UNIVERSIDAD POLITÉCNICA DE VALENCIA
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

DESIGN AND DEVELOPMENT OF CSP TECHNIQUES FOR FINDING ROBUST SOLUTIONS IN JOB-SHOP SCHEDULING PROBLEMS WITH OPERATORS

Master Thesis in
Artificial Intelligence, Pattern Recognition
and Digital Imaging

September 2012

Supervised by:
Dr. Miguel Á. Salido Gregorio
Dr. Federico Barber Sanchís

Presented by:
Joan Escamilla Fuster

Design and Development of CSP Techniques for Finding Robust Solutions in Job-Shop Scheduling Problems with Operators

Joan Escamilla Fuster

Master Thesis in
Artificial Intelligence, Pattern Recognition
and Digital Imaging

Departamento de Sistemas Informáticos y Computación

Valencia, 2012

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Introduction

Nowadays, the main objective of many companies and organizations is to improve profitability and competitiveness. These improvements can be obtained with a good optimization of resources allocation. The job-shop scheduling problem (JSP) is a possible representation of a typical problem of scheduling. Many real life problems can be modelled as a job-shop scheduling problem and can applied in some variety of areas, such as process scheduling in an industry, departure and arrival times of trains at stations, the delivery times of orders in a company, etc. To solve this problem many techniques have been developed such as branch and bound, constraint satisfaction techniques, neural networks, genetic algorithms or tabu search.

In general a scheduling problem is a combinatorial optimization problem. The aim of scheduling is the resource allocation of tasks when one or more objectives must be optimized. The resources can be workshop machines, work tools, working staffs, etc. Tasks can represent operations of a production process, executions of a computational program, steps of a process, arrivals or departures of a train, etc. The objective might be to minimize the time execution, to minimize the number of tasks after a given date, etc.

The job-shop scheduling problem with operators is an extension of the classical job-shop scheduling problem where each operation has to be assisted by one operator from a limited set of them. The job-shop scheduling problem with operators has been recently proposed by Agnetis et al. [1]. This problem is denoted as $JSO(n, p)$ where n is the number of jobs and p denotes the number of operators. It is motivated by manufacturing processes in which part of the work is done by human operators sharing the same set of tools. The problem is formalized as a classical job-shop scheduling problem in which the processing of a task on a given machine requires the assistance of one of p available operators. The state of the art of $JSO(n, p)$ is analyzed in Chapter 3. This problem have modelled and solved by [1] with a branch and bound technique were some heuristics algorithms are also

used. Another technique to solve the problem with operators have been developed by [29] using genetic algorithm. In [28] the problem is solved combining global pruning rules with depth-first search.

In Chapter 2 the state of the art of constraint satisfaction problem (CSP) is presented. A CSP is represented as a finite set of variables, a domain of values for each variable and a set of constraints that bound the combination of values that variables can simultaneously take. The goal of CSP is to select a value for each variable so as to satisfy all constraints of the problem.

$JSO(n, p)$ has been modelled as a CSP in Chapter 4 and a branch and bound (B&B) algorithm with techniques of backtracking has been implemented to solve it. The cost to solve a CSP is dependent of the domain and the number of variables so when any of these factors is high the problem is considered NP complexity. If the aim is to get the optimal solution or solutions optimized (CSOP) [4], a lot of solutions have to be found in an iterative process and this search is really hard. For this reason have been applied some techniques to try to simply the domain and has been modified the original algorithm B&B to be more specific to the problem. But also with these improvements the problems remains intractable. The algorithm need too much time to found an optimal solution minimizing the makespan. Makespan is the maximum end time of the operations so if makespan is minimized the end time is reduced also. The reason to found the optimal solution is because through this solution or modifying it can be created a solution that also optimize the makespan and the robustness [41].

