Identifying and exploiting commonalities for the job-shop scheduling problem

Marnix Kammer Marjan van den Akker Han Hoogeveen

[a] *Utrecht University, P.O.Box 80089 3508 TB Utrecht*

Abstract

For many combinatorial problems the solution landscape is such that near-optimal solutions share common characteristics: the so-called commonalities or building blocks. We propose a method to identify and exploit these commonalities, which is based on applying multistart local search. In the first phase, we apply the local search heuristic, which is based on simulated annealing, to perform a set of independent runs. We discard the solutions of poor quality and compare the remaining ones to identify commonalities. In the second phase, we apply another series of independent runs in which we exploit the commonalities. We have tested this generic methodology on the so-called job-shop scheduling problem, on which many local search methods have been tested. In our computational study we found that the inclusion of commonalities in simulated annealing improves the solution quality considerably even though we found evidence that the job-shop scheduling problem is not very well suited to the use of these commonalities. Since the use of commonalities is easy to implement, it may be very useful as a standard addition to local search techniques in a general sense.

Keywords. Local search; commonalities; building blocks; job shop scheduling; simulated annealing; multistart.

1 Introduction

When solving a combinatorial optimization problem by a multi-start local search, many alternative solutions are generated. The idea is that an element of the solution that occurs in many high quality solutions must be a good element. Such elements are called *commonalities* and can be used to guide the search process. As far as we know, the name commonality originates from the work by Schilham ([8]), who investigated local search methods for combinatorial optimization problems, like the job-shop problem and the traveling salesman problem. Based on his experiments, he formulated the following two hypotheses:

1. Good solutions have many building elements (which he called commonalities) in common.
2. The number of commonalities increases with the quality of the solution.

These observations led him to the following idea: when you get stuck in a run of a local search algorithm, do not apply a random restart, but use information from the solutions obtained so far. He implemented it by applying random perturbations to the current solution, where the probability of perturbing a building element depends on the number of times that it occurs in a reference pool containing 'good' solutions found earlier in the run.

Commonalities show strong resemblance to the so-called *building blocks*, which are widely believed to determine the success of genetic algorithms. The idea is that solutions sharing these parts will become dominant in the pool of solutions, which makes it very likely that they will be part of the final solution.

We have looked at the possibility of applying commonalities to find a good solution of the job-shop problem (see Section 2 for a description), just like Schilham did. In contrast to Schilham, we explicitly determine beforehand the commonalities occurring in an instance by running a first series of independent runs of a local search algorithm. After having determined the commonalities, we apply a second series of independent runs in which we add a bonus in the solution value for each commonality occurring in the solution; these bonus values gradually decrease during the run of simulated annealings. With some imagination, this approach can be viewed upon as the application of a genetic algorithm without having to bother about how to code a solution and how to define the cross-over operator and the selection mechanism. We have applied our algorithm to a number of benchmark instances.

The outline of the paper is as follows. In Section 2 we describe the job-shop scheduling problem, which we use to test the merits of our approach. In Section 3 we describe the disjunctive graph model. In Section 4 we present our initial simulated annealing algorithm, the derivation of the commonalities, and the incorporation of the commonalities in the simulated annealing algorithm. In Section 5 we present our computational results, and we compare our approach to designing a genetic algorithm. Finally, in Section 6 we draw some conclusions.

2 The job-shop scheduling problem

In a *job-shop scheduling problem* (JSSP) we have m machines, which have to carry out n jobs. In our variant of the JSSP, we assume that each job has to visit each machine exactly once; hence, each job consists of m operations, which have to be executed in a fixed order. For each operation we are given the machine by which it must be carried out without interruption and the time this takes, which is called the processing time. Each operation can only start when its job predecessor (the previous operation in its job) has been completed. All machines are assumed to be continuously available from time zero onwards, and each machine may only carry out one operation at a time. There is no time needed to switch from carrying out one job to another. Waiting between two operations of the same job is allowed, just like waiting between two operations on the same machine. The problem is to find a feasible *schedule*, which is fully determined by the completion time of each operation; the completion times can easily be computed when the order in which the operations are executed is known for each machine, since it is never advantageous to leave the machine idle if there exists an operation to start. The goal is to minimize the time by which the last machine (or job) finishes; this is also called the makespan or the length of the schedule.

