).

Algorithmus

1. *Initialisieren*

   Wähle eine Anfangslösung $s_0 \in S$ und setze
   $s^{now} = s_0$
   $s^{best} = s_0$
   $f^{best} = f(s_0)$.

2. *Iterationsschritt*, bzw. *Abbruch*

   Wähle $s^{next} \in N^*(s^{now})$, so daß $f(s^{next}) \leq f(s)$ für alle $s \in N^*(s^{now})$
   Gegebenenfalls Stopp bei Abbruchkriterium.

3. *Update*

   Falls $f(s^{now}) < f^{best}$
   $x^{best} = s^{now}$
   $f^{best} = f(s^{now})$

Nach Beenden des Algorithmus steht die beste gefundene Lösung in s^{best}.

3.5.6 Erweiterte Tabusuche

Für viele Probleme ist bereits die Anwendung der einfachen Tabusuche von 3.5.5 ausreichend.

Die Idee der Tabusuche, nämlich „eine Reihe von intelligenten Prinzipien zur Problemlösung herzuleiten und auszunützen“ [1, 4], wird aber erst nach einigen Erweiterungen sichtbar. Prinzipiell sollen keine zufälligen Elemente enthalten sein, da sie im Rahmen einer systematischen Suche kaum Vorteile bringen. Dies wird später an einer konkreten Programmversion (in 6.3) illustriert.

Attribute

In der einfachen Tabusuche bestand nur die Möglichkeit, Nachbarn explizit zu verbieten. Es soll aber auch möglich sein, sich nur auf einzelne (erwünschte oder unerwünschte) Merkmale von Zügen, bzw. ihren entsprechenden Nachbarn zu beziehen.

Tabustatus

Mit der Einführung von Attributen muß nun zwischen *tabu-aktiven Attributen* und *tabu-gesetzten Punkten* (bzw. Zügen) unterschieden werden. Punkte können tabu-aktive Attribute besitzen, ohne selbst tabu-gesetzt zu sein. Der Tabustatus von expliziten Punkten und ihren Attributen kann also auf vielerlei Arten zusammenhängen.

Tabudauer

Die Anzahl k von Iterationen, die ein Attribut tabu-aktiv sein soll, kann entweder fix gewählt sein (*fixe Tabudauer*) oder während der Suche verändert werden (*dynamische Tabudauer*). Bei letzterer unterscheidet [1] zwei Arten: Bei der *zufälligen, dynamischen Dauer* (*random dynamic tenure*) wird k zufällig (einer gleichmäßigen Verteilung folgend) zwischen k_{min} und k_{max} bestimmt. Ein neues t wird entweder alle αk_{max} Iterationen festgelegt oder für jedes Attribut neu bestimmt, das tabu-aktiv wird.
Die zweite Art, die *systematische, dynamische Dauer* (*systematic dynamic tenure*), legt vor Beginn der Suche eine Folge $(k_1, \ldots, k_r)$ von Werten zwischen k_{min} und k_{max} fest, die dann in der Suche wiederholt für die Wahl von k herangezogen wird. So kann k z.B. nach einem bestimmten Muster immer wieder vergrößert und verkleinert werden.

Gedächtnis

Die Suche soll ein Gedächtnis haben, das während der Suche relevante Daten speichert. Dabei können Informationen über Züge, ganze Punkte oder Attribute gesammelt werden.

In [1] werden prinzipiell die folgenden Informationen über Attribute unterschieden:

- Häufigkeit (*frequency memory*):
  Wie oft hatten Nachbarschaftspunkte ein bestimmtes Attribut?
- Neuigkeit (*recency memory*):
  Wann (in welcher Iteration) wurde zuletzt ein Nachbar gewählt, der das Attribut aufwies?
- Qualität (*quality memory*):
  Wie „gut“ ist das Attribut (haben die entsprechenden Punkte eine Verbesserung gebracht)?
- Einfluß (*influence memory*):
  Wie hat die Wahl von Nachbarn mit diesem Attribut die Suche beeinflußt (z.B. in der Suchrichtung)?

Kurzzeitgedächtnis

Meist wird das Kurzzeitgedächtnis (also Informationen über den jüngsten Suchverlauf) dazu verwendet, die Nachbarschaft $N(s)$ zu verkleinern, indem sie durch $N^*(s) \subseteq N(s)$ ersetzt wird.

Die in 3.5.5 beschriebene Tabustrategie ist ein Beispiel dafür.

Ein weiteres Beispiel ist die Verwendung einer sog. *Kandidatenliste*, wo die Menge der Züge M auf eine Menge $M^* \subseteq M$ von in gewisser Weise vielversprechenden Zügen verkleinert wird (etwa solche, die in den letzten Iterationen die gute Verbesserungen gebracht hätten). Damit wird die Nachbarschaft vorübergehend auf $N^*(s) = \{m(s) \mid m \in M^*\}$ reduziert.

Langzeitgedächtnis

Hier werden Informationen verwendet, die über große Zeiträume der Suche gesammelt wurden. Diese werden oft benützt, um die Zielfunktion zu modifizieren.

Aspiration

Auch bei der Aspiration muß unterschieden werden, ob der Tabustatus von ganzen Punkten oder nur von Attributen überschrieben wird. Grundsätzlich werden in [1] vier Arten der Aspiration unterschieden:

- Aspiration "by default":
  Manchmal kann es passieren, daß alle Punkte einer Nachbarschaft tabu-gesetzt sind. In so einem Fall wird der (tabu-gesetzte) Nachbar gewählt, der am wenigsten tabu-aktive Attribute aufweist.

- Aspiration anhand der Zielfunktion:
  Ein häufiges Beispiel ist die bereits in 3.5.5 beschriebene (Standard-) Aspirationsfunktion (engl. *improved-best aspiration*).

- Aspiration anhand der Suchrichtung:
  Manchmal ist es erwünscht, die Suchrichtung („bergauf" oder „berg-ab") nicht zu verändern. D.h. der Punkt s^{next} darf gewählt werden, falls
  $$\operatorname{sgn}(f(s^{now}) - f(s^{prev})) = \operatorname{sgn}(f(s^{next}) - f(s^{now}))$$
  ist, wobei s^{prev} der Punkt aus der vorigen Iteration ist.

- Aspiration nach dem Einfluß:
  Hier sollen z.B. Züge, die viele Attribute im entsprechenden Punkte verändern, trotzdem akzeptiert werden.

Diversifikation

Mit einer Diversifikationsstrategie soll die Suche in neue Teile von S gebracht werden. Dies geschieht oft, indem die Zielfunktion auf folgende Weise modifiziert wird:

$$f^*(s) = f(s) + \operatorname{pen}(s)$$

wobei pen: $S \to \mathbb{R}^+$ eine Straffunktion ist. pen kann z.B. angeben, wie oft bestimmte Attribute von s bereits in Nachbarschaftspunkten vorgekommen sind (attributiver Gebrauch von *frequency memory*).

Intensivierung

In einer Intensivierungsphase soll die Suche dazu gebracht werden, gewisse Teile von S besonders genau abzusuchen. So können etwa manche Attribute von einem Punkt zwingend verlangt werden. Auch durch Aspiration anhand der Suchrichtung kann die Suche in eine bestimmte Richtung gelenkt werden.

Kapitel 4

Komplexität lokaler Suchprobleme

4.1 Überblick

Die heuristischen Suchmethoden aus Kapitel 3 finden im allgemeinen nicht das globale Optimum und lösen in diesem Sinne auch nicht das gestellte Optimierungsproblem. Formuliert man das Problem als lokales Suchproblem bezüglich einer festgelegten Nachbarschaft, so sind die beschriebenen Algorithmen echte Lösungsverfahren.

In diesem Kapitel soll nun die Komplexität solcher *lokaler Suchprobleme* untersucht werden, die mit den in 2.4 beschriebenen Komplexitätsklassen nicht charakterisiert werden kann.

Dafür wird zuerst eine neue Klasse $\mathcal{PLS}$ von lokalen Suchproblemen eingeführt, ein Beispiel gebracht und die Komplexität dieser Klasse durch Einordnen in bekannte Strukturen beschrieben.

Die Definition und Notation der Berechnungsprobleme wird aus 2.4.1 übernommen. Diese Darstellung der Klasse $\mathcal{PLS}$ stammt aus MÜHLENBEIN [13].

4.2 Formale Definition

Definition 6 *Ein* **lokales Suchproblem Π** *erhält man aus einem Optimierungsproblem zusammen mit einer Nachbarschaftsstruktur N, die jedem $s \in S(x)$ seine Nachbarschaft $N(s) \subseteq S(x)$ zuordnet. Das Problem besteht darin, für jede Instanz x ist ein lokales Optimum für $S(x)$ bezüglich N zu finden.*

Die entsprechenden Entscheidungsprobleme

Formuliert man, analog zu den Optimierungsproblemen (vgl. 2.4.1), ein Entscheidungsproblem zu einem lokalen Suchproblem, so muß man Fragen wie „Gibt es ein lokales Minimum s_0 mit $f(s_0) \leq k$?“ stellen. Anders als bei globalen Problemen ist bei lokalen das entsprechende Entscheidungsproblem möglicherweise schwieriger, da ein Algorithmus, der ein (irgendein!) lokales Minimum findet, nichts zur Lösung der Entscheidungsfrage betragen muß.

Die lokalen Probleme müssen also wirklich als Suchprobleme untersucht werden und können daher nicht mit den Klassen $\mathcal{P}$ und $\mathcal{NP}$ von Entscheidungsproblemen charakterisiert werden.

4.3 Die Klasse $\mathcal{PLS}$

4.3.1 Definition

Die formale Definition der Klasse $\mathcal{PLS}$ lautet folgendermaßen:

Definition 7 *Ein lokales Suchproblem* Π *ist in* $\mathcal{PLS}$*, wenn es polynomiale Algorithmen A, B und C mit den folgenden Eigenschaften gibt:*

1. *A prüft, ob x eine Instanz für* Π *– also ein Element aus* D_π *– ist und produziert ein* $s_0 \in S(x)$*, wenn dies der Fall ist.*
2. *Für jede Instanz x und jedes s überprüft B, ob* $s \in S(x)$ *ist und berechnet gegebenenfalls den Wert der Zielfunktion* $f_x(s)$*.*
3. *Ist ein x und ein* $s \in S(x)$ *gegeben, so entscheidet C, ob s ein lokales Optimum ist – ist dies nicht der Fall, so gibt C einen Nachbar* $s' \in N(s)$ *mit* $f_x(s') < f_x(s)$ *aus.*

4.3.2 TSP in $\mathcal{PLS}$

Ein Beispiel für ein $\mathcal{PLS}$-Problem ist das Problem der Handlungsreisenden (die formale Definition des Problems wurde in 3.2.1 erwähnt) mit folgender Nachbarschaft:

Die *k-opt* Nachbarschaft

Für eine gültige Route $(\pi_1, \ldots, \pi_n)$ erhält man einen Punkt aus der k-opt-Nachbarschaft, indem man k der n Kanten (π_i, π_{i+1}) ersetzt, so daß wieder eine zulässige Route entsteht.

TSP ist mit der k-opt Nachbarschaft für beliebige $k \leq n$ in $\mathcal{PLS}$.

4.3.3 Die Rolle der Suchmethode

In $\mathcal{PLS}$ liegen also die lokalen Suchprobleme, deren Nachbarschaft in polynomialer Zeit durchsucht werden kann. Das impliziert natürlich nicht, daß deshalb ein lokaler Suchalgorithmus bei einem lokalen Minimum ebenso nach polynomial vielen Schritten terminieren muß.

Der Standardalgorithmus

In MÜHLENBEIN [13] wird das Verhalten des sog. lokalen Standardalgorithmus für Probleme aus $\mathcal{PLS}$ untersucht.

Der Standardalgorithmus ist das in 3.3.3 beschriebene Abstiegsverfahren. Frei gewählt werden darf hier lediglich die sog. *Pivot-Regel*, die festlegt, welcher von den besseren Nachbarn im Iterationsschritt ausgesucht wird.

„Exponentielle Probleme“ in $\mathcal{PLS}$

Es kann gezeigt werden (siehe [13]), daß der Standardalgorithmus – unabhängig von der Wahl der Pivot-Regel – exponentiell viele Schritte brauchen kann, um für ein TSP mit der k-opt-Nachbarschaft (das ja in $\mathcal{PLS}$ liegt) ein lokales Minimum zu finden.

Damit kann aber nicht allgemein ausgeschlossen werden, daß man auf andere Arten das Problem in polynomialer Zeit lösen könnte.

4.4 Wie komplex ist $\mathcal{PLS}$?

Man möchte nun wissen, wie die Klasse $\mathcal{PLS}$ in die bekannten Komplexitätsstrukturen paßt. Daß man $\mathcal{PLS}$ nicht direkt mit $\mathcal{P}$ und $\mathcal{NP}$ vergleichen kann, wird auf folgende Weise umgangen:

4.4.1 $\mathcal{P}_S$ und $\mathcal{NP}_S$

Zu $\mathcal{P}$ und $\mathcal{NP}$ definiert man analoge, allgemeinere Klassen $\mathcal{P}_S$ und $\mathcal{NP}_S$:

Für ein Suchproblem P (in der Definition von 2.4.1) soll für $x \in D_P$ und $y \in O_P(x)$ der Wert $|y|$ polynomial in $|x|$ begrenzt sein.
$\mathcal{P}_S$ besteht aus allen solchen Problemen P, für die es einen polynomialen Algorithmus gibt, der mit Input x ein $y \in O_P(x)$ produziert oder feststellt, daß $O_P(x) = \emptyset$ ist.
Die Klasse $\mathcal{NP}_S$ enthält alle jenen Suchprobleme P, für die man einen polynomialen Algorithmus kennt, der für eine Instanz x und ein y feststellt, ob $y \in O_P(x)$ ist.