A solution is robust if is it able to maintain its functionality under a set of incidences. The robustness in JSP can be obtained trough allocation buffer times in order to absorb incidences. Another technique with three steps has been developed, it is showed in Chapter 5, the JSP is solved without taking into account operators and thus simplifying the problem is named first step. In the second step with a post-procedure the solution is adapted to the operators constrains building a correct solution to $JSO(n, p)$, this adaptation is taking account a trade off between optimization and robustness. In the third step, if it is possible, the number of buffers is increased without losing optimality redistributing the existent buffers.

1.2 Motivation

The job-shop scheduling problem is a well-known general problem of scheduling. This is not a real problem but in many cases can be related to some real problems. JSP represents a problem where there are some specific resources which have to be used to carry out some tasks. The extension with operators ($JSO(n, p)$) can represent more real life problems, because it provides the problem to use a new resource (operators) and those resources are not tied to a specific task.

In scheduling, usually, the most important thing is to finish the task in the shortest possible time. However, in some cases robustness can be an important point to keep in mind. For this reason, it appears the idea to solve this scheduling

problem with the aim to get a solution that minimizes the ending time; and at the same time this solution should be able to absorb incidences in the schedule without modifying any task. This idea opens the motivation to found techniques that can be used to get robust solutions that try to absorb incidences that can appear due to machine failures, late deliveries, human errors, etc.

Chapter 2

Constraint Satisfaction Problems

2.1 Introduction

Constraint Programming is a software technology used to represent and solve large and complex problems from many real life areas. Many of these problems can be modeled as constraint satisfaction problems and solved using constraint programming techniques. Examples include scheduling, planning, temporal reasoning, design engineering, packing problems, cryptography, diagnosis, etc. The complexity of this type of problem is NP [27].

A CSP is composed with a set of variables where each one can take values of a specific domain. There is a set of constraints that limit the number of values that each variable can take. The objective is to select a value to each variable satisfying the set of constraints.

The resolution of a CSP has two phases: The first one is to model the problem using the correct syntax and the second part is to solve the problem using one of the different techniques.

2.2 Modelling a CSP

A constraint satisfaction problem (CSP) is a triple (X, D, C) where:

- $\mathcal{X} = \{x_1, x_2, ..., x_n\}$ is a set of n variables.
- $\mathcal{D} = \{d_1, d_2, ..., d_n\}$ is a set of domains, such that each variable $x_i \in X$ has a finite set of possible values d_i.
- $\mathcal{C} = \{c_1, c_2, ..., c_m\}$ is a finite set of constraints which restrict the values that the variables can simultaneously take.

A constraint can be defined in intensional or extensional form although both can be equivalent.

- Intensional constraints: Constraints are represented as mathematic or logical function.

  Example. $x_1 \leq 4$ is a unary intensional constraint.

- Extensional constraint: Constraints are represented as a set of valid or invalid tuples.

  Example. The set of tuples $\{(0), (1), (2), (3), (4)\}$ is the extensional representation of the constraint $x_1 \leq 4$ by means of valid tuples, considering the domain $d_1 : \{0..10\}$ for x_1.

2.3 Example of a CSP: Map Colouring Problem

The Map Colouring problem tries to colour the areas in map using a number of colours but with the condition that the neighbouring areas have different colours. The map colouring problem can be represented as a graph colouring problem to color the vertices of a given graph using predefined number of colours in such a way that connected vertices get different colours. It is very easy to model this problem as a CSP. There are as many variables as vertices, and the domain for each variable contains the colours to be used. If there is an edge between the vertices represented by variables x and y, then there is an inequality constraint referring to these two variables, namely: $x \neq y$.

