VU Computational Techniques
Programming Exercise C (Thursday Group)

M. Leitner

16. Dezember 2015

Diese Programmieraufgabe widmet sich dem *Tool Switching Problem*, einem Schedulingproblem welches bei flexiblen automatisierten Fertigungsanlagen in der Praxis auftritt.

Das Tool Switching Problem

Gegeben sind n *Jobs* $J = \{1, \ldots, n\}$ deren Abarbeitungsreihenfolge auf einer Maschine optimiert werden soll. Zur Abarbeitung der Jobs sind *Tools* $T = \{1, \ldots, m\}$ erforderlich (wie z.B. Bohrer, Schrauber). Konkreter beschreibt eine Boolsche $m \times n$ Matrix A welche Tools von welchem Job benötigt werden, d.h. $A_{i,j}$ mit $i = 1, \ldots, m$ und $j = 1, \ldots, n$ ist TRUE genau dann wenn Tool i benötigt wird um Job j abzuarbeiten.

Das Problem ist nun, dass die Fertigungsmaschine ein Magazin für nur eine bestimmte Anzahl $C < m$ (*Magazine Capacity*) an Tools besitzt und ein Toolwechsel (*Tool-Switch*) mit einem relativ großen Zeitaufwand (d.h. Kosten) verbunden ist. Grundsätzlich kann aber jedes Tool in jeder der C Magazin-Positionen montiert werden, und wir nehmen vereinfacht an, dass jeder Tool-Switch immer gleich aufwändig ist.

Gesucht ist daher (a) eine Reihenfolge (Permutation) in der alle Jobs J abgearbeitet werden sollen und (b) eine dazu passende Sequenz von Tool-Konfigurationen $T_1, \ldots T_n$, $T_j \subset T$, $|T_j| \leq C$, sodass jeweils die für die Abarbeitung eines jeden Jobs notwendigen Tools zu Verfügung stehen und die Gesamtanzahl der Tool-Switches minimal ist. Achtung: Die Montage der allerersten Tool-Konfiguration T_1 betrachten wir noch nicht als Tool-Switch, d.h. diese fließt nicht in die Zielfunktion ein.

Ihre Aufgabe

1. Versuchen Sie zunächst für folgendes Teilproblem einen effizienten Algorithmus zu finden: Für eine *vorgegebene* Reihenfolge aller Jobs soll eine Sequenz von Tool-Konfigurationen mit einer minimalen Anzahl von Tool-Switches bestimmt werden. Ein solcher Algorithmus kann später einiges vereinfachen.

2. Überlegen Sie sich eine möglichst sinnvolle *greedy* Konstruktionsheuristik zur Lösung des Gesamtproblems und implementieren Sie diese.

3. Definieren Sie zumindest zwei unterschiedliche, möglichst sinnvolle Nachbarschaftsstrukturen und verwenden Sie diese jeweils in einer einfachen lokalen Suche. Realisieren Sie mindestens eine der Schrittfunktionen *next improvement* oder *best improvement*. Hinweis: Evtl. können Sie die Performance ihres Algorithmus durch inkrementelle Bestimmung der Zielfunktionswerte von Nachbarlösungen verbessern. Die Startlösung für die lokale Suche soll von der Konstruktionsheuristik geliefert werden.

4. Testen und vergleichen Sie die implementierten Varianten der lokalen Suche mit Benchmark-Instanzen, die Sie von `http://www.unet.edu.ve/~jedgar/ToSP/ToSP.htm` bekommen können. Welche Nachbarschaft liefert wie gute Durchschnittsergebnisse? Verwenden Sie für Ihre Tests zumindest die jeweils ersten Instanzen der Klassen $4\zeta_{10}^{10}$, $15\zeta_{30}^{40}$ und $20\zeta_{40}^{60}$ und achten Sie auch darauf, dass insbesondere für stochastische Algorithmen über mehrere Läufe gemittelt werden muss, um aussagekräftige Ergebnisse zu erhalten.

5. Kombinieren Sie Ihre beiden Nachbarschaftsstrukturen in einer *Variable Neighborhood Descent* (VND) Strategie und testen Sie diese. Ist diese Kombination empfehlenswerter als die jeweils einfachen lokalen Suchalgorithmen?

6. Gehen Sie nun weiters *wahlweise* (alternativ) wie folgt vor. Natürlich ist auch diese Strategie zu testen und mit den bisherigen einfacheren Ansätzen zu vergleichen.

   (a) Randomisieren Sie Ihre Konstruktionsheuristik und implementieren Sie eine *Greedy Randomized Adaptive Search Procedure* (GRASP).

   (b) Betten Sie Ihre beste lokale Suche oder das VND in eine *Generalized Variable Neighborhood Search* (GVNS) Strategie ein.

   (c) Falls Sie andere, sinnvolle, Erweiterungsideen sind wir auch offen für diese. Sprechen Sie im Zweifelsfall Ihre Vorstellungen einfach mit uns ab!

Sie können die Aufgabe grundsätzlich in einer beliebigen Programmiersprache lösen.

WICHTIG: Eine Benutzeroberfläche ist *nicht* notwendig oder erwünscht. Ihr Programm muss jedoch neben der Endlösungen jeweils zumindest die folgenden Informationen ausgeben:

- Die Iterationen, in denen verbesserte Lösungen gefunden werden und die dabei erreichten Lösungswerte
- Gesamtanzahl der Iterationen und untersuchte Nachbarlösungen

Fassen Sie weiters die entwickelten Algorithmen, Erkenntnisse und Ergebnisse in einem kurzen *Bericht* sowie einer kurzen Präsentation zusammen. Den Bericht, sowie den Source-Code Ihres Programms senden Sie bitte **bis 24.1.2016 an markus.leitner@univie.ac.at**. Gehen Sie insbesondere auf die gewählten Nachbarschaftsstrukturen und eventuelle Besonderheiten ein. Zu den Testergebnissen ist im Bericht jedenfalls eine Interpretation bzw. Zusammenfassung notwendig. Zusätzlich können auch detaillierte Ergebnisse in Tabellenform angegeben werden. Die **Präsentationen der verschiedenen Algorithmen finden in der Vorlesung am 28.1.2016 statt**.

Die Aufgabe sollte grundsätzlich *zu zweit* oder alleine gelöst werden. Die Punkte 1-5 der Aufgabe sind gemeinsam zwei Punkte wert. Ein dritter Punkt kann durch Erfüllung von Punkt 6 erreicht werden.

Viel Erfolg!