$\mathcal{PLS}$, $\mathcal{P}_S$ und $\mathcal{NP}_S$

Setzt man für ein Suchproblem P aus $\mathcal{P}_S$ den Suchraum $S(x) := O_P(x)$, die Zielfunktion $f_x(y) := 0$ und die Nachbarschaft $N(y) := \{y\}$ für alle $y \in S(x)$, dann liegt P offensichtlich in $\mathcal{PLS}$. Ebenso ist jedes Problem aus $\mathcal{PLS}$ der Definition nach auch in $\mathcal{NP}_S$.

Nun kann $\mathcal{PLS}$ folgendermaßen eingeordnet werden:

$$\mathcal{P}_S \subseteq \mathcal{PLS} \subseteq \mathcal{NP}_S.$$

$\mathcal{P}_S = \mathcal{NP}_S$?

Daß die Klassen $\mathcal{P}_S$ und $\mathcal{NP}_S$ sinnvolle Analoga zu $\mathcal{P}$ und $\mathcal{NP}$ darstellen, zeigt die folgende Tatsache.

Satz 8 *$\mathcal{P}_S = \mathcal{NP}_S$ genau dann, wenn $\mathcal{P} = \mathcal{NP}$ ist.*

Einen Beweis dafür kann man in [13] nachlesen.

$\mathcal{PLS}$ und $\mathcal{NP}$-hart

Da kombinatorische, globale Optimierungsprobleme spezielle Suchprobleme sind, enthält $\mathcal{NP}_S$ offensichtlich auch $\mathcal{NP}$-harte Probleme (wie z.B. das Problem des Handlungsreisenden in der Formulierung von 3.2.1 als **globales** Problem). Daß aber auch in $\mathcal{PLS}$ Probleme $\mathcal{NP}$-hart sind, ist aufgrund des folgenden Satzes „unwahrscheinlich“.

Satz 9 *Ist ein Problem aus $\mathcal{PLS}$ $\mathcal{NP}$-hart, so muß $\mathcal{NP}$ = co-$\mathcal{NP}$ sein.*

Der Beweis dazu ist ebenfalls in [13] zu finden. co-$\mathcal{NP}$ sind die Entscheidungsprobleme, deren „Komplement“ in $\mathcal{NP}$ liegt. Das Komplement eines Entscheidungsproblems besteht aus der entgegengesetzten Frage. Für ein Optimierungsproblem wäre das komplementäre Problem folgendes: „Gibt es **kein** $s \in S(x)$, so daß $f_x(s) \leq k$ ist?“. Warum man annimmt, daß diese Fragen im allgemeinen andere Komplexität haben, wurde in 2.4.1 erwähnt.

4.4.2 $\mathcal{PLS}$-vollständig

Sind Π_1 und Π_2 lokale Suchprobleme, dann läßt sich Π_1 auf Π_2 reduzieren ($\Pi_1 \propto \Pi_2$), wenn es polynomial berechenbare Funktionen g_1 und g_2 gibt, so daß g_1 Instanzen von Π_1 in Instanzen von Π_2 überführt und g_2 für jede Instanz x von Π_1 Lösungen von $g_1(x)$ in Lösungen von x verwandelt.

Ein Problem Π aus $\mathcal{PLS}$ heißt *$\mathcal{PLS}$-vollständig*, falls sich jedes $\Pi' \in \mathcal{PLS}$ auf Π reduzieren läßt.

Ein Beispiel

Das Problem des Handlungsreisenden mit der k-opt-Nachbarschaft ist $\mathcal{PLS}$-vollständig (siehe MÜHLENBEIN [13]).

4.4.3 Die Rolle der Nachbarschaft

Bei lokalen Suchproblemen spielt die Wahl der Nachbarschaft eine zentrale Rolle. Je größer die Nachbarschaft ist, desto näher sind die lokalen Mimima am globalen Optimum. Auf der anderen Seite ist es natürlich rechenaufwendiger, große Nachbarschaften untersuchen zu müssen.

Exakte Nachbarschaften

Ist eine Nachbarschaft so groß, daß die lokalen Minima gleichzeitig die globalen sind, so nennt man die Nachbarschaftstruktur *exakt*. Lokale Suchprobleme mit exakten Nachbarschaften sind daher äquivalent zum (globalen) Optimierungsproblem.

$\mathcal{NP}$-harte als lokale Probleme

Wie schon ausführlich erwähnt, werden $\mathcal{NP}$-harte Optimierungsprobleme aufgrund ihrer Komplexität als Kompromiß oft als lokale Suchprobleme mit einer Nachbarschaftsstruktur N formuliert.

Daß man die Komplexität eines solchen Problems damit natürlich nicht umgehen kann, zeigt folgende Aussage (die samt Begründung in [13] zu finden ist):

Satz 10 *Ist* Π $\mathcal{NP}$*-hart, dann kann* N *nur exakt sein, wenn* $\mathcal{NP}$*=* co-$\mathcal{NP}$*.*

Wie bereits in 4.4.1 erwähnt, gilt $\mathcal{NP}$ = co-$\mathcal{NP}$ als unwahrscheinlich.

Kapitel 5

Implementierung des Modells

5.1 Überblick

In diesem Kapitel soll die Schnittstelle zwischen den Modell aus Kapitel 1 und seiner Implementierung in C beschrieben werden – dieses Programm soll dann die Tabusuche für das Optimierungsproblem durchführen.

In Kapitel 6 werden verschiedene Programmversionen aufgezählt, deren Gerüst die hier zusammengefaßten Bausteine sind (siehe 5.2).

Erst wird die Form der eingegebenen Auftragsliste und die Codierung der Auftragselemente erklärt.

Es wird erläutert, wie im Programm die Anfangslösung für die Tabusuche erstellt wird und die Implementierung der für die Zielfunktion relevanten Daten beschrieben. Außerdem wird erwähnt, auf welche Weise die Berechnung der Zielfunktion erfolgt und auf den Rechenaufwand und die Genauigkeit eingegangen.

Schließlich wird gezeigt, wie die Programme zu benutzen sind und der Output zu interpretieren ist.

5.2 Kurzbeschreibung des Programms

Im C-Programm *ts* soll nun die Tabusuche durchgeführt werden. Das Programm besteht prinzipiell aus folgenden Bausteinen:

- Einlesen der Auftragsdaten in eine für das Programm lesbare Form
- Vorsortieren der Aufträge
- Erstellen einer Anfangslösung

- Durchführung einer Tabusuche (TS-Phase)
- Ausgabe relevanter Daten

5.3 Input und Codierung

Die Eingabedatei mit der Bestelliste der Aufträge soll ein Textfile der folgenden Form sein (die erste Zeile dient nur als Erklärung der Spalten und ist im Inputfile nicht vorhanden):

5.3.1 Input

l	m_l	g_l b_l r_l	q_l	L_l	S_l	f_l
1	FLRY	SWGERT	0.75	10000	12865	12632
2	FLRY	SWGERT	0.75	24000	12865	12632
3	FLRY	BL	0.35	24000	67624	68206
4	FLRY	BR	0.75	20000	23274	68244
5	FLRYW	WS	0.75	10000	91661	87065
6	FLRY	SW	0.75	20000	23272	68244
7	FLRY	WS	0.75	20000	23275	68244
8	FLRY	SWBLGN	1.5	38400	33962	68298
9	FLRY	SWRTWS	1.5	38400	32048	68298
10	FLR12Y	GE	1.5	5000	70142	68367
11	FLR12Y	GR	1.5	5000	73291	68367
12	FLRYW	SWGE	0.5	30000	91537	83734
13	FLRY	SWBLGE	1.5	38400	53395	68298
14	FLRY	GN	0.5	60000	24235	68226
15	FLRYW	WS	0.75	10000	91661	87065
⋮						
298	FLRY	RTGNWS	0.75	10000	12832	12632
299	FLRY	SWGERT	0.75	10000	12865	12632
300	FLRY	SWGNVI	0.75	10000	12878	12632

Dieses File wird eingelesen und die Daten, teilweise codiert, in die programminterne Auftragsliste *joblist* geschrieben.

5.3.2 Codierung

In der ersten Spalte steht die Laufnummer, die den dazugehörigen Auftrag später in den Maschinenbelegungen „repräsentiert“.

Massetyp

Die zweite Spalte enthält den Massetyp, der innerhalb des Programms mit Zahlen von 1 bis 7 numeriert wird.

FLR12Y	...	1	FLY	...	5
FLRNY	...	2	FLYK	...	6
FLRY	...	3	FLYW	...	7
FLRYW	...	4			

Farben

Informationen über die Grundfarbe und eventuellen Kennstreifen und Ringsignierung sind in der dritten Spalte vermerkt und folgendermaßen zu verstehen: Farben werden mit jeweils zwei Buchstaben abgekürzt. Die erste Farbe kennzeichnet die Grundfarbe, die zweite (falls vorhanden) die Farbe des Kennstreifen und die dritte die Ringfarbe, falls eine Ringsignierung vorgesehen ist. (Diese Bezeichnung ist möglich, da es keine Aufträge mit Ringsignierung aber ohne Kennstreifen gibt.) Die 12 möglichen Farben werden gemäß der folgenden Tabelle mit Nummern ersetzt.

WS	(weiß)	...	1	RT	(rot)	...	7
BG	(beige)	...	2	RS	(rosa)	...	8
BL	(blau)	...	3	BR	(braun)	...	9
GE	(gelb)	...	4	GR	(grau)	...	10
GN	(grün)	...	5	SW	(schwarz)	...	11
OR	(orange)	...	6	VI	(violett)	...	12

Querschnitt

Die Querschnitte in der vierten Spalte erhalten ebenfalls Codierungen (von 1 bis 9).

0.22	(mm)	...	1	1.5	(mm)	...	6
0.35	(mm)	...	2	2.5	(mm)	...	7
0.5	(mm)	...	3	4.0	(mm)	...	8
0.75	(mm)	...	4	6.0	(mm)	...	9
1.0	(mm)	...	5				

Sachnummern

Die Sachnummer S_l in der fünften Spalte braucht nicht codiert zu werden, da sie nur zum Unterscheiden der Aufträge dient, die unterschiedliche Produkte

enthalten. In der letzten Spalte steht die farbneutrale Sachnummer. Dem Programm sind 80 verschiedene Nummern bekannt, sie werden mit Zahlen von 1 bis 80 ersetzt.

5.4 Vorsortieren

Es ist wünschenswert, daß Aufträge mit gleicher Sachnummer hintereinander produziert werden, da Umrüst- und Anfahrzeiten zwischen diesen Aufträgen gänzlich wegfallen (siehe 1.3). Im Programm wird deshalb eine zweite Auftragsliste erzeugt, in der Aufträge mit gleicher Sachnummer zu einem einzigen (mit entsprechender Länge) zusammengefaßt werden. Damit ist gewährleistet, daß solche Aufträge von vornherein hintereinander gereiht sind und auch während der Tabusuche nicht mehr „auseinandergerissen“ werden können. Ab jetzt werden nur mehr die Aufträge aus dieser Liste betrachtet.

5.5 Anfangslösung

Die Anfangslösung als Ausgangspunkt der Tabusuche wird folgendermaßen erstellt: Zuerst werden die Aufträge in zwei Listen L_1 und L_2 getrennt, von denen die erste alle Produkte mit Ringsignierung, die zweite diejenigen ohne enthält. Innerhalb dieser Listen werden Aufträge mit gleicher Grundfarbe hintereinandergereiht. Die Aufträge aus L_1 werden nacheinander auf M_1 und M_2 geschrieben, diejenigen aus L_2 auf möglicherweise „verbleibenden“ Platz auf M_1, bzw. M_2 (insgesamt soll auf jeder Maschine in etwa die gleiche Menge Kabel produziert werden) und auf die restlichen Maschinen.

Das Sortieren nach Grundfarbe entspricht in etwa dem Schema, nach dem Aufträge im Kabelwerk zur Zeit händisch an die Maschinen verteilt werden.

5.6 Die Zielfunktion in *ts*

Für die Auswertung der Zielfunktion an einem zulässigen Punkt sind die in 1.5 beschriebenen Daten nötig. Da alle relevanten Auftragsdaten mit fortlaufenden Nummern (beginnend bei 1) „codiert“ wurden, können die Daten für die Zielfunktion im Programm nun einfach in Felder der entsprechenden Dimensionen geschrieben werden. So wird beispielsweise für die Umrüstzeit auf neue Ringfarbe ein Feld

float setup_time1[n][13][13];

deklariert, das beschrieben wird mit setup_time1[i][k][l] = $T_{u_1}(i,$ Farbe $l,$ Farbe $k)$ mit $i = 1, \ldots, 11$ und $k, l = 1, \ldots, 12$. Außerdem enthält das Feld

eine zusätzliche 0-te „Spalte“ und „Zeile“:

$$\text{setup\_time1}[i][k][0] = \text{setup\_time1}[i][0][k] = 0.$$

Damit wird formal die Umrüstzeit von und auf Ringfarbe 0 gleich 0 gesetzt. Somit ist später bei der Auswertung der Zielfunktion keine neuerliche Abfrage nötig, ob der Auftrag überhaupt eine Ringsignierung beinhaltet.

Die restlichen Daten sind auf ähnliche Weise im Programm enthalten.

5.6.1 Datenbeschaffung

Von der Firma standen EXCEL-Dateien mit Rohdaten zur Verfügung. Diese wurden in Textdateien umgewandelt und mit Hilfe der UNIX utilities “awk” und “perl” in der gewünschten Form in ein C-Programm mit den entsprechenden Datenfeldern übertragen.

