Permutation Flow Shop Problem

Dávid Ocetník
Thursday, 16:15
Open Informatics
ocetndav@fel.cvut.cz

***Abstract*—The objective of this paper is to demonstrate method for a production scheduling problem known as permutation flow shop problem (PFSP).**

I. Assignment

A. Problem Statement

A production scheduling problem can be described as follows. Given N jobs (i = 1, 2, ..., N) with processing times which have to be processed on M different machines (j = 1, 2, ..., M). Moreover, a job cannot be processed by more than one machine at a time and a machine can process at most one job at a time. A schedule is an optimal allocation of jobs to machines over time, in order to minimize sum of completion times of all jobs (makespan).

In flow shop problem a job can start processing on machine j, only after completing its operation on machine (j-1). All jobs follow the same machine order but the job order each machine may differs. Thus for each machine we need to find a job order. Permutation flow shop problem is a special case of flow shop problem with an additional constraint that the job order is same on all machines. Gantt charts are used to graphically represent a schedule. X-axis represents the time and Y-axis represents machines.

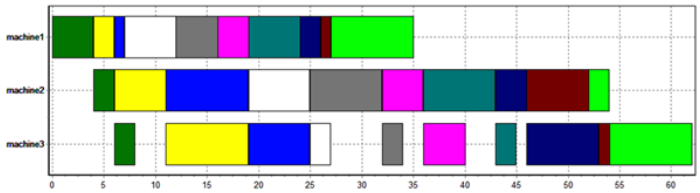

Fig. 1: First solution

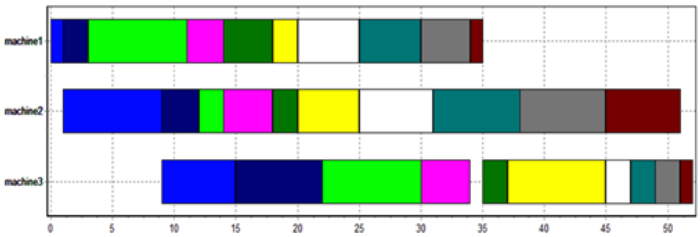

Fig. 2: Optimal solution

B. Problem Categorization

The permutation flow shop problem belongs to the class of NP-hard problems in the strong sense which remains probably one of the most computationally unsolvable combinatorial problem at present. Therefore it seems to be highly computational hard to find optimal solution. Only small instances can be exactly solved.

II. Related works

This problem can be solved in many different ways.

A. Heuristics

Widmer and Hertz [22] show heuristics method named *Spirit* which is composed of two phases (the first finds an initial sequence using an analogy with the travelling salesman problem and the second tries to improve this solution using taboo search techniques). Next heuristics present Nawaz et al. [16], Ruiz et al. [21] and Koulamas [8]. More complex techniques are tabu search, genetic algorithms, shifting bottleneck procedure, ant colony algorithm and branch and bound.

B. Tabu search

Nowicki and Smutnicki [17] published tabu search technique with a specific neighborhood definition and executed 10x10 hard benchmark problem. Other tabu search technique described Paula deSzoeke and Faith Barnard [11].

C. Genetic algorithms

Commonly used heuristics such as shortest processing time (SPT) and earliest due date (EDD) can be used to calculate a feasible schedule quickly, but usually do not produce schedules that are close to optimal in these job shop environments. Manikas et al. [13] demonstrate genetic algorithms which can be used to produce solutions in times comparable to common heuristics but closer to optimal. Murata et al. [14], in order to improve the performance of the genetic algorithm, examine the hybridization of the genetic algorithms. They show two hybrid genetic algorithms: genetic local search and genetic simulated annealing.

D. Shifting bottleneck procedure

Balas et al. [2] developed a new variable depth search procedure, GLS (Guided Local Search), based on an interchange scheme and using the new concept of neighborhood trees. Structural properties of the neighborhood are used to guide the search in promising directions. While this procedure competes successfully with others even as a stand-alone, a hybrid procedure that embeds GLS into a Shifting Bottleneck framework and takes advantage of the differences between the two neighborhood structures proves to be particularly efficient.

E. Ant colony

Blum and Sampels [3] developed an ant colony optimization approach, which uses a strong non-delay guidance for constructing solutions and which employs black-box local search procedures to improve the constructed solutions. They compare this algorithm to an adaptation of the tabu search by Nowicki and Smutnicki [17] and show that their algorithm improves the best known solutions for 15 of the 28 tested instances.

F. Branch and bound

Method that occurs in the most papers is definitely Branch and bound. Authors who deal with it are e.g. Carlier and Rebaï [5], C.N. Potts [20], Brucker et al. [4], Carlier and Pinson [19], Chung et al. [7], [6], Haouari and Ladhari [12], [9], Lageweg et al. [1], Madhushini et al. [15], Okamoto et al. [18], Le Pape et al. [10], and many others.