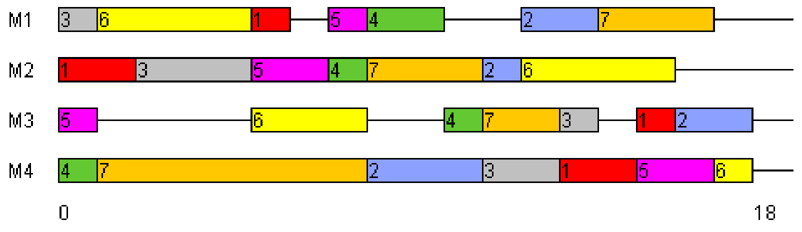

Figure 1: Example with optimal solution for a JSSP instance with 4 machines and 7 jobs.

There exist many practical problems that boil down to a job shop scheduling problem (see for example [9]). Unfortunately, this problem is known to be $\mathcal{NP}$-hard in the strong sense, even if each job visits each machine in the same order (the so-called flow-shop problem). Moreover, Williamson et al. ([12]) have shown that already the problem of deciding whether there exists a feasible schedule of length 4 is $\mathcal{NP}$-hard in the strong sense, which implies that no polynomial algorithm can exist with worst-case bound less than 5/4, unless $\mathcal{P} = \mathcal{NP}$. Furthermore, these problems are also very hard to solve in practice; instances with more than 20 jobs usually are computationally intractable. Therefore, many researchers have studied local search methods, like for example tabu search based algorithms ([10], [6]), simulated annealing based algorithms ([11]) and, more recently, hybrid genetic algorithms ([2], [5]); all of these studies report that good results are obtained. We will use simulated annealing as our basic local search algorithm, in which we incorporate the use of commonalities.

3 The disjunctive graph model

It has become standard now to model a job shop scheduling algorithm using a disjunctive graph, as was introduced by Roy and Sussman ([7]). This graph is constructed as follows. The vertices V of the disjunctive graph represent the operations; vertex v_i, corresponding to operation i, gets weight equal to its processing time p_i. Furthermore, there are two dummy vertices v_{start} and v_{end}. We draw an arc (v_i, v_j) between vertices v_i and v_j if the operation j is the *direct* successor of operation i in some job. Furthermore, we include an edge between each pair of vertices that correspond to two operations that must be executed by the same machine and that do not belong to the same job. All arcs and edges get weight zero. Finally, we add arcs from v_{start} to the first operation of each job and arcs from the last operation of each job to v_{end}. In the example Figure 2 each job has a separate color and the edges are depicted by dotted lines.

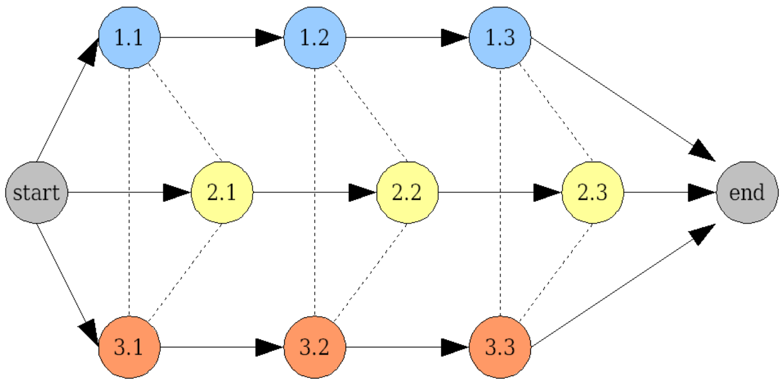

Figure 2: A disjunctive graph representing a JSSP instance.

Since a schedule is fully specified when the order of the operations on the machines is given, we have to direct the edges such that an acyclic graph remains. See Figure 3 for an example. After the edges have been oriented, we call them *machine arcs*; to distinguish these from the original arcs in the graph, the latter ones are called *job arcs*.