5.6.2 Programmcode

Die Zielfunktion setzt sich zusammen aus der Summe der Kosten aller Aufträge. Die Kosten eines Auftrags sind abhängig von der Maschine, auf der der Auftrag produziert wird, dem Auftragsprodukt selbst und dem davor produzierten Auftrag.
In der Notation der Programms ist *job_cost(job_prev,job_cur,i)* die Funktion, die die Kosten des Auftrags *job_cur* berechnet, der auf der Maschine M_i produziert wird und dem der Auftrag *job_prev* vorangegangen ist. Die Berechnung der Zielfunktion erfolgt in der Funktion *eval_solution*, in Pseudo-C-Notation sieht sie folgendermaßen aus:

```
for i=1 to 11
{
      job_cur = job(i,1)
      job_prev = 0;
      objective += job_cost(job_prev,job_cur,i);
      for k=2 to n_i
      {
            job_cur = job(i,k)
            job_prev = job(i,k-1)
            objective += job_cost(job_prev,job_cur,i);
      }
}
```

Für jede Maschine M_i (erste *for*-Schleife) und jeden Auftrag $job(i,k)$ auf M_i (zweite *for*-Schleife) werden die Kosten von $job(i,k)$ berechnet und in der Variable *objective* aufsummiert.

5.6.3 Genauigkeit und Rechenaufwand

Die meisten Tabellen, die zur Berechnung der Zielfunktion nötig sind, sind als Felder von Gleitkommazahlen implementiert (vgl. 5.6). Nun wird f nicht immer in der Form von 5.6.2 berechnet. Damit sichergestellt ist, daß die Zielfunktion wohldefiniert ist, hat ihr kleinster einheitliche Baustein, die Funktion *job_cost*, einen Rückgabewert vom Typ *integer*. „Danach“ werden nur mehr Operationen mit ganzzahligen Werten durchgeführt.

Als Maß für den Rechenaufwand eines Programmdurchlaufs wird die Zahl der Funktionsaufrufe von *job_cost* herangezogen.

5.7 Benutzung der Programme

Wird die EXCEL-Datei mit den Auftragsdaten als Textdatei `inputfile` gespeichert, so kann das Programm mit

```
schedule inputfile
```

aufgerufen werden.

Dabei werden erst aus der Datei `inputfile` die relevante Daten herausgelesen und überprüft, ob diese einen gültigen Input für das Programm darstellen. Ist dies nicht der Fall (kommen z.B. Daten vor, die dem Programm nicht bekannt sind), so wird abgebrochen und etwa folgende Nachricht ausgegeben:

```
Unbekannte Sachnummer im Auftrag 175!!
Stop.
```

Ist der Input gültig, so wird die Tabusuche durchgeführt und am Bildschirm erscheint folgende Meldung:

```
Daten in Ordnung!!
Anzahl der Aufträge: 300
Programm durchgeführt.
Benötigte Zeit: 21 Sekunden.
Verbesserung seit der Anfangslösung: 8.998801 %
Gefundene Belegung in Datei schedule_out
```

Die vom Programm gefundene Maschinenbelegung hat folgende, selbsterklärende Form:

Sachnr.	Grundf.	Kennf.	Ringf.	Masset.	Qschn.	Menge	fn. Snr
Anzahl der Aufträge auf Maschine 1: 29							
job(1,1): Laufnummer 224							
34363	rot	braun	gelb	FLRY	1.5	14000	68303
job(1,2): Laufnummer 139							
29946	grün	blau	gelb	FLRY	0.5	30000	68229
job(1,3): Laufnummer 202							
31821	rot	weiss	gelb	FLRY	1.0	10000	68281
⋮							
Anzahl der Aufträge auf Maschine 9: 23							
job(9,1): Laufnummer 103							
25374	gelb	braun	keine	FLRY	0.5	75000	68228
job(9,2): Laufnummer 66							
65951	braun	keine	keine	FLRY	0.35	24000	68206
job(9,3): Laufnummer 97							
29292	braun	orange	keine	FLRY	0.5	45000	68228
⋮							

Kapitel 6

Die einzelnen Programmversionen

In diesem Kapitel wird auf die TS-Phase der einzelnen Programmen genauer eingegangen. Die Ideen aus 3.5.6 sollen an den konkreten Implementationen illustriert werden.
Insgesamt werden 9 verschiedene Programmversionen beschrieben und ihre Ergebnisse verglichen. Die Aufzählung der Programmversionen entspricht der chronologischen Reihenfolge ihres Entstehens – die einzelnen Programme sollen jeweils „besser“ sein als die vorige Version.
Eine Tabelle der Ergebnisse befindet sich am Ende des Kapitels auf Seite 79.
Alle Programme wurden mit 3 verschiedenen Auftragslisten (alle der Länge 300) aufgerufen. Da die entsprechenden Ergebnisse auch im Vergleich der einzelnen Versionen zueinander ähnlich ausfielen, werden hier nur die Resultate einer bestimmten Liste herangezogen.

6.1 ts1 – Die grundlegende Version

Diese Version entspricht der in 3.5.5 beschriebenen einfachen Tabusuche, und dient als Ausgangspunkt für die darauffolgenden Programme.

6.1.1 Die Suche

Nachbarschaft

Die Nachbarschaft einer zulässigen Belegung $s \in S$ sollen alle zulässigen Punkte sein, die man durch Vertauschen zweier Aufträge erhält. Ein Zug m ist repräsentiert durch ein Quadrupel $m = (i_1, k_1, i_2, k_2)$, m bewirkt das Vertauschen von $job(i_1, k_1)$ mit $job(i_2, k_2)$. Ungültig sind Züge, die Aufträge

mit Ringsignierung auf eine Maschine zuweisen, auf der eine solche nicht möglich ist (diese Züge können erst ausgeschlossen werden, wenn die aktuelle Belegung bekannt ist). Formal sieht die Klasse M folgendermaßen aus: M beinhaltet also alle Quadrupel (i_1, k_1, i_2, k_2) mit $i_1, i_2 = 1, \ldots, m;\ k_1 = 1, \ldots, n_{i_1}, k_2 = 1, \ldots, n_{i_2}$ und $i_1 \leq i_2$. Falls $i_1 = i_2$ ist, dann muß $k_1 < k_2$ sein. $i_1 \in R$ und $i_2 \notin R$ ist nur möglich, wenn $r_{job(i_1,k_1)} = 0$ ist. (Ein solcher Zug wäre ungültig.)

Tabustrategie

Die Umkehrzüge der letzten k Züge sollen tabu-gesetzt werden. In einer Tabu-Liste werden einfach die Züge der letzten k Iterationen vermerkt. (In der Bezeichnung von oben ist ein Zug mit seinem Umkehrzug identisch.)

Aspiration

Für die Aspirationsfunktion wird die Standardfunktion (*improved-best*) gewählt. Auch bei den folgenden Programmversionen ist überall die Standardfunktion implementiert.

Abbruch

Der Algorithmus terminiert nach einer vorgegebenen Anzahl von Iterationen.

6.1.2 Rechenaufwand und Ergebnisse

Rechenaufwand

Im Iterationsschritt wird also erst jeder gültige Zug generiert (und mit den k Zügen aus der Tabuliste verglichen) und dann die Zielfunktion am entsprechenden Nachbarschaftspunkt ausgewertet. Vernachlässigt man die Zahl der ungültigen Züge, so wird für jeden der $\frac{1}{2}n(n-1)$ Züge die Funktion *job_cost* n mal aufgerufen werden. Das ergibt einen Aufwand von $\frac{1}{2}n^2(n-1)$, also $O(n^3)$ Funktionsaufrufen.

Ergebnisse

*ts*1 liefert einen Output der folgenden Form:

> input file: tab300
>
> number of iterations: 1000
>
> length of tabu list: 40

f(initial solution): 152587
f(last solution) = 142736
f(best solution) = 142736 in iteration 85

During TS:

* 85 new best solutions found
* no tabu moves
* 915 moves with no change in objective

#(function calls of job_cost): 5899806249

total improvement 6.455989 %
time used for TS: 42882 seconds = 11 hours 11 minutes 42 seconds

Man sieht also, daß nach 1000 Iterationen ein Punkt gefunden wurde, dessen Kosten im Vergleich zur Anfangslösung ca. 6.46 % geringer sind. In den ersten 85 Iteration konnten 85 echte Verbesserungen erreicht werden, danach wurden allerdings nur mehr Züge gewählt, die den Wert der Zielfunktion unverändert lassen. Die Funktion *job_cost* wurde in etwa 7 Milliarden mal aufgerufen und die Laufzeit betrug rund 11 Stunden.

6.2 ts2 – Die beschleunigte Version

Um überhaupt neue Suchstrategien sinnvoll ausprobieren zu können, soll vorerst der Rechenaufwand drastisch verringert werden. Die Suchstrategie (**Nachbarschaft**, **Tabustrategie** und **Aspiration**) von *ts*2 bleibt die gleiche wie bei *ts*1, allerdings soll die Laufzeit durch intelligente Kostenauswertung verkürzt werden.

6.2.1 Rechenaufwand und Ergebnisse

Rechenaufwand

Da in jedem Schritt die Zielfunktion f an mehreren Punkten ausgewertet wird, die sich von der aktuellen Belegung nur leicht unterscheidet, legt die Struktur von f legt nahe, nicht jedesmal die Kosten jedes einzelnen Auftrags zu berechnen, sondern den Wert an den geänderten Stellen entsprechend auszubessern.

Dies geschieht in der Funktion *update_eval*. Abhängig davon, ab die zu vertauschenden Aufträge hintereinander gereiht sind oder nicht, müssen die

Kosten von 6, bzw. 8 Aufträgen berechnet werden. Die Abb. 6.1 veranschaulicht, welche Aufträge betroffen sind, wenn der Zug $m = (i_1, k_1, i_2, k_2)$ mit $i_1 = i_2$ und $k_1 + 1 = k_2$ durchgeführt werden soll.

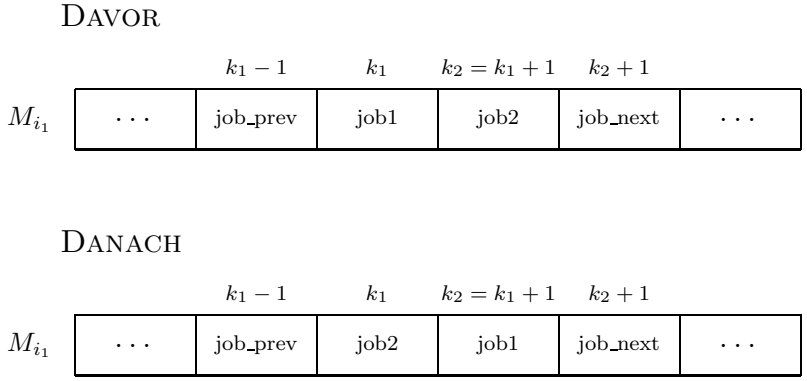


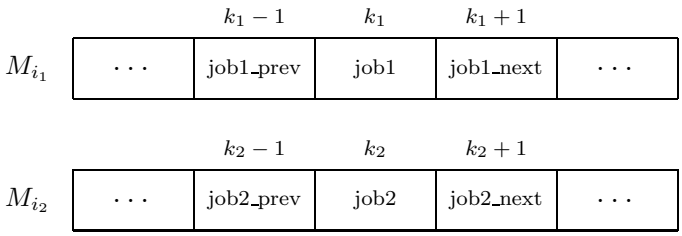
Abbildung 6.1: 1. Art eines Austauschzugs

Man sieht, daß nicht nur die Kosten der vertauschten Aufträge ausgebessert werden müssen, sondern auch auch die des darauffolgenden Auftrags, der ja nach dem Vertauschen einen neuen „Vorgänger“ erhält. Liegt der oben beschriebene Zug m vor, so sieht die entsprechende Passage in $ts2$ folgendermaßen aus ($job(.,.)$ ist die Belegung, von der ausgegangen wird, f_old der Wert der Zielfunktion an diesem Punkt.):

$$
\begin{aligned}
&\text{job\_prev} = \text{job}(i_1,k_1 - 1);\\
&\text{job1} = \text{job}(i_1,k_1);\\
&\text{job2} = \text{job}(i_2,k_2);\\
&\text{job\_next} = \text{job}(i_2,k_2 + 1);\\
&\text{old\_cost} = \text{job\_cost}(\text{job\_prev},\text{job1},i_1) +\\
&\qquad\qquad \text{job\_cost}(\text{job1},\text{job2},i_1) +\\
&\qquad\qquad \text{job\_cost}(\text{job2},\text{job\_next},i_1);\\
&\text{new\_cost} = \text{job\_cost}(\text{job\_prev},\text{job2},i_1) +\\
&\qquad\qquad \text{job\_cost}(\text{job2},\text{job1},i_1) +\\
&\qquad\qquad \text{job\_cost}(\text{job1},\text{job\_next},i_1);\\
&\text{f\_new} = \text{f\_old} - \text{old\_cost} + \text{new\_cost};
\end{aligned}
$$

Liegen die zu vertauschenden Aufträge nicht hintereinander, so müssen die Kosten bei 6 Aufträgen ausgebessert werden (siehe Abb. 6.2).