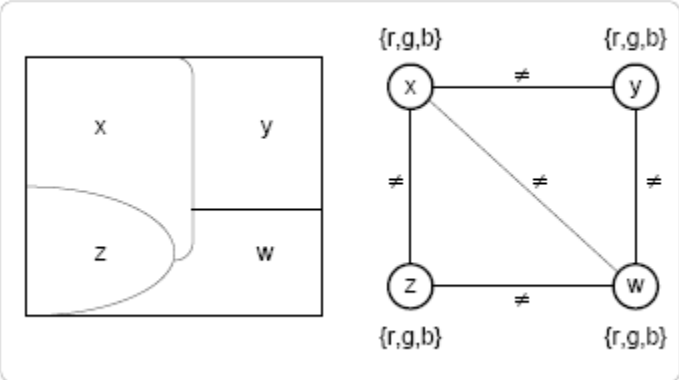

Figure 2.1: An example colouring problem

Graph colouring is known to be NP-complete, so one does not expect a polynomial-time algorithm to be found for solving this problem. It is easy to generate a large number of test graphs with certain parameters, which are more or less difficult to be coloured, so the family of graph colouring problems is appropriate to test algorithms thoroughly. Furthermore, many practical problems, like ones from the field of scheduling and planning, can be expressed as an appropriate graph colouring problem [37].

Figure 2.1 shows an example of map colouring problem and its graph representation. The map is composed on four regions/variables x, y, z, w to be coloured. Each region can be coloured in three different colours: red (r), green (g) or blue (b). So the domain of each variable is r,g,b. Each edge represents the binary constraint which restricts that two adjacent regions must be coloured with different colours. There are five constraints because there are five edges. A possible solution is the assignation (x=r), (y=g), (z=g) and (w=b).

2.4 Solving a CSP

A CSP can be solved by assigning values to each variable, the solution space can be seen as a search tree. In each level, a variable is instantiated and the successors of a node are the variable values of this level. Most algorithms for solving CSPs search systematically through the possible assignments of values to variables. Such algorithms are guaranteed to find a solution or to prove that the problem is insoluble. Prune is only used if the partial search tree contains not solution. The disadvantage of these algorithms is that they may take a very long time to find a solution.

2.4.1 Consistency Techniques

Consistency techniques were introduced for improving the efficiency of search techniques. The number of possible combinations can be huge, while only very few are consistent. By eliminating redundant values from the problem definition, the size of the solution space decreases. Reduction of the problem can be done once, as a pre-processing step for another algorithm, or step by step, interwoven with the exploration of the solution space by a search algorithm. Local inconsistencies are single values or combination of values for variables that cannot participate in any solution because they do not satisfy some consistency property [27].

For instance, if a value a of variable x_i is not compatible with all the values in a variable x_j that is constrained with x_i, then a is inconsistent and this value can be removed from the domain of the variable x_i. In the following paragraphs we introduce the most well-known and widely used algorithms for binary CSPs.

- A CSP is node-consistent if all the unary constraints hold for all the elements of the domains. The straightforward node-consistency algorithm (NC), which removes the redundant elements by checking the domains one after the other, has $O(dn)$ time complexity, where d is the maximum size of the domains. Thus, enforcing this consistency ensures that all values of the variable satisfy all the unary constraints on that variable.

- A CSP is arc-consistent [27] if for any pair of constrained variables x_i, x_j, for every value a in D_i there is at least one value b in D_j such that the assignment (x_i, a) and (x_j, b) satisfies the constraint between x_i and x_j. Any value in the domain D_i of variable x_i that is not arc-consistent can be

removed from D_i since it cannot be part of any solution. Arc-consistency has become very important in CSP solving and it is in the heart of many constraint programming languages. The optimal algorithms to make the CSP arc-consistent require time $O(ed^2)$, where e is the number of constraints (arcs in the constraint network) and d is the size of domains. Arc-consistency can also be easily extended to non-binary constraints.

- A CSP is path-consistent [30], if for every pair of values a and b for two variables x_i and x_j , such that the assignments of a to x_i and b to x_j satisfies the constraint between x_i and x_j, there exist a value for each variable along any path between x_i and x_j such that all constraints along the path are satisfied. When a path-consistent problem is also node-consistent and arc-consistent, then the problem is said to be strongly path-consistent.