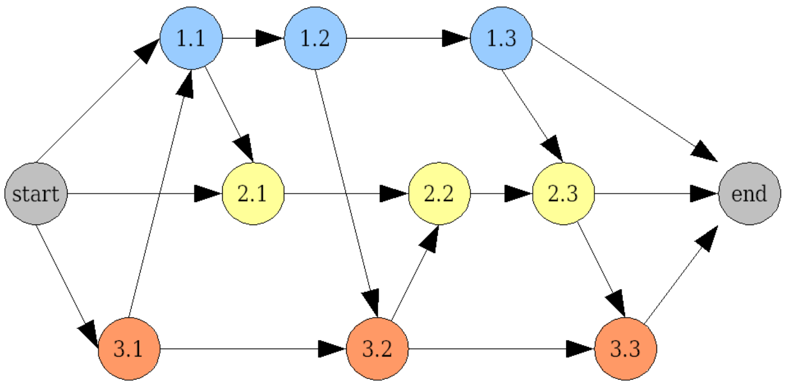

Figure 3: Directed acyclic graph representing a solution for a JSSP instance.

Given the directed graph, we can compute the starting time of each operation as the length of the longest path in the graph from v_{start} to the vertex corresponding to this operation. Hence, the makespan is equal to the length of the longest path to v_{end}. Adams et al. ([1]) have shown that the calculation of the longest path on a directed acyclic graph can be done in linear time. A longest path in the directed acyclic graph is also called a *critical path*; the critical path does not have to be unique. We can decompose a critical path into *critical blocks*, where each critical block consists of one or more operations that are carried out contiguously on the same machine; at the end of a critical block, the critical path jumps to another machine.

Although the exact details of the implementation of our algorithm are for the most part irrelevant in this paper, it is important to note that objects representing graph nodes (operations) have at most four explicit edges in our implementation. Two job arcs are represented by references to the *job predecessor* and *job successor*, and two machine arcs are represented by references to the *machine predecessor* and *machine successor*. Job arcs between two operations that are not directly consecutive are not needed by the algorithm and left out of the model, and the remaining machine arcs between all operations on a machine are only implicitly defined by keeping a list of operations on each machine in order to save memory (and possibly to increase speed).

4 A simulated annealing based algorithm

We need an initial solution to get the local search algorithm going. There are many methods for generating a good starting solution for the JSSP like the Shifting Bottleneck procedure (see [1]). We decided to start with a random initial solution, since we observed in our experiments that our algorithm always moved to a good schedule quickly. The same behavior was shown in the second phase of the algorithm in which we used the commonalities.

In our simulated annealing algorithm we use the standard neighborhood of reversing machine arcs on the longest path. Van Laarhoven et al. ([4]) have shown that this will lead to a feasible schedule. Moreover, Nowicki and Smutnicki ([6]) have shown that we can restrict ourselves to moves in which we reverse the execution order of either the last two operations in the first critical block, the first two operations in the last critical block, or the first or last two operations in any intermediate critical block. In an iteration, we choose one of these at random. Consequently, we need to update the references to the machine predecessor and successor of two operations.

Determining critical paths and makespan
There are a few calculations which we have to do frequently on the solution graph, like calculating the makespan, which is equal to the start time of the dummy node v_{end}. Calculating the start times of the operations can be done by first obtaining a topological sorting of all graph nodes with a simple depth-first search. Then, we iterate over the topological sorting and determine the start time S_i of each operation i as

$$\max_{h \in P_i} S_h + p_h;$$

here P_i is the list of i's predecessors (both on the machine and in the job) and p_h is the processing time of operation h; we initialize by putting the start time of the dummy node v_{start} equal to 0. This procedure for calculating all start times runs in linear time. Note that when two operations are swapped only the start times of all operations in the subgraph beginning at the swapped operations are changed, so the calculation of start times with a topological sorting can be done with the first of the swapped operations as start node, saving some calculation time.

In order to identify possible operation swaps we need to calculate all critical paths. This can also be done with a topological sorting: for each node i in the topological sorting, a *critical predecessor* j can be determined as

$$j \leftarrow \arg\max_{h \in P_i} S_h + p_h;$$

because of the computation of S_i, we have that j is simply a predecessor for which $S_j + p_j = S_i$. Note that j does not have to be unique. When all critical predecessors are known, the critical paths can be obtained by walking back from the end dummy node to the start dummy node following only arcs that connect an operation to its critical predecessor, which can be implemented to run in linear time.