DAVOR



DANACH

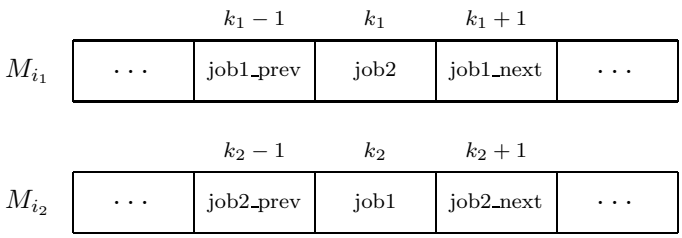


Abbildung 6.2: 2. Art eines Austauschzugs

Der entsprechende Programmcode ist:

job1_prev = job(i_1,$k_1 - 1$);
job1 = job(i_1,k_1);
job1_next = job(i_1,$k_1 + 1$);
job2_prev = job(i_2,$k_2 - 1$);
job2 = job(i_2,k_2);
job2_next = job(i_2,$k_2 + 1$);
old_cost = job_cost(job1_prev,job1,i_1) +
job_cost(job1,job2_next,i_1) +
job_cost(job2_prev,job2,i_2) +
job_cost(job2,job2_next,i_2);
new_cost = job_cost(job1_prev,job2,i_1) +
job_cost(job2,job1_next,i_1) +
job_cost(job2_prev,job1,i_2) +
job_cost(job1,job2_next,i_2);
f_new = f_old − old_cost + new_cost;

Je nach der Art des Zuges muß also die Funktion *job_cost* statt n-mal nur mehr 6, bzw 8 mal aufgerufen werden. Der Aufwand pro Iteration beträgt nun mehr höchstens $4n(n-1)$, also $O(n^2)$ Aufrufe.

Ergebnisse

Die gefundenen Punkte von $ts2$ sind natürlich identisch mit den Ergebnissen von $ts2$.

> #(function calls of job_cost): 184326929
>
> total improvement 6.455989 %
> time used for TS: 3550 seconds = 59 minutes 10 seconds

Die Funktionsaufrufe von konnten also auf rund 180 Millionen (das sind nur mehr 3.12 % im Vergleich zu $ts1$), die Laufzeit auf etwa 1 Stunde reduziert werden.

6.3 ts3 – Zufallszüge

In $ts3$ soll ein zufälliges Element eingebaut werden und die Nachbarschaft drastisch verkleinert werden.

6.3.1 Die Suche

Nachbarschaft

Die Nachbarschaft wird „zufällig“ gewählt: In jeder Iteration werden n viele Zufallszüge generiert (Quadrupel von Zufallszahlen, die im gewünschten Bereich liegen, sodaß ein gültiger Zug entsteht), deren entsprechende Punkte die Nachbarschaft bilden.

Die **Tabustrategie** und **Aspirationsfunktion** wurden wie in 6.1 und 6.2 implementiert.

6.3.2 Rechenaufwand und Ergebnisse

Rechenaufwand

Für jeden der n Nachbarn wird zur Auswertung von f die Funktion *update_eval* herangezogen, die wie oben beschrieben *job_cost* 6 oder 8-mal aufruft. Das ergibt einen Rechenaufwand von höchstens $8n$ Aufrufen pro Iteration, also nur mehr $O(n)$.

Ergebnisse

Je nach Wahl des Anfangswerts für den Zufallsgenerator (*seed*) brachte *ts*3 verschiedenen Output. Nach 1000 Durchläufen mit verschiedenen Anfangswerten lag die Verbesserung zwischen 2.23 % und 4.20 % (durchschnittlich bei 3.36%). Das „beste“ Ergebnis war folgendes:

> input file: tab300
>
> number of iterations: 1000
>
> length of tabu list: 40
>
> seed for random numbers: 41
>
> f(initial solution): 152587
> f(last solution) = 146176
> f(best solution) = 146176 in iteration 720
>
> During TS:
>
> * 71 new best solutions found
> * 10 tabu moves
> * 942 moves with no change in objective
>
> total improvement 4.201537 %
>
> #(function calls of *job_cost*): 1832249
>
> time used for TS: 21 seconds

Die geringste Verbesserung brachte der Anfangswert 482:

> input file: tab300
>
> number of iterations: 1000
>
> length of tabu list: 40
>
> seed for random numbers: 482
>
> f(initial solution): 152587
> f(last solution) = 149188
> f(best solution) = 149188 in iteration 348
>
> During TS:
>
> * 58 new best solutions found
> * 5 tabu moves chosen
> * 929 times no change in objective

total improvement 2.227582 %

#(function calls of *job_cost*): 1840249

time used for TS: 21 seconds

Die Zahl der Aufrufe von *job_cost* liegt bei etwa 2 Millionen und beträgt nur 0.03 % im Vergleich zu *ts*1. Die besten gefundenden Punkte allerdings sind im Vergleich zu *ts*1 und *ts*2 wesentlich schlechter und sehr stark vom Anfangswert des Zufallsgenerators abhängig. Diese Wahl der Nachbarschaft ist also „zu zufällig“ und keine intelligente Strategie im Sinne der Tabu Suche.

6.4 ts4 – Die schnelle Suche

6.4.1 Die Suche

Mit einem “smart update” soll einerseits die Laufzeit ähnlich klein wie bei *ts*3 gehalten werden, andererseits gute Resultate wie bei *ts*1 und *ts*2 erzielt werden.

Nachbarschaft

Die Nachbarschaft selbst ist die gleiche wie in *ts*1 und *ts*2, allerdings wird eine andere Repräsentation gewählt, die ein noch schnelleres Updaten erleichtert. Die Positionen der Aufträge auf den Maschinen werden von 1 bis n numeriert und mit (p, q) (wobei $p < q$) das Vertauschen des Auftrags auf der Position p mit dem Auftrag auf der Position q bezeichnet.

Notation

Falls nicht extra erwähnt, sollen mit *job*1 und *job*2 die entsprechenden Aufträge des Zugs (p, q) gemeint sein und mit den *Koordinaten* (i, k) von p die tatsächliche Position auf der Maschine. (Diese muß ja für die Kostenberechnung eines Auftrags bekannt sein!)

Tabustrategie

Im Grunde soll mit der Tabustrategie erreicht werden, daß bestimmte Aufträge eine gewisse Zeit lang nicht mehr in bestimmte Positionen zurückkehren. Mit dem bisherigen Konzept werden einerseits viele Züge unnötigerweise verboten, andererseits nicht unbedingt verhindert, daß manche Aufträge zu früh in ihre alten Positionen zurückgetauscht werden. Außerdem muß jeder

in Betracht gezogene Zug erst mit allen k Elementen aus der Tabuliste verglichen werden, um seinen Tabustatus zu bestimmen.

Folgende Idee soll nun dahingehend Verbesserungen bringen: Deklariert wird ein Feld

$$\text{int tabu}[n][n],$$

wobei tabu[*job*][*pos*] die Iteration angibt, bis zu der der Auftrag *job* nicht in die Position *pos* zurückkehren darf. Damit ist die Bestimmung des Tabustatus und auch das Update ist einfach: wird in der Iteration *current_iter* der Zug (p, q) gewählt, so setzt man

$$\begin{aligned} \text{tabu}[job1][p] &= current\_iter + tabu\_depth \text{ und} \\ \text{tabu}[job2][q] &= current\_iter + tabu\_depth, \end{aligned}$$

wobei *tabu_depth* die Anzahl der Iterationen angibt, die ein Auftrag nicht in seine vorige Position zurückkehren darf. *tabu_depth* wird zu Beginn des Programmdurchlaufs gewählt und dann nicht mehr verändert.

Somit hat nun jeder Nachbar 2 mögliche Tabuattribute – der entsprechende Zug soll tabu-gesetzt sein, falls **beide** Attribute tabu-aktiv sind.

6.4.2 Rechenaufwand und Ergebnisse

Rechenaufwand

Im Programm gibt es ein Feld

$$\text{int swap\_cost}[n][n].$$

In swap_cost$[p][q]$ werden die Kosten (die Differenz im Wert der Zielfunktion, wenn der Zug (p, q) ausgeführt wird) gespeichert.

Da sich, wie bereits erwähnt, ein benachbarter Punkt von dem aktuellen nur geringfügig unterscheidet, verändern sich von einer Iteration zur nächsten auch die Kosten der meisten Züge nicht (bei $n = 300$ bleiben beispielsweise 98 % unverändert). Genauer gesagt sind, wieder abhängig davon, ob zwei hintereinanderliegende Aufträge vertauscht werden oder nicht, $6(n-1)$, bzw. $8(n-1)$ Züge betroffen. Die Berechnung der Kosten eines Zuges erfolgt in der Funktion *swap_move_cost* analog zur Funktion *update_eval* (die Kosten eines Zuges entsprechen dem Wert *f_old* − *f_new*). Dazu wird *job_cost* ebenfalls 6, bzw 8-mal aufgerufen. Damit ergibt sich pro Iteration ein Aufwand von höchstens $64(n-1)$ Funktionsaufrufen, also wie bei *ts*3 nur mehr $O(n)$.

Ergebnisse

input file: tab300

number of iterations: 1000

tabu_depth = 40

f(initial solution): 152587
f(last solution) = 142503
f(best solution) = 142503 in iteration 97

During TS:

* 97 new best solutions found
* no tabu moves
* 903 moves with no change in objective

#(function calls of job_cost): 3988025

total improvement 6.608689 %
time used for TS: 40 seconds

Im Vergleich zu *ts*1 (und *ts*2) bringt *ts*4, abgesehen von der geringen Laufzeit (40 Sekunden und vgl. mit *ts*1 nur 0.07 % der Funktionsaufrufe!), auch ein besseres Resultat beim besten gefundenen Punkt. Dies ist auf die neue Tabustrategie zurückzuführen.

Das schnelle Update macht auch die Verwendung von Kandidatenlisten überflüssig, da schon bei geringem Aufwand die gesamte Nachbarschaft untersucht werden kann!

6.5 ts5 – Mit Diversifikation

Die Ergebnisse der bisherigen Programmversionen zeigen, daß das zwar anfangs rasch viele bessere Punkte gefunden werden (bis zum Auffinden des besten Punktes gibt es in jeder Iteration eine echte Verbesserung), danach aber nur mehr Nachbarn gewählt werden, die keine Änderungen in der Zielfunktion verursachen (Züge, die zu solchen Nachbarn führen, heißen *Nullkostenzüge*).

Mit einer Diversifikationsstragtegie soll nun die Suche in weitere Teile von S vordringen können.

6.5.1 Die Suche

Nachbarschaft, **Tabustrategie** und **Aspiration** sind unverändert zu *ts*4.

Diversifikation

Die Idee dieser Strategie beruht auf der Annahme, daß neue (gute) Punkte gefunden werden können, wenn Züge gewählt werden, deren Komponentenaufträge bisher selten getauscht wurden. Die Wahl solcher Nachbarn soll erleichtert werden, indem Züge mit bereits oft verschobenen Aufträgen „bestraft“ werden: Dafür gibt es ein Feld

$$\text{int penalty\_record}[n].$$

penalty_record[*job*] gibt an, wie oft der Auftrag *job* bereits Teil eines Zuges war. (Für die Aktualisierung müssen also in jeder Iteration einfach die entsprechenden zwei Komponenten um 1 erhöht werden.)

Die Kosten der Züge werden auf folgende Weise modifiziert (*PEN_TIMES* ist eine Konstante):

$$\begin{aligned} \text{cost} \;=\; & \text{swap\_cost}[p][q] + \\ & \mathit{PEN\_TIMES} * (\text{penalty\_record}[\mathit{job}1] + \text{penalty\_record}[\mathit{job}2]) \end{aligned}$$

Das Feld *penalty_record* wird während der gesamten Suche geführt, die Strafterme der Diversifikation werden erst „eingeschaltet“ nachdem für eine gewisse Anzahl von Iterationen keine echte Verbesserung erzielt werden konnte. Die Strafkosten werden deaktiviert, sobald ein Nachbar gefunden wird, der f verringert und wieder aktiviert, wenn ein Zug mit nicht-negativen Kosten gewählt wird.

6.5.2 Rechenaufwand und Ergebnisse

Rechenaufwand

Die Anzahl der Aufrufe der Funktion *job_cost* bleibt die gleiche wie in *ts*5, nämlich $O(n)$.

Ergebnisse

Leider brachte die Diversifikationsstrategie in *ts*5 nur wenige weitere echte Verbesserungen:

input file: tab300

number of iterations: 1000

length of tabu list: 40

penalties multiplied by 2

f(initial solution): 152587
f(last solution) = 142420
f(best solution) = 142420 in iteration 639

During TS:

* 109 new best solutions found
* no tabu moves
* 889 moves with no change in objective
* 1 "just-improving" move
* 1 deteriorating move
* penalty record was active 879 times

penalty record first active in iteration 109
total improvement 6.663084 %

#(function calls of *job_cost*): 5416681

time used for TS: 48 seconds

Die etwas längere Laufzeit ist auf den Einsatz von *penalty_record* zurückzuführen. Zu bemerken ist, daß die Suche nun gezwungen wurde, auch „bergauf" zu gehen – was für das Auffinden weiterer guter Punkte unerläßlich ist. Die Strafterme wurden aktiviert, sobald die Lücke zwischen der letzten echten Verbesserung und der aktuellen Iteration größer als 10 war. (Da in $ts4$ – die mit $ts5$ bis auf die Diversifikation übereinstimmt – die letzte echte Verbesserung in Iteration 97 gefunden wird, wird in $ts5$ die Diversifikation nach der 108. Iteration gestartet und kommt daher in der 109. erstmals zum Einsatz.)

6.6 ts6 – Eine neue Nachbarschaft

Nachdem mit der bisherigen Art von Zügen (*Austauschzüge*) von vornherein nur ein geringer Teil des Suchraums erkundet werden kann (mit der Anfangslösung werden die n_i bestimmt und werden danach nicht mehr verändert), werden nun *Einfügezüge* eingeführt.