Consistency techniques can be exploited during the forward checking stage of search algorithms in the following way. Each time some search decision is done (for example, a value is assigned to the variable), the problem is made arc consistent (arc consistency is typically used during search due to it is low time and space complexity). If failure is detected (any domain becomes empty) then it is not necessary to instantiate other variables and backtracking occurs immediately.

2.4.2 Search techniques

Most algorithms for solving CSPs search systematically through the possible assignments of values to variables. Such algorithms are guaranteed to find a solution, if one exists, or to prove that the problem is insoluble. The disadvantage of these algorithms is that they may take a very long time to do so. The actions of many search algorithms can be described by a search tree.

Generate and Test. The generate-and-test (GT) method originates from the naive approach to solving combinatorial problems. First, the GT algorithm guesses the solution, and then it tests whether this solution is correct, that is, whether the solution satisfies the original constraints. In this paradigm, each possible combination of the variable assignments is systematically generated and tested to see if it satisfies all the constraints. The first combination that satisfies all the constraints is the solution. The number of combinations considered by this method is the size of the Cartesian product of all the variable domains.

The main disadvantage is that it is not very efficient because it generates many assignments of values to variables which are rejected in the testing phase. In addition, the generator leaves out the conflicting instantiations and generates other assignments independently of the conflict. Visibly, one can get far better efficiency if the validity of the constraint is tested as soon as its respective variables are instantiated.

Backtracking. A simple algorithm for solving a CSP is backtracking search (BT) [5]. Backtracking works with an initially empty set of consistent instantiated variables and tries to extend the set to a new variable and a value for that variable. If successful, the process is repeated until all variables are included. If unsuccessful, another value for the most recently added variable is considered. Returning to an earlier variable in this way is called a backtrack. If that variable doesn't have any further values, then the variable is removed from the set, and the algorithm backtracks again. The simplest backtracking algorithm is called chronological backtracking because at a dead-end the algorithm returns to the immediately earlier variable in the ordering.

In the BT method, variables are instantiated sequentially and as soon as all the variables relevant to a constraint are instantiated, the validity of the constraint is checked. If a partial solution violates any of the constraints, backtracking is performed to the most recently instantiated variable that still has alternatives available. Clearly, whenever a partial instantiation violates a constraint, backtracking is able to eliminate a subspace from the Cartesian product of all variable domains.

2.4.3 Hybrid techniques

In section 2.4.2 some search techniques to solve CSPs have been presented. In section 2.4.1 some consistency techniques that delete inconsistent values from the variable domain have been presented. Therefore, consistency techniques can be used as a pre-process step where inconsistencies can be detected and removed. Otherwise, consistency techniques can be used in the search process. Some of these hybrid techniques are based on look-back and look-ahead techniques.

- **Look-Back Algorithms:** BT can suffer from thrashing; the same dead end can be encountered many times. If x_i is a dead end, the algorithm will backtrack to x_{i-1}. Suppose a new value for x_{i-1} exists, but that there is no constraint between x_i and x_{i-1}. The same dead end will be reached at x_i again and again until all values of x_{i-1} have been explored. Look-back algorithms try to exploit information from the problem to behave more efficiently in dead-end situations. Like BT, look-back algorithms perform consistency checks backward (between the current variable and past variables).

  Backjumping (BJ) [10] is an algorithm similar to BT except that it behaves in a more intelligent manner when a dead end (x_i) is found. Instead of BT to the previous variable (x_{i-1}), BJ backjumps to the deepest past variable x_j, j, i that is in conflict with the current variable x_i. It is said that variable x_j is in conflict with the current variable x_i if the instantiation of x_j precludes one of the values in x_i. Changing the instantiation of x_j may make it possible to find a consistent instantiation of the current variable. Thus, BJ avoids any redundant work that BT does by trying to reassign variables between x_j and

the current variable x_i. Conflict-directed BJ [33], backmarking [11], and learning [9] are examples of look-back algorithms.