Exploiting commonalities in solutions
In the first phase of our algorithm, the regular simulated annealing method described in the previous sections is run for a number of times we consider large enough to gather useful information with. During these runs, datastructures counting the number of occurrences of machine arcs are continuously updated. After all the regular simulated annealing runs have been completed, commonalities among the best solutions found in each run are identified using these datastructures. We distinguish four types of commonalities; we discuss the thresholds later.

1. *all-pairs commonalities* (APC). An APC corresponds to an ordered pair of operations i and j that are executed in this order on the same machine with or without idle time and/or other operations in between. Note that such a pair corresponds to a machine arc in the disjunctive graph.

2. *start/end commonalities* (SEC): An SEC corresponds to a single operation that occurs either first or last on a machine.

3. *direct-pairs commonalities* (DPC): A DPC corresponds to an ordered pair of operations that are executed contiguously on the same machine.

4. *critical-pairs commonalities* (CPC): A CPC corresponds to an ordered pair of contiguously executed operations on the same machine that are part of a critical path.

For each commonality type we require that a commonality occurs in at least a certain threshold percentage of the best solutions found in the first phase in order to be identified as a piece of useful information.

After all these commonalities have been identified, the second phase is started. The simulated annealing algorithm that we apply is essentially the same one as in the first phase but with penalties given out to solutions that violate the commonalities. The objective function in the first phase was to minimize the makespan; in the second phase the objective is to minimize the sum of the makespan and the total penalty in order to generate good solutions that make the best use of the commonalities. The four commonality types differ greatly in both significance and occurrence frequency and therefore the penalties also differ. Just like the temperature in the simulated algorithm, the values of the penalties are lowered over time. The idea behind this is that we first steer the algorithm in a good direction quickly, after which it tries to preserve commonalities while exploring new and possibly better areas of the search space.

5 Experiments and computational results

5.1 Parameter tuning

Even though we are mainly concerned with the measure of improvement due to adding the commonalities, we want to tune the parameters such that the simulated annealing algorithm finds reasonably good solutions in the first phase; after all, according to the hypothesis by Schilham, the better the solutions, the more commonalities they share. To make the algorithm run, we must specify a starting temperature, a cooling off speed, and an end temperature at which point the algorithm terminates. The simulated annealing settings which seemed to work best on most problems instances and which we used throughout the rest of our experiments are as follows:

- The initial acceptance threshold $t = 0.5$ (the probability that a move to a worse solution is accepted)
- After every 500 iterations, the acceptance threshold is multiplied by $\alpha = 0.95$
- The algorithm terminates at $t = 0.01$ (this works out to 38,500 iterations per run)

Next, we needed to determine the thresholds for accepting commonalities and the penalties for violating them. During many experiments with running the regular simulated annealing algorithm of the first phase on various problem instances of m machines and n jobs, we observed the following frequencies with which the different types of commonalities occurred in the best solutions:

1. *all-pairs commonalities* (APC): These occur a lot, usually around $n \cdot m$ times even with a high acceptance threshold (> 0.9).

2. *start/end commonalities* (SEC): Usually around m times with a low acceptance threshold (< 0.7).

3. *direct-pairs commonalities* (DPC): Usually less than m times even with a low acceptance threshold (< 0.7).

4. *critical-pairs commonalities* (CPC): Most uncommon, these do not occur at all on larger problem instances.

From these observations, type 4 did not turn out to be useful and was not used in any further research. It is also clear that type 1 occurs a lot and should have a high acceptance threshold and low penalties, and type 2 and 3 should have a low acceptance threshold and higher penalties than type 1. During further experiments with the height of penalties, we observed that the settings from Table 1 below produced the best results on most of the problem instances we used for testing (ranging from size 5×10 to 20×20), so we use these values in the rest of our experiments.

Commonalities Type	Acceptance Threshold	Penalty
APC	0.95	0.1
SEC	0.60	1.0
DPC	0.60	1.0

Table 1: Settings for our algorithm involving commonalities

5.2 Survival of commonalities

During all experiments, it became clear that commonalities did not appear as frequently as we expected beforehand. A possible explanation for this is that we observed that many good schedules for the same problem instance are very different from one another, making it hard to identify commonalities. For the problem instance abz9, we identified four good but very different solutions: not only the order of the operations on the machines is very different, but the critical paths differ greatly in every schedule as well (see [3]).