6.6.1 Die Suche

Nachbarschaft

Ein Einfügezug wird ebenfalls durch ein Tupel (s, t) dargestellt, das bedeutet, daß der Auftrag auf der Position s gelöscht und auf der Position t wieder

eingefügt werden soll. Zu bemerken ist, daß es n viele Positionen gibt, auf dem ein Auftrag gelöscht werden kann (*Löschpositionen*), aber $n + m$ viele, auf denen er wieder eingefügt werden kann (*Einfügepositionen*) – Aufträge können vor dem ersten und nach dem letzten dazugefügt werden! Um die Berechnung der Kosten zu vereinfachen, werden nur Züge betrachtet, die Aufträge auf andere Maschinen verschieben.
Außerdem sollen nun pro Maschine höchstens K Aufträge erlaubt sein, damit durch die Einfügezüge die Maschinen nicht (in gewissem Sinn) geleert oder überfüllt werden können. Das Ausmaß dieser Restriktion kann durch die Wahl von K geregelt werden. (K sollte in Abhängigkeit von n und m bestimmt werden.) So kann es passieren, daß Maschinen durch das Einfügen von Aufträgen „voll“ werden. Ist dies der Fall, so werden die Züge, die auf solchen Maschinen Aufträge einfügen sollen, ungültig.

Notation

Beim Einfügezug (s, t) sollen mit (i_1, k_1) die Koordinaten der Position s, mit (i_2, k_2) die von t gemeint sein. *job* sei der Auftrag, der auf s gelöscht und auf t eingefügt wird.

Tabustrategie

Da die Einfügezüge viele Aufträge um eine Position verschieben, ohne sie im eigentlichen Sinne zu bewegen, ist die bisherige Tabustrategie bei Einfügezügen sinnlos. Stattdessen ist für einen Auftrag signifikant, auf welcher Maschine er sich befindet und welcher Auftrag sein „Vorgänger“ ist.

Wie bei *tabu*1 (vgl. 6.4.1), gibt tabu2$[i][job_prev][job]$ an, bis zu welcher Iteration der Auftrag *job* nicht auf die Maschine M_i als Nachfolger von *job_prev* zugewiesen werden darf.

Der in Abb. 6.5 beschriebene Einfügezug wäre tabu, falls

$$\begin{aligned}
&\text{tabu2}[i_1][job1_prev][job1_next] \leq current_iter \quad \text{oder} \\
&\text{tabu2}[i_2][job2_prev][job] \leq current_iter \quad \text{oder} \\
&\text{tabu2}[i_2][job][job2_next] \leq current_iter
\end{aligned}$$

ist. (Hier hat ein Nachbar also 3 mögliche Tabuattribute – ist **eines** davon aktiv, so ist der entsprechende Punkt tabu.)

Das Update erfolgt analog zum Update von *tabu*1.

Eine **Diversifikationsstrategie** ist vorerst nicht vorhanden.

6.6.2 Rechenaufwand und Ergebnisse

Analog zu den Austauschzügen gibt es ein Feld *insert_cost* von $n \times (n + m)$ integern, wobei insert_cost$[s][t]$ die Kosten des Einfügezugs (s, t) angibt. Abb. 6.3 veranschaulicht, wie sich der gegebene Punkt durch den Zug verändert.

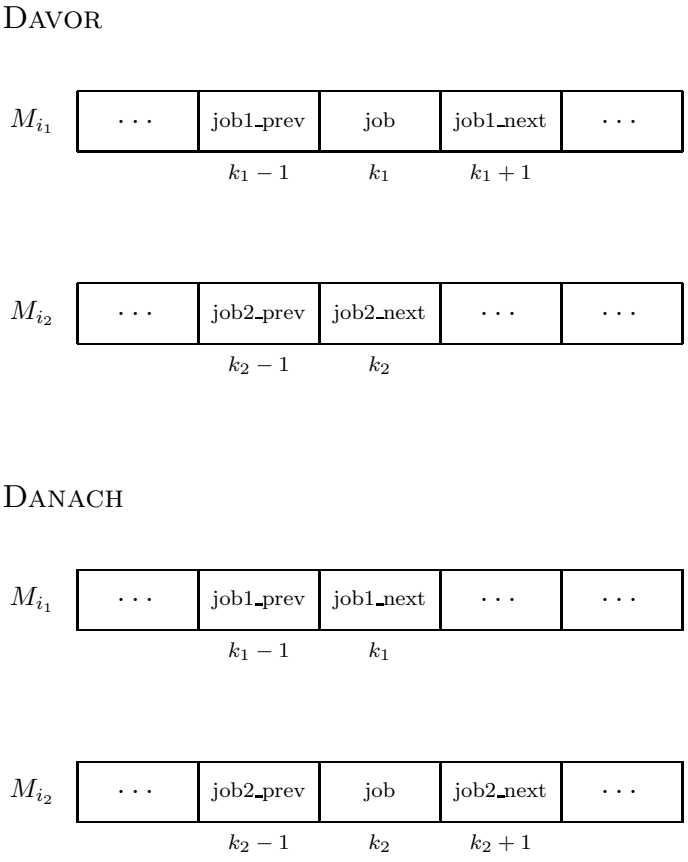


Abbildung 6.3: Einfügezug

Die Kosten eines Zuges (s, t) werden aufgespalten in die Kosten, die das Löschen vom Auftrag *job* auf Position s mit sich bringt (*Löschkosten* oder *del_cost*) und in diese, die mit dem Einfügen von *job* auf t verbunden sind. (*Einfügekosten* oder *ins_cost*). Lösch- und Einfügekosten können unabhängig voneinander berechnet werden, da ja $i_1 \neq i_2$ vorausgesetzt wird. Die Berechnung erfolgt in der Funktion *ins_move_cost* (*job*(.,.) ist der aktuelle Punkt, von dem ausgegangen wird):

job = job(i_1, k_1)

// Berechne *del_cost*

if ($n_{i_1} = 1$)
    del_cost = −job_cost(0,job,i_1);

else if($k_1 = 0$)
{
    job1_next = job($i_1, k_1 + 1$);
    del_cost = − job_cost(0,job,i_1)
               − job_cost(job,job1_next,i_1)
               + job_cost(0,job1_next,i_1);
}

else if($k_1 = n_{i_1}$)
{
    job1_prev = job($i_1, k_1 − 1$);
    del_cost = − job_cost(job1_prev,job,i_1);
}

else
{
    job1_prev = job($i_1, k_1 − 1$);
    job1_next = job($i_1, k_1 + 1$);
    del_cost = − job_cost(job1_prev,job,i_1)
               − job_cost(job,job1_next,i_1)
               + job_cost(job1_prev,job1_next,i_1);
}

// Berechne *ins_cost*

if($n_{i_2} = 0$)
    ins_cost = job_cost(0,job,i_2);

else if($k_2 = 1$)
{
    job2_prev = job(i_2, k_2);
    ins_cost = − job_cost(0,job2_prev,i_2)
               + job_cost(0,job,i_2)
               + job_cost(job,job2_prev,i_2);
}

else if($k_2 = n_{i_2} + 1$)
{
    job2_prev = job($i_2, k_2 − 1$);
    ins_cost = − job_cost(job2_prev,job2_prev,i_2)
}

else
{
    job2_prev = job($i_2, k_2 − 1$);
    job2_prev = job(i_2, k_2);

```
        ins_cost = − job_cost(job2_prev,job2_prev,i_2)
                   + job_cost(job2_prev,job,i_2)
                   + job_cost(job,job2_prev,i_2);
}
cost = del_cost + ins_cost;
```

Die Funktion *job_cost* wird also im „schlechtesten Fall“ 6-mal aufgerufen. Außerdem stellt die Funktion *ins_move_cost* fest, ob mit dem Tupel (s, t) überhaupt ein gültiger Zug vorliegt. Ist dies nicht der Fall, so wird der entsprechende Eintrag mit einem großen Wert (INVALID) beschrieben. Die erstmalige Berechnung des Feldes *insert_cost* erfolgt folgendermaßen:

```
for s=1 to n
{
      for t=1 to n + m
            insert_cost[s][t] = ins_move_cost(s, t)
}
```

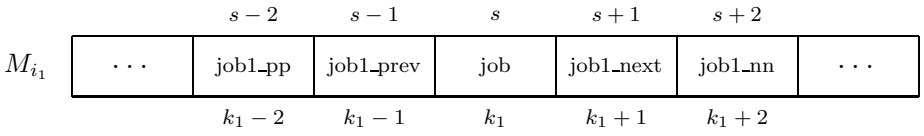
Das Update

Das Update in jeder Iteration soll natürlich weniger rechenaufwendig erfolgen als es bei der erstmaligen Berechnung nötig ist. Da nicht vorhersehbar ist, wie viele Züge veränderte Kosten haben, ist die Art, wie die Kosten bei den Austauschzügen aktualisiert wurden (siehe 6.4.2), unbrauchbar.

Im “smart update” von *ts*6 wird von der Tatsache Gebrauch gemacht, daß sich die Kosten in *del_cost* und *ins_cost* teilen lassen und sich viele Kosten in *insert_cost* auf gewisse Weise nur „verschieben“. Wie das Update für den Fall $i_1 < i_2$ funktioniert, soll nun kurz beschrieben werden. Es besteht aus folgenden Teilen:

1. Ausbessern und Verschieben der veränderten Löschkosten auf M_{i_1}
2. Neuberechnen und Verschieben der veränderten Einfügekosten auf M_{i_1}
3. Aktualisieren der ungültigen Züge
4. Neuberechnen der veränderten Löschkosten auf M_{i_2}
5. Neuberechnen der veränderten Einfügekosten auf M_{i_2}
6. Aktualisieren der ungültigen Züge

DAVOR



DANACH

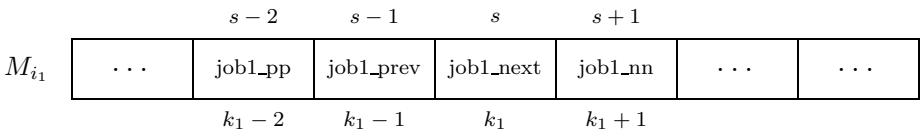


Abbildung 6.4: Löschen von *job*

1. **Ausbessern und Verschieben der Löschkosten**

   Was auf M_{i_1} geschieht, zeigt die Abb. 6.4.

   Durch das Löschen von *job* haben sich also die Kosten verändert, *job*1 oder *job*2 zu löschen, nicht aber die Kosten, diese Aufträge auf andere Maschinen einzufügen. Die Züge der Form $(s-1,.)$ und $(s,.)$. haben also einen neuen Löschkostenanteil *del_cost*, die Einfügekosten *ins_cost* bleiben gleich.

   Die bisherigen Kosten, *job_prev* zu löschen, waren

$$\begin{aligned}\text{del\_cost\_old1} = &- \text{job\_cost(job1\_pp,job1\_prev,}i_1\text{)}\\ &- \text{job\_cost(job1\_prev,job,}i_1\text{)}\\ &+ \text{job\_cost(job1\_pp,job,}i_1\text{)}\end{aligned}$$

   Die neuen, veränderten Kosten sehen folgendermaßen aus:

$$\begin{aligned}\text{del\_cost\_new1} = &- \text{job\_cost(job1\_pp,job1\_prev,}i_1\text{)}\\ &- \text{job\_cost(job1\_prev,job1\_next,}i_1\text{)}\\ &+ \text{job\_cost(job1\_pp,job1\_next,}i_1\text{)}\end{aligned}$$

   Die Differenz *del_cost_new*1 − *del_cost_old*1 kann also zu

$$\begin{aligned}\text{del\_cost\_change1} = &- \text{job\_cost(job1\_prev,job1\_next,}i_1\text{)}\\ &+ \text{job\_cost(job1\_pp,job1\_next,}i_1\text{)}\\ &+ \text{job\_cost(job1\_prev,job,}i_1\text{)}\\ &- \text{job\_cost(job1\_pp,job,}i_1\text{)}\end{aligned}$$

zusammengefaßt werden. Analog wird auch *del_cost_change*2 berechnet. War der alte Zug ungültig, so muß natürlich auch der entsprechende neue mit INVALID beschrieben werden. Das Feld *insert_cost* kann nun so ausgebessert werden (In C-Notation bedeutet a += b soviel wie $a = a + b$.):

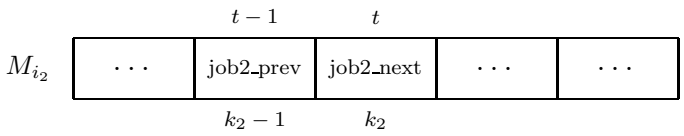
```
for q = 1 to n + m
{
    if(insert_cost[s − 1][q] != INVALID)
        insert_cost[s − 1][q] += del_cost_change1
    if(insert_cost[s][q] != INVALID)
        insert_cost[s][q] = insert_cost[s + 1][q] + del_cost_change2
    else
        insert_cost[s][q] = INVALID
}
```

Die folgenden Aufträge (ab der Position s+1) sind also um eine Position nach vorne gerückt. Dementsprechend muß auch das Feld *insert_cost* verschoben werden. Zuerst geschieht dies für die Löschpositionen:

```
for p = (s + 1) to (t − i2 − 3)
{
    for q = 1 to n + m
        insert_cost[p][q] = insert_cost[p+1][q]
}
```

Warum bis zur Position $t - i_2 - 3$ verschoben wird, ist aus Abb.6.5 ersichtlich.