III. Problem solution

A. Design

In this paper I focus on the exact Branch and bound method that is described by Carlier and Rebaï [5] in section 4. It's a variation of the depth-first search strategy and it consists in recursively enumerating jobs potentially processed first (inputs) and jobs which would be processed last (outputs) on machines. At each node of the search-tree, I fix one job as input or output of machines, i.e. I fix a job to be processed before or after all others.

At the beginning I compute initial values for root node as it is described in Carlier and Rebaï paper [5]. Then I define *M "set_heads"*, *M "set_tails"*, *M "set_processing_times"* and a *lower bound LB* at each node k. A *set.head* vector is the vector of dates prior to which the associated machine is not available. The word *"set"* is introduced here because these heads are common to all jobs of J. A *"set_tail"* vector is the dual case of a *"set_head"* vector. It is related to tails of jobs of set J. A *"set_processing_time"* vector is the sum of processing times of set J jobs on each machine.

From these three values I compute lower bound for each node of search tree. If node hasn't got any descendants, I'm backtracking. Otherwise I choose the descendant with smallest lower bound and I'm branching again from this node. During backtracking I'm searching closest unbranched node up to root. If I found that node, I'm branching from it.

The reason why I choose this method is, as Carlier and Rebaï [5] wrote in their paper, this new approach gave better results than the first method they are described, because we disregard disjunctions and consider sets of jobs instead. Therefore larger problems could be solved optimally.

B. Implementation

For implementation of my solution I use Java. Algorithm is highly inspired by those which are described by Carlier and Rebaï paper [5] and Haouari and Ladhari [9].

Notice, that I compute upper bound (I don't set it to e.g. *Integer.MAX_VALUE*). First I sort jobs by sum of their processing times. If there are more jobs combinations with the same sum of their processing times I create permutations. So I get initial feasible schedules. For each initial feasible schedule I compute makespan and then I set upper bound to lowest makespan of initial feasible schedules.

As I wrote above, at the beginning of algorithm I initialize root node with values computed as described Carlier and Rebaï [5]. In each node I store unbranched descendants and unscheduled jobs. Also jobs added to head, jobs added to tail, set_head, set_tail set_processing_time, lower bound, parent of node, job id and boolean value which determines if job will be added to head or to tail is stored there.

From this root node I start branching. It is needed to check if lower bound of newly created nodes is not greater than actual upper bound. In the case it's not true, we don't need to branch from this node anymore. The same case is when we are in leaf. Then we are going back to the root (backtracking) and we are searching closes unbranched node. Until we reach the root again. Then we write best minimum makespan which we store every time when we find feasible solution. Makespan is equal to lower bound of the leaf node.

```
// calculation of setHead vector
private static int[] computeSetHead(int[] oldSetHead, int job, boolean addToHead) {
   if (addToHead) {
      int[] newSetHead = new int[numOfMachines];
      newSetHead[0] = oldSetHead[0] + processingTimes[job][0];
      for (int i = 1; i < numOfMachines; i++) {
         newSetHead[i] = Math.max(oldSetHead[i], newSetHead[i - 1]) + processingTimes[job][i];
      }

      return newSetHead;
   }

   return oldSetHead;
}
```

```
// calculation of setTail vector
private static int[] computeSetTail(int[] oldSetTail, int job, boolean addToHead) {
   if (!addToHead) {
      int[] newSetTail = new int[numOfMachines];
      newSetTail[numOfMachines - 1] = oldSetTail[numOfMachines - 1] +
          processingTimes[job][numOfMachines - 1];
      for (int i = numOfMachines - 2; i >= 0; i--) {
         newSetTail[i] = Math.max(oldSetTail[i], newSetTail[i + 1]) + processingTimes[job][i];
      }

      return newSetTail;
   }

   return oldSetTail;
}
```

```
// calculation of setProcessingTime vector
private static int[] computeSetProcessingTime(int[] oldSetProcessingTime, int job) {
   int[] newSetProcessingTime = new int[numOfMachines];
   for (int i = 0; i < numOfMachines; i++) {
      newSetProcessingTime[i] = oldSetProcessingTime[i] - processingTimes[job][i];
   }

   return newSetProcessingTime;
}
```

```
// calculation of lower bound is the largest sum of these three vectors for each machine
private static int computeLb(int[] setHead, int[] setTail, int[] setProcessingTime) {
   int lb = -1;
   for (int i = 0; i < numOfMachines; i++) {
      int sum = setHead[i] + setTail[i] + setProcessingTime[i];
      if (sum > lb) {
         lb = sum;
      }
   }

   return lb;
}
```

