Computers & Operations Research 38 (2011) 1556–1561

Contents lists available at ScienceDirect

Computers & Operations Research

journal homepage: www.elsevier.com/locate/caor

Identifying and exploiting commonalities for the job-shop scheduling problem

Marnix Kammer, Marjan van den Akker, Han Hoogeveen *

Department for Information and Computing Sciences, Utrecht University, P.O. Box 80089, 3508 TB Utrecht, The Netherlands

ARTICLE INFO

Available online 19 February 2011

Keywords:
Local search
Commonalities
Building blocks
Job-shop scheduling
Simulated annealing
Multistart

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.

1. Introduction

In this paper, we present a simple way to improve a local search algorithm like simulated annealing by identifying and exploiting *commonalities*. The basic idea is that we determine and compare a large number of good solutions: an element of the solution that occurs in so many high quality solutions is most likely to be a good element. We then run our local search algorithm again, where we favor solutions that contain the commonalities that we have discovered before.

As far as we know, the name commonality originates from the work by Schilham [1], 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 the commonalities 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 favor the occurrence of the commonalities. This resembles the working of a genetic algorithm, which combines building blocks to get a good solution. The nice thing about our procedure is that we run some kind of 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 tested our algorithm on 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

* Corresponding author. Tel.: +31 30 253 4089; fax: +31 30 251 3791.
E-mail addresses: mlkammer@cs.uu.nl (M. Kammer), marjan@cs.uu.nl (M. van den Akker), slam@cs.uu.nl (H. Hoogeveen).

0305-0548/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cor.2011.01.014

model of the job-shop scheduling problem, which we need for our local search algorithm. 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 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 (Fig. 1).

There exist many practical problems that boil down to a job-shop scheduling problem (see for example [2]). 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. [3] 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; hence, since all outcome values are integral, this result implies that there cannot exist a polynomial algorithm with worst-case bound less than 5/4, unless $\mathcal{P}=\mathcal{NP}$, as this algorithm must find a solution with outcome value smaller than 5 (and hence with value 4), if and only if a solution with outcome value 4 exists, thus deciding in polynomial time this strongly $\mathcal{NP}$-hard problem. 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 [4,5], simulated annealing based algorithms [6] and, more recently, hybrid genetic algorithms [7,8]; all of these studies report that good results are obtained. Because of the simplicity of implementation, we 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 an instance of a job-shop scheduling problem using a disjunctive graph, as was introduced by Roy and Sussman [9]. 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 Fig. 2 each job j ($j=1,2,3$) consists of three operations, which are depicted as vertices ($j.i$) ($i=1, 2, 3$); all operations belonging to the same job are on the same horizontal height and share the same color. Finally, the edges are depicted by dotted lines.

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 Fig. 3 for an example.

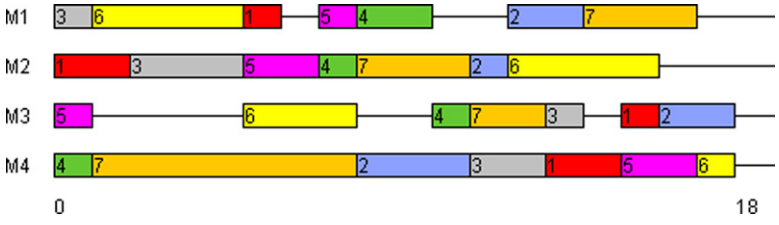

Fig. 1. Example with optimal solution for a JSSP instance with four machines and seven jobs.

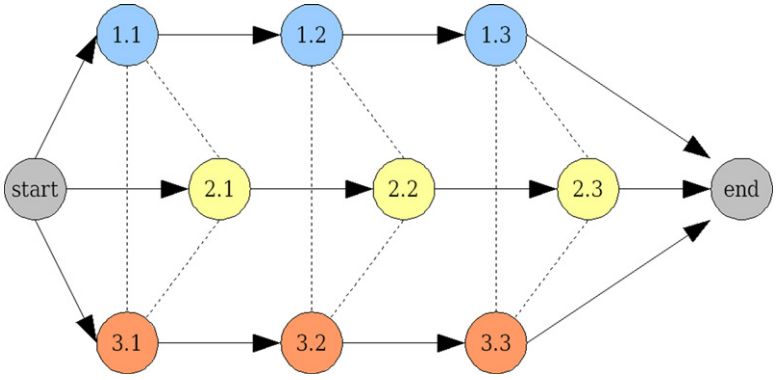

Fig. 2. A disjunctive graph representing a JSSP instance.