DAVOR



DANACH

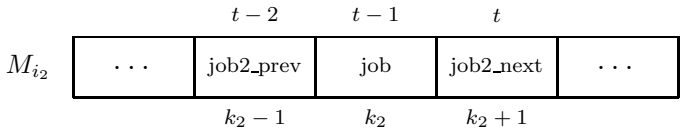


Abbildung 6.5: Einfügen von *job*

2. **Aktualisieren der ungültigen Züge**

   War M_{i_1} vor dem Löschen von *job* voll, so waren alle Züge, die auf dieser Maschine einen Auftrag eingefügt hätten, ungültig und sind nun wieder zulässig. In diesem Fall müssen sie neu berechnet werden:

```
if(n_{i_1} = K - 1)
{
    for p = 1 to n
    {
        for q = (s + i_1 - k_1) to (s + i_1 - k_1 + n_{i_1})
            insert_cost[p][q] = ins_move_cost(p,q)
    }
}
```

3. **Neuberechnen und Verschieben der Einfügekosten**

   Beim Einfügen liegen nur veränderte Kosten vor, wenn ein Auftrag auf (i_1, k_1) geschrieben werden soll – betroffen sind also die Züge der Form $(., s + i_1)$. Diese Kosten müssen komplett neu berechnet werden, da wegen des vorangegangenen Shiften ein Update analog zu den Löschenkosten nicht mehr möglich ist. (Wurden die Kosten in 2. bereits berechnet, so entfällt dieser Punkt.)

```
for p = 1 to n
    insert_cost[p][s + i_1] = ins_move_cost(p,s + i_1)
```

   Nun werden die Kosten auch für die Einfügepositionen „zurechtgerückt“ :

```
for q = (s + i_1 + 1) to (t - 1)
{
    for p = 1 to n
        insert_cost[p][q+1] = ins_move_cost(p,q+1)
}
```

4. **Neuberechnen der Löschkosten**

   Abb. 6.5 veranschaulicht, wie sich die Auftragskonfiguration auf M_{i_2} verändert. Zu beachten ist, daß sich die fortlaufende Numerierung der Aufträge anders als die maschineninterne verschiebt. (Es wurde ja ein Auftrag auf einer Position $s < t$ gelöscht!)

```
for q = 1 to n + m
{
    insert_cost[t - i_2 - 2][q] = ins_move_cost(t - i_2 - 2,q)
```

```
    insert_cost[t − i2 − 1][q] = ins_move_cost(t − i2 − 1,q)
    insert_cost[t − i2][q] = ins_move_cost(t − i2,q)
}
```

5. **Aktualisieren der ungültigen Züge**

   Wird M_{i_2} durch das Einfügen von *job* voll, so werden alle Züge, die einen Auftrag auf dieser Maschine einfügen sollen, ungültig:

```
if(n_{i2} = K)
{
    for p = 1 to n
    {
        for q = s − 1 − k2 to s − 1 − k2 + n_{i2}
            insert_cost[p][q] = ins_move_cost(p,q)
    }
}
```

6. **Neuberechnen der Einfügekosten**

   Neue Kosten gibt es für die Einfügezüge eines Auftrags vor und nach dem Auftrag *job*. (Dieser Abschnitt kann wieder entfallen, falls die Züge in 5. bereits als ungültig erklärt wurden.)

```
for p = 1 to n
{
    insert_cost[p][t − 1] = ins_move_cost(p,t − 1)
    insert_cost[p][t] = ins_move_cost(p,t)
}
```

Die anderen Fälle

Das oben beschriebene Update gilt für den allgemeinen Fall für $i_1 < i_2$ (der aus den Abb. 6.4 und 6.5 hervorgeht). Prinzipiell sind noch Fallunterscheidungen für die Position des Auftrags *job* auf M_{i_1} und M_{i_2} zu treffen (ist er beispielsweise erster, zweiter, vorletzter, letzter oder einziger Auftrag auf der Maschine?).

Rechenaufwand

Für das Update wird *job_cost* höchstens $64n + 8$ mal aufgerufen, der Rechenaufwand liegt also auch bei $O(n)$.

Ergebnisse

input file: tab300

number of iterations: 1000

length of tabu list: 40

f(initial solution): 152587
f(last solution) = 139500
f(best solution) = 139500 in iteration 89

During TS:

* 89 new best solutions found
* no tabu moves
* 911 moves with no change in objective

total improvement 8.576746 %

#(function calls of *job_cost*): 7217965

time used for TS: 39 seconds

Erfreulicherweise brachten die Einfügezüge eine Verbesserung von ca. 8.58 %. Mit etwa 7.2 Millionen Funktionsaufrufen von *job_cost* braucht *ts*6 0.10 % der Aufrufe von *ts*1, also ein wenig mehr als *ts*5, was sich aber aufgrund der fehlenden Diversifikation nicht in der Laufzeit auswirkt. Wie auch bei den Austauschzügen ohne eine solche Strategie wurden erst nur echte Verbesserungen erzielt und danach ausschließlich Nullkostenzüge gewählt.

6.7 ts7 – Eine Nachbarschaft ist nie genug

In *ts*7 werden erstmals beide Arten von Nachbarschaften für die Suche herangezogen. *ts*7 ist eine Kombination von *ts*5 und *ts*6, ohne vorerst die jeweiligen Suchmethoden aneinander anzupassen.

6.7.1 Die Suche

Nachbarschaft

Jeweils abwechselnd werden 100 Iterationen mit Austauschzügen und 100 Iterationen mit Einfügezügen durchgeführt.

Tabustrategie

Für die Austauschzüge ist die Tabustrategie von $ts4$ (siehe 6.4.1), für die Einfügezüge die von $ts6$ (siehe 6.6.1) implementiert. Die beiden Strategien ignorieren einander vorerst, d.h. $tabu1$ wird nur nach einem Austauschzug, $tabu2$ nur nach einem Einfügezug aktualisiert.

Diversifikation

Die Diversifikationsstrategie für Austauschzüge wird von $ts5$ (vgl. 6.5.1) übernommen. Einstweilen wird – wie bei der Tabustrategie – vernachlässigt, daß auch Punkte aus einer anderen Nachbarschaft gewählt werden.

6.7.2 Rechenaufwand und Ergebnisse

Rechenaufwand

Zusätzlich zu den $O(n)$ Funktionsaufrufen von *job_cost* pro Iteration, gibt es alle 100 Iterationen einen weiteren Aufwand von $O(n^2)$, da entweder *swap_cost* ($4n(n-1)$ Funktionsaufrufe) oder *insert_cost* ($6n^2$ Aufrufe) gänzlich neu berechnet werden muß.

Ergebnisse

input file: tab300

number of iterations: 1000

length of tabu list: 40

maximum number of jobs per machine: 100

f(initial solution): 152587
f(last solution) = 139190
f(best solution) = 139190 in iteration 701

During TS:

* 99 new best solutions found with swap moves
* 89 new best solutions found with insert moves
* 188 new best solutions found in total
* no tabu moves
* 812 moves with no change in objective
* 500 iterations with swap moves
* 500 iterations with ins moves

* penalty record was active 299 times

penalty record first active in iteration 402
improvement since initial solution 8.779909 %
#(function calls of job_cost): 8446421

time used for TS: 37 seconds

6.8 ts8 – advanced TS

*ts*8 baut großteils auf *ts*7 auf. Zusätzlich ist eine unterstützende Tabustrategie vorhanden, auf der auch eine neue Diversifikation der Austauschzüge beruht. Außerdem wird endlich erfolgreich gegen die vielen Nullkostenzüge angekämpft.

6.8.1 Die Suche

Die **Nachbarschaft** und **Tabustrategie** sind die aus *ts*7.

Intensivierungsphase

Bis jetzt wurde durch die Tabustrategie Aufträgen verboten, auf gewisse Positionen zurückzukehren – die Suche wurde also von gewissen Teilen von S ferngehalten. Die folgende Idee soll als Intensivierung dienen und der Suche die Möglichkeit geben, bestimmte Regionen von S genauer untersuchen zu können.

Das ($n \times n$ große) Feld tabu_sup[*job*][*pos*] gibt die Iteration an, in der der Auftrag *job* der Position *pos* zugewiesen wurde. Dort muß er nun mindestens für eine Iteration bleiben.

tabu_sup wird erst nach $\frac{2}{3}N$ Iterationen aktiviert. (N ist die Gesamtanzahl der Iterationen).

Diversifikation

Die bisherige Strategie hat die Suche ermuntert, Züge zu wählen, deren Komponentenaufträge selten vertauscht wurden. Diese Diversifikation soll das Auffinden neuer guter Punkte erleichtern, indem Aufträge in Positionen gezwungen werden, auf die sie noch nie oder selten geschrieben wurden.

Das Feld res_freq[*job*][*pos*] soll die Anzahl von Iterationen angeben, die der Auftrag *job* bereits in der Position *pos* verbracht hat (engl. *residence*

frequency). Die Aktualisierung von *res_freq* geschieht folgendermaßen: Wird ein Zug gewählt, bei dem der Auftrag *job* die Position *pos* verläßt, so wird

$$\text{res\_freq}[job][pos] += current\_iter - \text{tabu\_sup}[job][pos]$$

gesetzt.

Außerdem soll es nach einer großen Anzahl von Iterationen keinen Unterschied mehr machen, ob Aufträge eine oder zwei Iterationen länger in einer gewissen Postion waren. Dazu gibt es für *res_freq* Schranken (engl. *penalty levels*): Hat *job* bisher l Iterationen in *pos* verbracht, so hat *res_freq* eigentlich den Wert

$$\text{res\_freq}[job][pos] = PEN\_LEVEL * (l / PEN\_LEVEL)$$

wobei hier mit / die integer Division gemeint sein soll.

Die Strafterme werden wie bei der alten Strategie eingesetzt:

$$\begin{aligned} \text{cost} &= \text{swap\_cost}[p][q] + \\ &\quad PEN\_TIMES * (\text{res\_freq}[job1] + \text{res\_freq}[job2]) \end{aligned}$$

Das Aktivieren und Deaktivieren erfolgt wie bei der alten Diversifikation (siehe 6.5.1), nur bleiben die Strafterme auch noch für wenige Iterationen ($DET\_ALLOWED$) ausgeschaltet, wenn ein nicht-verbessernder Zug gewählt wird. (Da die Suche ja auf dem Weg zu einem besseren Punkt typischerweise „Hügel“ und „Plateaus“ zu überwinden hat.) Nach einer gewissen Anzahl von nicht-verbessernden Nachbarn aber kommt die Diversifikation wieder zum Einsatz.

Verbot der Nullkostenzüge

In *ts*8 sollen einfach nicht mehr als 5 Nullkostenzüge hintereinander gewählt werden dürfen. Insgesamt sind nur $\frac{N}{2}$ Nachbarn erlaubt, die keine Änderung in f bewirken.

6.8.2 Rechenaufwand und Ergebnisse

Der **Rechenaufwand** bleibt unverändert zu *ts*7.

Ergebnisse

input file: tab300

number of iterations: 1000

length of tabu list: 40

length of tabu supplement: 1
starting in iteration 666

maximum number of jobs per machine: 100

maximum number of 0-cost-moves in a row: 5

total number of 0-cost-moves allowed: 500

level for residence frequency penalty: 2

penalties multiplied by 5

f(initial solution): 152587
f(last solution) = 139016
f(best solution) = 139015 in iteration 899

During TS:

* 142 new best solutions found with swap moves
* 89 new best solutions found with insert moves
* 231 new best solutions found in total
* no tabu moves
* 225 "just-improving" moves
* 323 moves with no change in objective
* 221 deteriorating moves
* 500 iterations with swap moves
* 500 iterations with ins moves
* penalty record was active 149 times

improvement since initial solution 8.894598 %

#(function calls of job_cost): 8095791
time used for TS: 34 seconds

Die verbesserte Suche mit beiden Nachbarschaften konnte also etwas bessere Punkte als *ts*7 finden. Auffällig ist, daß die Anzahl echten Verbesserungen nur bei den Austauschzügen vergrößert wurde, mit Einfügezüge konnten – wie bei *ts*6 und *ts*7 89 neue beste Punkte erreicht werden.
Dies legt eine Diversifikation auch für Einfügezüge nahe!

6.9 ts9 – smart diversificaton

In der letzten vorhandenen Programmversion soll die Abfolge von Austausch- und Einfügezügen besser auf die Suche abgestimmt werden. Eine gemeinsame Tabu- und Diversifikationsstrategie für beide Arten von Nachbarschaft ist ebenfalls vorhanden.

6.9.1 Die Suche

Nachbarschaft

Die 2 Arten von Nachbarschaften wechseln einander nun nicht mehr nach einer vorgegebenen Anzahl von Iterationen ab, sondern je nachdem, welche Ergebnisse damit erzielt werden konnten. Genauer wird ein Nachbarschaftstyp wird für mindestens $LEAST\_ITER\_NUM$ Iterationen verwendet – danach noch solange bis die Lücke zwischen der aktuellen Iteration und der letzten echten Verbesserung kleiner als GAP_LIMIT1 ist. Diese Lücke darf im Laufe der Suche größer werden, indem sie

$$GAP\_LIMIT1 = FAC * current\_iter$$

mit $0 < FAC < 1$ gesetzt wird.

Tabustrategie

Die bisherige Tabustrategie für die Einfügezüge (beschrieben in 6.6.1) wird nun auch für Austauschzüge übernommen.

Die **Intensivierung** (unterstützende Tabustrategie) und das **Verbot von Nullkostenzügen** bleiben unverändert.

Diversifikation

Die Suche soll dazu gebracht werden, Aufträge auf solche Positionen zu schreiben, wo sie für den Wert von f neue Beiträge bringen können. Ausschlaggebende für die Kosten eines Auftrags sind ja, abgesehen vom Auftrag selbst, die Maschine und der vorangegangene Auftrag.

freq$[i][job_prev][job]$ gib an, wieviele Iterationen lang der Auftrag *job* bereits auf der Maschine M_i als Nachfolger von *job_prev* verbracht hat. Mit *tabu*2 erfolgt die Aktualisierung von *freq* analog zum Update von *res_freq* in *ts*8 (wie in 6.8.1 beschrieben). Ebenso ist die Art, wie und wann die Strafterme eingesetzt werden, und die Verwendung von *penalty levels* wie in *ts*8 implementiert.