IV. Experiments

A. Benchmark Settings

I provided benchmark tests on my personal laptop with Intel Core i5 2,9 GHz processor, 16 GB 1867 MHz DDR3 memory and OS X El Capitan. All test instances are stored in *instances/* folder. During programing I used basic instance from Carlier and Rebaï [5]. It's stored in *papers/Carlier_Rebai.in* file. Second basic instance for debugging with 5 jobs and 4 machines I found on webpage http://bit.ly/1Wxwb7W. It's stored in *papers/Columbia_edu.in* file. A lot of instances with more jobs and machines I found at this webpage http://bit.ly/1qOvpHj in section *Flow shop sequencing*. Beside each input instance with suffix *.in* there is a file with the same name and suffix *.out* which contains best makespan for instance.

B. Results

Makespans of instances which I tested with the time of their execution is in the table below. (Execution times are in miliseconds)

Instance	Num of jobs	Num of machines	Execution time
papers/Carlier_Rebai	4	3	121
papers/Columbia_Edu	5	4	171
taillard/20_5/01	20	5	778
taillard/20_5/02	20	5	5 064
taillard/20_5/03	20	5	476
taillard/20_5/04	20	5	188
taillard/20_5/05	20	5	1 372
taillard/20_5/06	20	5	224
taillard/20_5/07	20	5	433
taillard/20_5/08	20	5	166
taillard/20_5/09	20	5	318
taillard/20_5/10	20	5	252
taillard/20_10/01	20	10	2 826
taillard/20_10/02	20	10	8 358
taillard/20_10/03	20	10	3 190
taillard/20_10/04	20	10	2 634
taillard/20_10/05	20	10	1 769
taillard/20_10/06	20	10	3 357
taillard/20_10/07	20	10	254 614
taillard/20_10/08	20	10	3 812
taillard/20_10/09	20	10	2 925
taillard/20_10/10	20	10	3 784
taillard/20_20/02	20	20	439 707 447
taillard/50_5/01	50	5	411
taillard/50_5/02	50	5	384 284
taillard/50_5/03	50	5	981
taillard/50_5/04	50	5	1 284
taillard/50_5/05	50	5	2 679
taillard/50_5/06	50	5	2 134
taillard/50_5/07	50	5	2 790
taillard/50_5/08	50	5	373
taillard/50_5/09	50	5	3 405
taillard/50_5/10	50	5	331
taillard/50_10/01	50	5	18 063
taillard/50_10/02	50	10	814 256
taillard/50_10/03	50	10	309 698
taillard/50_10/04	50	10	179 599
taillard/50_10/05	50	10	48 709
taillard/100_5/02	100	5	100 590
taillard/100_5/04	100	5	33 048
taillard/100_5/05	100	5	7 525

C. Discussion

I have done 42 tests with given instances. Some results are incomprehensible to me. For example my program returns for instance *05* from dataset *20_5* makespan 1235 but correct result by author of instances is 1236 so my result is about 1 less. The same phenomenon is observed in some other instances. For example instance *04* from dataset *20_10* (my result 1377; author result 1378), instance *02* from dataset *20_20* (my result 2099; author result 2100), instance *04* from dataset *50_10* (my result 3063; author result 3064).

There is next group of results, which are different from those reported by author (but defection is not equal 1). That are instances *01*, *02*, *03*, *05* from dataset *50_10*, instance *07* from dataset *20_5* and instance *05* from dataset *50_5*. In all of this instances makespan computed by me is a little bit lower than authors makespan.

It's interesting that instance *02* from dataset *50_5* takes much more time than other instances from this dataset. Instance *02* from dataset *20_20* takes extremely long time, 439 707 447 miliseconds is more than 5 days. I don't know as I could measure such a time (it could not cause an integer overflow).

I stopped computation of a lot instances because it takes a lot of time, therefore I don't mentioning here about them. Instance *01* from dataset *100_5* takes as many time as the exception *java.lang.OutOfMemoryError: GC overhead limit exceeded* was thrown.

All other instances (31 pieces) which I tested returned me correct makespan (the same as author write).

V. Conclusion

This seminar work was very interesting for me. I worked on it from a day one since I chose this assignment. I spent on implementation more than 150 hours of time. At the beginning it was nothing clear for me. Gradually I began to understand that more and more. The main cause of these long hours of work was that papers which I found were not written clearly. Pseudo codes was written in bullets, some facts missing and sometimes it's complicated to understand algorithms written in this form.

I tried to create enormous number of randomly generated instances with small number of jobs and machines in order to detect causes of errors in some big instances. But without success. All my randomly generated instances returned correct solutions. There is still a possibility that I found even better makespan than the author, but I don't really bealive it.