When comparing the first and second phase, we see that in our experiments about 80 % of the commonalities identified in the first phase is actually included in final solutions of the second phase. However, for the APC commonality this percentage is about 95. When comparing results from different runs of the second phase, our experiments suggest that in most cases the best solution contains the most commonalities. However, somewhat peculiar is that the best and worst solutions seem to contain slightly more commonalities than solutions with an average value, i.e. commonalities do either right or wrong.

5.3 Other, less satisfactory experiments

During our research project, there were a few ideas we tried to apply to our algorithm but we discarded them for various reasons. These include:

- Fixing certain commonalities in solutions in the second phase, as opposed to giving penalties for violating them. This narrowed the search space down to the point where the algorithm never reached any good solution at all.
- Trying to determine a *bottleneck* machine and then focusing the algorithm on fixing common machine arcs on that machine. This also narrowed down the search space too much. Furthermore, many problem instances turn out not to have a single machine that is the bottleneck so the algorithm focuses on the wrong information.
- Making the penalties for violating commonalities dependent on the occurrence of those commonalities: the more often a commonality occurs, the higher the penalty. This had no noticeable effect on the quality of the found schedules.
- Executing more than two phases, identifying commonalities again after each phase. The idea behind this was that the identified commonalities would be increasingly useful after each phase so that the quality of found schedules would improve. However, no improvement in quality was found in any third phase or later.
- Using bigger structures as commonalities than just pairs of operations. No such structures could be identified that occurred in schedules often enough to be useful.

5.4 Final experiment results

The final experiment we carried out involved running the algorithm with and without commonalities on a series of benchmark problem instances used throughout the literature, retrieved mainly from the OR-library. On each problem instance, we ran the regular simulated annealing algorithm without commonalities (SA) for 100 runs, and then ran the two-phase algorithm that exploits commonalities (SA+C) with 50 runs in both phases, in order to make a fair comparison. In all experiments, we used the simulated annealing parameters and settings involving commonalities established in the previous sections. The computational results are listed below. For a wider comparison with current technology, the results for the recently developed advanced hybrid genetic algorithm GTS (see [5]) are also included in the table. The 'time' column, included to be able to make a comparison in speed as well, indicates the average CPU time in seconds needed to execute one run of the algorithm. Our algorithm was run on a 2GHz processor, the GTS algorithm was run on a Sun Sparc station.

Problem			SA			SA+C			GTS	
instance	$n \times m$	opt	avg	best	time	avg	best	time	best	time
la02	10×5	655	663	655	14	661	655	15	655	1
la19	10×10	842	848	842	37	845	842	39	842	4
ft10	10×10	930	961	934	38	958	930	39	930	7
orb1	10×10	1059	1097	1066	41	1090	1059	44	-	-
la21	15×10	1046	1070	1055	51	1063	1054	56	1047	12
la27	20×10	1235	1280	1250	58	1275	1239	60	1235	26
la40	15×15	1222	1254	1232	69	1252	1229	75	1226	19
abz7	20×15	655	687	672	86	684	669	91	658	176
abz9	20×15	656	719	699	105	713	694	115	682	125
yn1	20×20	846	921	900	174	916	892	193	-	-

Table 2: Computational results of regular simulated annealing (SA), our algorithm with commonalities (SA+C), and a recent advanced hybrid genetic algorithm (GTS).

A few obvious remarks can be made after reviewing the computational results above:

- SA+C performs better than SA on all problem instances (both with average makespan and best makespan).
- SA+C finds an optimum on all instances of 10×10 or smaller.
- SA is always a little bit faster than SA+C.
- GTS gives the best results of all three algorithms.
- GTS is faster on all smaller problem instances, but the running time increases dramatically on larger instances.

5.5 Comparison to Genetic Algorithms

As observed in the introduction, commonalities can be viewed upon as building blocks, which are used in genetic algorithms. In the computational experiments our approach is outperformed by the GTS algorithm. Still, we are convinced that our approach is a valuable addition to the area of metaheuristics. Below, we will compare it to a GA and state the pros and cons.

