ADS - Heuropt - Abgabe 1

Kern, Weichselbaum

December 6, 2009

The first problem that we encountered when working on the tool switching problem was to find the cheapest tool assignment, meaning lowest number of tool switches possible, for a given job sequence.

Doing some research, we discovered that there already existed a well known method called **KTNS** which solved our first problem quite well. KTNS is an algorithm which was developed by Tang and Denardo in 1988. It is based on the idea of keeping and/or inserting the tool, which is needed the soonest, in/to the magazine configuration.
Although there are some improvements available to the KTNS in terms of runtime performance, we *didn't need such an improvement* because fortunately enough, we aren't spending much time in our KTNS class.

Then we developed a **greedy** algorithm, which we applied on a *job-smiliarity matrix*. There we choose the biggest (ie. the one requiring the most tools) job first. The selection of the following jobs is based on the jobs similarity. The most similar job with the least difference is chosen next until there are no jobs left for selection.

The next task was to find and construct two or more rather reasonable and different **neighbourhood functions**, which have to be used in the local search later on. For this neighbourhood structures we had to implement **random neighbour, next neighbour and best improvement**. Here the only freedom we got was the selection of the appropriate neighbourhood function, since the step functions algorithm are designed rather straight forward and do not leave much room for any improvement. The implemented neighbourhood structures are PairSwitch, BinSwitch, Rotation and TwoOpt.

PairSwitch: PairSwitch is a neighbourhood function where the neighbours of a solutions job sequence are defined by the exchange of two jobs.

Rotation: In the rotation neighbourhood, jobs are rotated triplet wise. The last job of the triplet is moved to its beginning and the other two jobs are shifted right. This step is performed from the beginning to the end of the job sequence as long as three jobs are available in the next step.

BinSwitch: Here, the jobs are divided into N bins. After that, all bin combinations are created. Ie. 1—2—3 results in 1—3—2, 3—1—2 etc. This is heavily used in the VND. N is passed into the function, meaning, we are flexible about the size of the bin and the number of bins.

TwoOpt: Enhancement of the PairSwitch. First a regular PairSwitch is done, after that, another PairSwitch is applied to each of the solutions of the first PairSwitch.

Our **VND** is customised. We apply PairSwitch, TwoOpt and Rotation onto the solution of the GRASP. The order of the solution doesn't play a big part of getting our best solution. What helps is the BinSwitch.
If no better solution is found after all the neighbourhood functions are applied, we randomly BinSwitch (N is a number between 4 and 20 for the big instance). VND improves our GRASP a lot (from 270-280 to 216) and is also better than a local search, no matter which neighbourhood function is used for the local search (best LS result: 235). The VND runs infinitely. Downside is the insane number of iterations, which basically depends on a manual stop. Our VND is a never ending process.

Next we randomised our construction heuristic and implemented a **GRASP**. We start with N randomised greedy heuristics, where N depicts the number of cores (we are using Java's concurrency namely Threads). Again, we take advantage of our job similarity matrix to order the jobs and calculate the costs between them. The jobs are stored in a *Cl*, which gets filtered into a *RCL*. Our best results are with an *alpha* set to around 0.65 ± 0.1.
In the end, the best of the N randomised greedy heuristics is selected.

Our results:

Matrix:	10 * 10	30 * 40	40 * 60
Best cost:	10	104	216
Iterations:	47	214	2875