References

[1] A. H. G. Rinnooy Kan B. J. Lageweg, J. K. Lenstra. A general bounding scheme for the permutation flow-shop problem. *Operations Research*, 26(1):53–67, 1978.

[2] Egon Balas and Alkis Vazacopoulos. Guided local search with shifting bottleneck for job shop scheduling. *Manage. Sci.*, 44(2):262–275, February 1998.

[3] Christian Blum and Michael Sampels. An ant colony optimization algorithm for shop scheduling problems. *Journal of Mathematical Modelling and Algorithms*, 3(3):285–308.

[4] Peter Brucker, Bernd Jurisch, and Bernd Sievers. A branch and bound algorithm for the job-shop scheduling problem. *Discrete Applied Mathematics*, 49(1–3):107 – 127, 1994. Special Volume Viewpoints on Optimization.

[5] Jacques Carlier and Ismaïl Rebaï. Two branch and bound algorithms for the permutation flow shop problem. *European Journal of Operational Research*, 90(2):238 – 251, 1996.

[6] Chia-Shin Chung, James Flynn, and Omer Kirca. A branch and bound algorithm to minimize the total flow time for m-machine permutation flowshop problems. *International Journal of Production Economics*, 79(3):185 – 196, 2002.

[7] Chia-Shin Chung, James Flynn, and Ömer Kirca. A branch and bound algorithm to minimize the total tardiness for m-machine permutation flowshop problems. *European Journal of Operational Research*, 174(1):1 – 10, 2006.

[8] Christos Koulamas. A new constructive heuristic for the flowshop scheduling problem. *European Journal of Operational Research*, 105(1):66 – 71, 1998.

[9] Talel Ladhari and Mohamed Haouari. A computational study of the permutation flow shop problem based on a tight lower bound. *Computers & Operations Research*, 32(7):1831 – 1847, 2005.

[10] Claude Le Pape and Philippe Baptiste. Resource constraints for preemptive job-shop scheduling. *Constraints*, 3(4):263–287, October 1998.

[11] Rasaratnam Logendran, Paula deSzoeke, and Faith Barnard. Sequence-dependent group scheduling problems in flexible flow shops. *International Journal of Production Economics*, 102(1):66 – 86, 2006.

[12] T. Ladhari M. Haouari. A branch-and-bound-based local search method for the flow shop problem. *The Journal of the Operational Research Society*, 54(10):1076–1084, 2003.

[13] Andrew Manikas and Yih-Long Chang. Multi-criteria sequence-dependent job shop scheduling using genetic algorithms. *Computers & Industrial Engineering*, 56(1):179 – 185, 2009.

[14] Tadahiko Murata, Hisao Ishibuchi, and Hideo Tanaka. Genetic algorithms for flowshop scheduling problems. *Computers & Industrial Engineering*, 30(4):1061 – 1071, 1996.

[15] Y. Deepa N. Madhushini, C. Rajendran. Branch-and-bound algorithms for scheduling in permutation flowshops to minimize the sum of weighted flowtime/sum of weighted tardiness/sum of weighted flowtime and weighted tardiness/sum of weighted flowtime, weighted tardiness and weighted earliness of jobs. *The Journal of the Operational Research Society*, 60(7):991–1004, 2009.

[16] Muhammad Nawaz, E Emory Enscore, and Inyong Ham. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega*, 11(1):91 – 95, 1983.

[17] Eugeniusz Nowicki and Czeslaw Smutnicki. A fast taboo search algorithm for the job shop problem. *Manage. Sci.*, 42(6):797–813, June 1996.

[18] Shusuke Okamoto, Ichie Watanabe, and Hajime Iizuka. A new optimal algorithm for the permutation flow-shop problem and its parallel implementation. *Computers & Industrial Engineering*, 27(1–4):39 – 42, 1994. 16th Annual Conference on Computers and Industrial Engineering.

[19] E. Pinson. A practical use of jackson's preemptive schedule for solving the job shop problem. *Ann. Oper. Res.*, 26(1-4):269–287, January 1991.

[20] C.N. Potts. An adaptive branching rule for the permutation flow-shop problem. *European Journal of Operational Research*, 5(1):19 – 25, 1980.

[21] Rubén Ruiz and Concepción Maroto. A comprehensive review and evaluation of permutation flowshop heuristics. *European Journal of Operational Research*, 165(2):479 – 494, 2005. Project Management and Scheduling.

[22] Marino Widmer and Alain Hertz. A new heuristic method for the flow shop sequencing problem. *European Journal of Operational Research*, 41(2):186 – 193, 1989.