6.9.2 Rechenaufwand und Ergebnisse

Rechenaufwand

Wie in *ts*7 und *ts*8 gibt es – zusätzlich zu den $O(n)$ Funktionsaufrufen pro Iteration – bei jedem Wechsel zwischen den Nachbarschaften einen Aufwand von $O(n^2)$. Allerdings ist in *ts*9 nicht vorhersehbar, wie oft der Wechsel geschieht.

Ergebnisse

input file: tab300

number of iterations: 1000

length of tabu list: 40

length of tabu supplement: 2
starting in iteration 666

maximum number of jobs per machine: 100

maximum number of 0-cost-moves in a row: 5

total number of 0-cost-moves allowed: 500

level for residence frequency penalty: 2

penalties multiplied by 5

penalty record started in iteration 283

f(initial solution): 152587
f(last solution) = 138861
f(best solution) = 138861 in iteration 956

During TS:

* 72 new best solutions found with swap moves
* 93 new best solutions found with insert moves
* 165 new best solutions found in total
* 62 tabu moves
* 377 "just-improving" moves
* 96 moves with no change in objective
* 362 deteriorating moves
* 790 iterations with swap moves
* 210 iterations with ins moves
* penalty record was not active for swap moves
* penalty record was active 19 times for insert moves
* 4 times changed between swap and insert moves

improvement since initial solution 8.998801 %

#(function calls of job_cost): 4262893
time used for TS: 34 seconds

*ts*9 brachte also eine Verbesserung von rund 9%, obwohl wurden insgesamt weniger echte Verbesserungen gefunden als in *ts*8. Allerdings wurden erstmals mehr neue beste Lösungen während eines Einfügezuges entdeckt. Dennoch wurden 790 Iterationen mit Austausch- und nur 210 Iterationen mit Einfügezügen durchgeführt. Dies legt die Vermutung nahe, daß die Diversifikation bei den Einfügezügen gute Dienste geleistet hat, (die Straffterme waren 19-mal aktiv). Nicht aktiviert wurden sie während eines Austauschzugs. Das Kriterium zum Aktivieren der Diversifikationsstrategie sollte daher für den jeweiligen Zugtyp eingens angepaßt werden.

Zwischen den Suchtypen wurde seltener gewechselt (in den beiden vorigen Versionen wurden jeweils immer 100 Iterationen mit einem Zugtyp durchgeführt – also 9-mal gewechselt). Das erklärt den geringeren Rechenaufwand im Vergleich zu *ts*7 und *ts*8.

6.10 Zusammenfassung

6.10.1 Wirksame Elemente

Es soll kurz zusammengefaßt werden, welche Strategien die Verbesserungen in den Programmen bewirkt haben und in *ts*9, der letzten Version, noch vorhanden sind.

Einfügezüge

Die größte Verbesserung der Ergebnisse hat in erster Linie die Einführung der Einfügezüge gebracht, die die Nachbarschaft vergrößert haben.

Wechsel zwischen den Nachbarschaften

In *ts*9 sind beide Zugtypen vorhanden – damit kann die Suche in weitere Teile von S vordringen. Den Wechsel abhängig vom Suchverlauf zu gestalten, hat sich ebenfalls bewährt.

Nullkostenzüge

Weitere Fortschritte wurden durch das Verbieten von zu vielen Nullkostenzügen erzielt, das erstmals in *ts*8 implementiert ist.

Diversifikation und Intensivierung

Auch die unterstützende Tabustrategie zusammen mit der Diversifikation der *residence frequency* waren gewinnbringende Neuerungen – sie wurden al-

lerdings in *ts*9 in veränderter Form übernommen, die auf die Struktur der Zielfunktion mehr Rücksicht nimmt: Für die Straffunktionen werden Aufträge nicht mehr auf ihren starren Positionen betrachtet, sondern mit dem Vorgänger und der Maschine, auf der sie sich befinden, in Bezug gebracht.

Schnelle Updates

Um den Rechenaufwand in Grenzen zu halten, ist die Verwendung der Felder *swap_cost* und *insert_cost* ausschlaggebend – dadurch konnten neue Strategien erst sinnvoll ausprobiert werden.

Strategien ohne Einfluß

Die Hinzunahme der *penalty levels* und das längere Deaktivieren der Strafterme nach einer Verbesserung hatte keinen erkennbaren Einfluß auf die Suche (die Ergebnisse änderten sich nicht, wenn die beiden Neuerungen weggelassen wurden).

Diese Ideen sind in *ts*9 noch implementiert und können möglicherweise bei weiteren Versionen besser genutzt werden.

Schlechte Diversifikation

Wenig geholfen hat die Diversifikationsstrategie, die in *ts*5 implementiert ist (vgl. 6.5.1). Dort wurden (Austausch-)Züge bestraft, deren Komponentenaufträge schon oft verschoben wurden. Diese Information hat scheinbar wenig Verbindung zu den Strukturen der tatsächlich besuchten Punkte.

6.10.2 Freie Parameter

Zum Überblick sollen kurz die Parameter aufgezählt werden, mit der die Suche (in der letzten Version *ts*9) gelenkt werden kann. In den Klammern darunter stehen jeweils die Variablen oder Funktionen, die im Programm die Parameter bestimmen. Der Wert, den die Variable in *ts*9 haben, wird ebenfalls angeführt.

- Tabustrategie
  - Anzahl der Iterationen, für die ein Attribut tabu-gesetzt wird ($tabu\_depth = 40$)
  - Zusammensetzung der Attribute
    (in den Funktionen *swap_tabu* für Austausch- und *ins_tabu* für

Einfügezüge; ein Zug ist jeweils tabu, falls eines seiner Attribute tabu-aktiv ist – siehe 6.6.1)

- Nullkostenzüge
  - Anzahl der hintereinander erlaubten Nullkostenzüge ($ZERO\_COST\_MAX = 5$)
  - Gesamtanzahl der erlaubten Nullkostenzüge ($NCH\_MAX = ts\_iter\_num/2$)
- Wechsel zwischen den Zugtypen
  wird in der Funktion *check_search*1 bestimmt:
  - Anzahl der erlaubten Iterationen ohne echte Verbesserung (GAP_LIMIT1; $FAC = 0.17$ – siehe 6.9.1)
- Intensivierung (unterstützende Tabustrategie)
  - Zeitpunkt (Iteration) des Aktivierens ($SUP\_TABU\_START = 2{*}ts\_iter\_num/3$)
  - Anzahl der Iterationen, die ein Auftrag festgehalten wird ($SUB\_TABU\_DEPTH = 2$)
- Diversifikation
  - Zeitpunkt der ersten Aktivierens
    (dieser wird mit der Funktion *check_search*2 bestimmt; $GAP\_LIMIT2 = 70$)
  - Steuerung des Aktivierens und Deaktivierens
    (geschieht in der Funktion *pick_swap*, bzw. *pick_ins*, in der die neuen Nachbarn mit Austausch-, bzw. Einfügezügen gewählt werden; $DET\_ALLOWED = 2$ – siehe 6.8.1)
  - konstanter Faktor der Strafterme ($PEN\_TIMES = 5$)
  - Höhe der Sprungstufen ($PEN\_LEVEL = 2$)
- maximale Anzahl von Aufträgen pro Maschine
  ($K = 100$)

6.10.3 Die Ergebnisse auf einen Blick

Ergebnisse der verschiedenen Versionen

	***ts*1**	***ts*2**	***ts*3** seed 482	***ts*3** seed 41	***ts*4**	***ts*5**	***ts*6**	***ts*7**	***ts*8**	***ts*9**
$f(s_0)$	152587	152587	152587	152587	152587	152587	152587	152587	152587	152587
$f(s\_best)$	142736	142736	149188	146176	142503	142420	139500	139190	139021	138856
gef. in Iteration	85	85	348	720	97	639	89	701	899	979
Verbesserung bez. $f(s_0)$	**6.46 %**	**6.46 %**	**2.23 %**	**4.20 %**	**6.61 %**	**6.66 %**	**8.58 %**	**8.78 %**	**8.89 %**	**9.00%**
echte Verbesserungen:	85	85	58	71	97	109	89	188	231	165
• mit Austauschzügen	85	85	58	71	97	109	–	99	142	72
• mit Einfügezügen	–	–	–	–	–	–	89	89	89	93
Funktionsaufrufe	$5.9\ 10^9$	$1.8\ 10^8$	$1.8\ 10^6$	$1.8\ 10^6$	$4.0\ 10^6$	$5.4\ 10^6$	$7.2\ 10^6$	$8.4\ 10^6$	$8.1\ 10^6$	$4.3\ 10^6$
reduziert zu	–	3.12 %	0.03 %	0.03 %	0.07 %	0.09 %	0.12 %	0.14 %	0.14%	0.07 %
Laufzeit $\sim$	11 Std.	1 Std.	20 sek.	20 sek.	40 sek.	40 sek.	40 sek.	40 sek.	40 sek.	20 sek.

1000 Iterationen
$n = 300$
tabu_depth = 40

Kapitel 7

Mögliche Erweiterungen

7.1 Überblick

In diesem Kapitel werden mögliche Erweiterungen beschrieben, die denkbar wären, um noch bessere Lösungen zu erzielen. Das betrifft sowohl Veränderungen im Programm als auch im zugrundeliegenden Modell.

Insbesondere wird gezeigt, wie das Problem mit gewissen Einschränkungen als Problem des Handlungsreisenden aufgefaßt werden kann.

7.2 Verbesserung der Suche

7.2.1 Aspiration

Bei Strategien, die die Suche einschränken, muß stets darauf geachtet werden, daß sie den Algorithmus nicht von guten Punkten fernhalten. Für jede solche Strategie sollte also sorgfältig eine Aspirationsmethode gewählt werden.

Beispielsweise kann folgende Überlegung als Aspiration für die Diversifikation von 6.9.1 dienen: $quality[i][job\_prev][job]$ soll den besten Wert der Zielfunktion angeben, der erreicht wurde, als der Auftrag job auf die Maschine M_i als Nachfolger von job_prev geschrieben wurde. Bringt ein entsprechender Zug einen besseren Wert, so werden sie Strafterme weggelassen.

7.2.2 TSP als Intensivierung

Betrachtet man das vorliegende Maschinenbelegungsproblem eingeschränkt auf nur eine Maschine M_i, so kann es auf folgende Weise als ein Problem des Handlungsreisenden gesehen werden:

Die Städte sind die Aufträge 0, job$(i,1)$,..., job(i,n_i) (0 fungiert als „unsichtbarer“ Auftrag vor dem ersten, bzw. nach dem letzten), die für Formulierung des TSP von 3.2.1 durchnumeriert werden (also $S = \{0,\ldots,n_i\}$). Die Kostenmatrix C ist dann

$$\begin{aligned} c_{jl} &= \text{job\_cost}(i,j,l), \quad j = 0,\ldots,n_i \text{ und } l = 1,\ldots,n_i \\ c_{j0} &= 0 \text{ für } j = 0,\ldots,n_i \end{aligned}$$

Die billigste Route würde also die günstigste Reihenfolge der Aufträge auf einer Maschine festlegen (wobei der erste Auftrag jener ist, der in der Route nach der „Stadt“ 0 kommt).

Eine mögliche Form der Intensivierung wäre es also, während der Suche ab und zu für jede Maschine M_i ein TSP der obigen Form zu lösen.

Bis zum jetzigen Zeitpunkt war mir nur C-Code verfügbar, der TSP's für bis zu 10 Städten lösen kann. Bei entsprechenden Eingaben von $n = 100$ Aufträgen hat sich gezeigt, daß TSP-Phasen während der Suche die Aufträge auf den Maschinen in der Reihenfolge belassen, in die durch die Züge der Tabusuche gebracht wurden. Bei Problemen dieser Größe findet also bereits die Tabusuche die optimale Auftragskonfiguration innerhalb einer Maschine.

7.2.3 Rechenaufwand

Naheliegend wäre es, *swap_cost*, bzw. *insert_cost* auch nach einem Einfüge- bzw. Austauschzug zu aktualisieren., damit die Suche flexibel zwischen den beiden Nachbarschaftstypen wechseln kann, ohne den Rechenaufwand zu erhöhen. Für *insert_cost* ist es auf den ersten Blick naheliegend, einfach die Kosten der veränderten Einfügezüge mit der Funktion *ins_move_cost* neu zu berechnen. Im allgemeinen Fall eines Austauschzugs (siehe Abb. 6.2) gibt es auf beiden Maschinen jeweils 3 veränderte Lösch- und 2 veränderte Einfügepositionen. Insgesamt sind $6n+4(n+m)$ Züge betroffen. Da für *ins_move_cost* bis zu 6 Funktionsaufrufen von *job_cost* nötig sind, ergibt dies einen Aufwand von $60n + 24m$. Bei $n = 300$ ist die Verwendung dieses Updates nur sinnvoll, wenn nicht mehr als 30 Iterationen von Austauschzügen hintereinander ausgeführt werden.

Für ein gutes Update von *swap_cost* muß bestimmt werden, wie sich Kosten nach einem Einfügezug verschieben.

Es sollte also für beide Kostenfelder ein “smart update” ähnlich der Aktualisierung von *ins_cost* nach einem Einfügezug geschrieben werden.

7.2.4 Abfolge der Zugtypen

Qualitätsabhängiger Wechsel

Das Kriterium, das den Wechsel von einem Zugtypen zum anderen bestimmt, könnte nicht nur die Häufigkeit, sondern auch die Qualität der echten Verbesserungen berücksichtigen.