- Cross-over. Defining a good cross-over operator in a GA that leaves the building blocks intact is difficult in many applications. In our approach we do not need to bother about it.
- Mutation. In a GA you need to define a mutation operator to prevent premature convergence. Mutation in our approach is automatically taken care of by the moves in the neighborhood; to avoid premature convergence, we must set the penalties for violating commonalities at a modest value (the poor solutions that occurred when we fixed some of the commonalities were due to premature convergence).
- Parameter tuning. Implementing a GA is easy, but making it perform well is a different story: a GA is heavily parametrized, and it requires a lot of parameter tuning to get the best out of it. Our approach, on the other hand, is fairly simple to implement: all it requires is a straightforward implementation of SA (or for example Tabu Search) and a mechanism to detect commonalities; all fancy improvements turned out to be a waste of time.

6 Conclusions

From the computational results in the previous section it can be concluded that exploiting commonalities in solutions for the JSSP improves the quality of schedules for both the average makespan and the makespan of the best solution. The improvement is less than a percent of the value of the optimum, but it closes a reasonable part of the gap to the optimum. We find that the only types of commonalities that are somewhat useful are *all-pairs commonalities*, *start/end commonalities* and *direct-pairs commonalities*. An explanation for the improvement being small is that in the JSSP, good solutions have so many differences (see [3]) that it is hard to identify commonalities between them. Therefore, using commonalities is not well suited to the JSSP and our algorithm does not perform as well as current, advanced algorithms specialized for the JSSP. We believe that the Traveling Salesman problem might be a better candidate, but this has to be tested.

However, we have shown that the technique is easy to implement in addition to a more standard local search algorithm. Whereas a highly specialized algorithm such as the one by Moraglio et al. ([5]) is well suited to only the JSSP, the use of commonalities is a more general concept and applicable to many problems. It is undoubtedly an improving addition to local search based algorithms for any problem in which some type of commonality can be detected. Moreover, we can apply the concept of a commonality to reduce heuristically the size of the instance: if a commonality occurs in (almost) all good solutions, then we may fix it; this clearly does not apply to the job shop problem.

We are currently working on an experiment in which we compare the run of a genetic algorithm to a simulated annealing algorithm with commonalities.

References

[1] J. Adams, E. Balas, and D. Zawack (1988). The shifting bottleneck procedure for job shop scheduling. *Management Science* 34, 391-401.

[2] J.F. Gonçalves, J.J. de Magalhães Mendes, and M.G.C. Resende (2005). A hybrid genetic algorithm for the job shop scheduling problem. *European Journal of Operational Research* 167, 77-95.

[3] M.L. Kammer, J.M. van den Akker, and J.A. Hoogeveen (2010). *Identifying and exploiting commonalities for the job-shop scheduling problem*, Technical Report UU-CS-2010-002, University of Utrecht, The Netherlands. http://www.cs.uu.nl/research/techreps/repo/CS-2010/2010-002.pdf

[4] P.J.M. van Laarhoven, E.H.L. Aarts, and J.K. Lenstra (1992). Job shop scheduling by simulated annealing. *Operations Research* 40, 113-125.

[5] A. Moraglio, H. ten Eikelder, and R. Tadei (2005). *Genetic Local Search for Job Shop Scheduling Problem*, Technical Report CSM-435, University of Essex, UK.

[6] E. Nowicki and C. Smutnicki (1996). A fast taboo search algorithm for the job shop problem. *Management Science* 42, 797-813.

[7] B. Roy and B. Sussmann (1964). Les Problèmes d'ordonnancement avec contraintes disjonctives, Note DS no. 9 bis, SEMA, Montrouge.

[8] R.M.F. Schilham (2001). *Commonalities in local search.* PhD Thesis, Eindhoven University of Technology, The Netherlands.

[9] K. Schmidt (2001). *Using tabu search to solve the job shop scheduling problem with sequence dependent setup times.* Master's thesis, Brown University, USA.

[10] E.D. Taillard (1994). Parallel taboo search techniques for the job shop scheduling problem. *ORSA Journal on Computing* 6, 108-117.

[11] T. Yamada and R. Nakano (1996). Job-shop scheduling by simulated annealing combined with deterministic local search. I.H. Osman and J.P. Kelly (Eds.). *Meta-heuristics: theory and applications.* Kluwer academic publishers MA, USA, pp. 237-248.

[12] D.P. Williamson, L.A. Hall, J.A. Hoogeveen, C.A.J. Hurkens, J.K. Lenstra, S.V. Sevast'janov, and D.B. Shmoys (1997). Short shop schedules. *Operations Research* 45, 288-294.