Vereinigung der Nachbarschaften

Gäbe es ein schnelles Update für die Kostenfelder der beiden Zugtypen nach dem jeweils anderen Zug, so könnte man in jeder Iteration beide Nachbarschaften durchsuchen, und den besten unter allen möglichen Zügen wählen.

7.2.5 Abbruchkriterium

Alle beschriebenen Programmversionen terminieren nach einer vorgegebenen Anzahl von Iterationen. Es wäre aber auch denkbar, stattdessen folgende Idee zu verfolgen:

Lücken

Man betrachtet die Lücken bezüglich der Iterationen und der Zielfunktion zwischen den gefundenen echten Verbesserungen.

Es wird erwartet, daß die Differenz d_1 im Wert der Zielfunktion von zwei aufeinander folgenden besten Punkten im Laufe der Suche immer kleiner, die Anzahl d_2 der dazwischen verstrichenen Iterationen hingegen immer größer wird.

Zeitpunkt des Abbruchs

Man will nun den Zeitpunkt finden, ab dem die Lücken d_1 „zu klein“ oder d_2 „zu groß“ sind, d.h. die Suche zu langsam geworden ist und sich ein Mehraufwand für nur sehr geringe Verbesserungen nicht mehr lohnt.

Qualität

In diesem Sinne kann die Suche beispielsweise abhängig von d_1 (der Qualität der Verbesserungen) abgebrochen werden: Ein besonders einfaches Kriterium ist es, für d_1 eine untere Schranke D festzusetzen und abzubrechen, sobald $d_1 \leq D$ ist.

Ein andere Möglichkeit ist es, nach einem „großen“ d_1 wieder mehrere „kleine“ d_1 zu erlauben.

Quantität

Zieht man den Wert d_2 (die Häufigkeit der Verbesserungen) heran, um den Zeitpunkt des Abbruchs zu bestimmen, so kann man analoge Überlegungen zum qualitätsabhängigen Abbruch anstellen – wie z.B. eine *obere* Schranke für d_2 festsetzen.

Kombination

Natürlich ist es auch denkbar, die beiden Arten zu kombinieren, wo etwa ein günstiger Wert von d_1 einen schlechteren Wert von d_2 zuläßt.

Analyse des Suchverlaufs

Die Abbildung 7.1 zeigt Zeitpunkt und Fortschritt der gefundenen echten Verbesserungen nach 1000 und 10000 Iterationen von *ts*9.

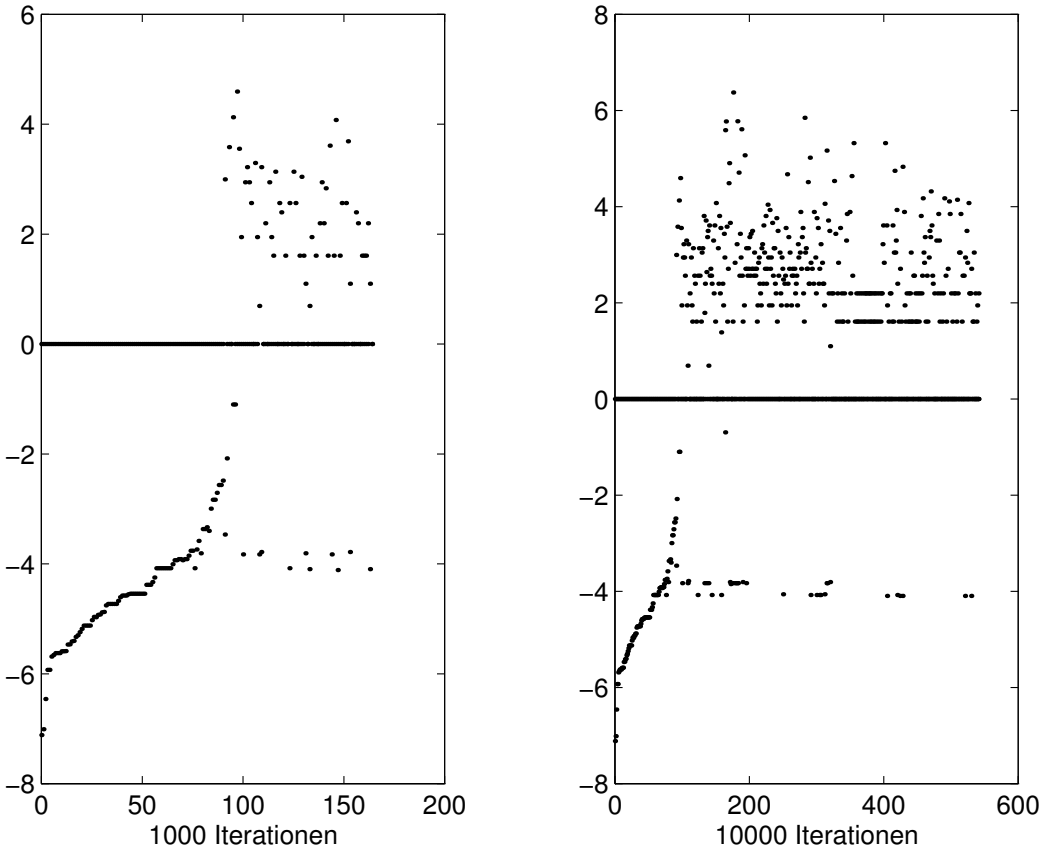


Abbildung 7.1: Lücken

Auf der x-Achse ist die Anzahl der echten Verbesserungen aufgetragen – für jeden solchen Punkt wird auf der y-Achse der der Wert $\log(d_2)$ vermerkt. Zusätzlich wird die Zahl $-\log(d_1)$ eingetragen.
Während der ersten ca. 90 Iterationen (wo nur echte Verbesserungen gefunden wurden und $\log(d_2) = 0$, also $d_2 = 1$ ist) nimmt d_1 sukzessive ab.

Danach ist d_1 entweder gleich 1 (was die kleinstmögliche Verbesserung bedeutet) oder $\log(d_1) \approx 4$, also $d_1 \approx 50$.

Die Iterationslücken d_2 werden (nach den erste 90 Iterationen) nicht wie erwartet immer größer, sondern schwanken stark (hauptsächlich zwischen den Werten $d_2 \approx 6$ und $d_2 \approx 50$). Gegen Ende der 10000 Iterationen nimmt d_2 sogar tendenziell ab.

Erfahrungswerte

Abb. 7.1 zeigt, daß sich der Wert d_1 besser eignet, den Zeitpunkt des Abbruchs bestimmen:

Die Iterationslücken d_2 werden zwar gegen Ende der 10000 Iterationen wieder kleiner – die entsprechenden d_1 bedeuten allerdings meist nur mehr eine Qualitätverbesserung von $d_1 = 1$, was bei den bisherigen Werten der Zielfunktion weniger als 0.01 % zusätzliche Verbesserung ausmacht. Eine so kleine Verbesserung ist auch bei kleineren d_2 nicht mehr wünschenswert.

Das Abbruchkriterium sollte daher nur von d_1 abhängig sein.

Naheliegende Kriterien

Bei einem ersten, einfachen Kriterium kann man abbrechen, nachdem $d_1 = 1$ für eine gewisse Anzahl von Iterationen war.

Oder man kann eine maximale Anzahl von *aufeinanderfolgenden* Iterationen angeben, in denen $d_1 = 1$ erlaubt ist.

7.3 Erweiterungen im Modell

7.3.1 Gleichmäßige Auslastung der Maschinen

Gewünscht wäre eine „möglichst gleichmäßige Auslastung“ der Maschinen: an jeder Maschine soll etwa die gleiche Länge an Kabeln produziert werden.

Bei Programmversionen, wo nur Austauschzüge implementiert sind, wird auf die gleichmäßige Auslastung nur in der Anfangslösung Rücksicht genommen. Bei Einfügezügen gibt es eine maximale Anzahl von Aufträgen, die pro Maschine erlaubt sind. Stattdessen wäre eine maximale Gesamtlänge von

Kabeln pro Maschine denkbar (die etwa der Gesamtlänge der Auftragsliste durch die Anzahl der Maschinen entspricht).

7.3.2 Produktionszeiten

In der bisherigen Modellierung wurde die terminliche Situation der Aufträge noch vernachläßigt. Gibt es allerdings für jeden Auftrag j einen Zeitpunkt d_j, an dem er fertiggestellt sein muß (eine *Deadline*), so kann dies folgendermaßen berücksichtigt werden:

Es wurden bereits Funktionen implementiert, die für eine zulässige Belegung den Zeitpunkt des Produktionsbeginns und der Fertigstellung eines Auftrags $j = job(i, k)$ festsetzten und diese in ein Feld prod_time$[i][k]$.start, bzw. prod_time$[i][k]$.start schreiben.

Die Funktion *calc_prod_time* berechnet *prod_time* für einen gegebenen Punkt komplett, *update_prod_time* kann die Zeiten nach einem Austauschzug aktualisieren.

Um termingerechte Belegungen zu erhalten, könnte man Verletzungen der Deadlines ($d_j <$ prod_time$[i][k]$.end) als Strafterme in die Zielfunktion verpacken.

Literaturverzeichnis

[1] F. Glover, M. Laguna. *Tabu Search*, Kluwer Academic Publishers, 1997.

[2] K. Dowsland. *Simulated Annealing* pp.20 - 69 in C. R. Reeves (Editor). *Modern Heuristic Techniques for Combinatorial Problems*, McGraw-Hill International, 1995.

[3] E. Aarts, J. Korst, P. van Laarhoven. *Simulated Annealing*, pp. 91 - 120 in E. Aarts, J. K. Lenstra (Editors). *Local search in combinatorial optimization*, Wiley, 1997.

[4] F. Glover, M. Laguna. *Tabu Search* pp. 70 - 150 in C. R. Reeves. Modern Heuristic Techniques for Combinatorial Problems, McGraw-Hill International, 1995.

[5] A. Hertz, E. Taillard, D. de Werra. *Tabu Search*, pp. 121 - 136 in E. Aarts, J. K. Lenstra (Editors). *Local search in combinatorial optimization*, Wiley, 1997.

[6] W. Domschke, R. Klein, A. Scholl. *Tabu Search – Durch Verbote zum schnellen Erfolg*, publiziert als *Taktische Tabus, Tabu Search - Durch Verbote schneller optimieren.*, CT - Magazin für Computertechnik, Heft 12/1996. `http://www.bwl.tu-darmstadt.de/bwl3/forsch/projekte/tabu/paper.htm`

[7] C. R. Reeves. *Genetic Algorithms* pp. 151 - 188 in C. R. Reeves. *Modern Heuristic Techniques for Combinatorial Problems*, McGraw-Hill International, 1995.

[8] H. Mühlenbein. *Genetic Algorithms*, pp. 137 - 172 in E. Aarts, J. K. Lenstra (Editors). *Local search in combinatorial optimization*, Wiley, 1997.

[9] C. R. Reeves, J. E. Beasley. *Introduction* pp. 1 - 19 in C. R. Reeves. *Modern Heuristic Techniques* pp. 1 - 25 in V. J. Rayward-Smith, I. H.

Osman, C. R. Reeves, G. D. Smith (Editors). *Modern Heuristic Search Methods*, John Wiley & Sons, England, 1996.

[10] S. U. Thiel. *Overview of Modern Heuristic Methods*, WWW-Dokument, `http://www.cs.cf.ac.uk/User/S.U.Thiel/ra/section3_7.html`

[11] P. Brucker. *Scheduling Algorithms*, Springer Verlag, 1998.

[12] M. Pinedo. *Scheduling – Theory, Algorithms and Systems*, Prentice-Hall, 1995.

[13] M. Yannakakis. *Computational Complexity*, pp. 19 - 56 in E. Aarts, J. K. Lenstra (Editors). *Local search in combinatorial optimization*, Wiley, 1997.

[14] A. Schaerf. *Tabu search techniques for large high-school timetabling problems*, CWI, Amsterdam, 1996.

[15] P. Brucker, S. Knust. *Complexity results of scheduling problems*, WWW-Dokument,
`http://www.mathematik.uni-osnabrueck.de/research/OR/class`

[16] H. W. Lenstra. *Integer programming with a fixed number of variables*, pp.538 - 362 in Mathematics of Operations Research 8, 1983.

Abschließend möchte ich allen herzlich danken, die mich bei der Anfertigung dieser Arbeit unterstützt haben. Besonderer Dank gilt Prof. Dr. Arnold Neumaier und Prof. Dr. Fred Glover.

LEBENSLAUF

Ulrike Schneider

persönliche Daten

Geburtsdatum	25. April 1975
Geburtsort	Pretoria, Südafrika
Adresse	1180 Wien, Wallrißstraße 62/4

Schulbildung

1981 - 1985	Volksschule, Wien 18
1985 - 1993	neusprachliches Gymnasium, BG 18, Klostergasse
Juni 1993	Matura mit Auszeichnung

Studium

WS 93	Beginn des Diplomstudiums Mathematik an der *Universität Wien*
Juni 95	Beenden des ersten Studienabschnitts mit ausgezeichnetem Erfolg
WS 96/97	Auslandssemester (*Joint Study*) an der *Macquarie University* in Sydney, Australien
Aug. 97 – Juni 98	Mitarbeit am FWF-Projekt GLOPT über globale Optimierung
seit SS 98	Tutorin am Institut für Mathematik
Okt./Nov. 98	Auslandsaufenthalt für die Diplomarbeit (*kurzfristige wissenschaftl. Aufenthalte im Ausland*) an der *University of Colorado* in Boulder
Dez. 98	Abschluß des zweiten Studienabschnitts mit ausgezeichnetem Erfolg