- **Look-Ahead Algorithms:** As we have explained, look-back algorithms try to enhance the performance of BT by more intelligent behavior when a dead end is found. Nevertheless, they still perform only backward consistency checks and ignore the future variables. Look-forward algorithms make forward checks at each step of the search. Let us assume that, when searching for a solution, the variable x_i is given a value which excludes all possible values for the variable x_j. In case of uninformed search, this will only turn out when x_j will be considered to be instantiated. Moreover, in case of BT, thrashing will occur: the search tree will be expanded again and again until x_j, as long as the level of BT does not reach x_i. Both anomalies could be avoided by recognizing that the chosen value for x_i cannot be part of a solution as there is no value for x_j, which is compatible with it. Lookahead algorithms do this by accepting a value for the current variable only if, after having looked ahead, it could not be seen that the instantiation would lead to a dead end. When checking this, problem reduction can also take place by removing values from the domain of the future variables that are not compatible with the current instantiation. The algorithms differ in how far and thorough they look ahead and how much reduction they perform.

  Forward-checking (FC) [16] is one of the most common look-forward algorithms. It checks the satisfiability of the constraints, and removes the values which are not compatible with the current variable's instantiation. At each step, FC checks the current assignment against all the values of future variables that are constrained with the current variable. All values of future variables that are not consistent with the current assignment are removed from their domains. If a domain of a future variable becomes empty, the assignment of the current variable is undone and a new value is assigned. If no value is consistent, then BT is carried out. Thus, FC guarantees that at each step the current partial solution is consistent with each value in each future variable. Thus, FC can identify dead ends and prune the search space sooner.

2.5 Robustness and stability

The concepts of robustness and stability in constraint satisfaction have been mixed and there is a misunderstanding between the two concepts. Some researchers talk about stability and others about robustness. But, what is the difference between stable and robust? It is the first question that comes to mind, especially for researchers who work with quantitative models or mathematical theories.

In general, a solution is stable in a dynamic system, if by means of a few changes in the solution we can obtain a new solution that is similar to the original one. However the robustness concept is broader than the stability concept. Ro-

bustness is a measure of feature persistence in systems that compel us to focus on perturbations because they represent changes in the composition or topology of the system. The perturbations are small differences in the actual state of the system [21].

Taking into account all these concepts, we can classify the nature of the solutions as:

- The stability (also called flexibility) of a solution is the ability of a solution to share as many values as possible with a new solution if a change occurs [17]. It is measured in terms of similarity of the new solution with the original one.

- The robustness of a solution is the measure of the persistence of the solution after modifications in the original solution. Thus, a solution is robust if it has a high probability of remaining valid faced with changes in the problem. It is measured in terms of the persistence of the solution.

Sometimes the robustness is related with the information of the problem. Normally, in a problem that no information is given, the probability of something goes wrong or the probability that something cannot be finished in the predicted time is equal in all the cases. However sometimes there is information about the problem because there are some historic as previous experiments where some probabilistic cases can be used, or the help of an expert can be used. When there is no information usually the probability that something happens has to be proportional.

Chapter 3

Job-Shop Scheduling Problem With Operators

3.1 Introduction to Job-Shop Scheduling Problem

Job-shop scheduling problems (JSP) are among the most intensive combinatorial problems studied in literature. An instance with ten jobs to be processed on ten machines, formulated in 1963, was open for more than 25 years. It was finally solved by a branch-and-bound algorithm. Very simple special cases of the job-shop problem are already strongly NP-hard.

After a short review of these old challenges, we consider practical applications in flexible manufacturing, multiprocessor task scheduling, robotic cell scheduling, railway scheduling, air traffic control which all have an underlying job-shop structure. Methods to solve these problems and new challenges in connection with them are indicated.

3.2 Problem Description

The job-shop scheduling problem can be defined as follows. We are given a set $\{J_1, \ldots, J_n\}$ of n jobs that require, for their processing, a set of m resources or machines $\{R_1, \ldots, R_m\}$. Each job J_i consists of a sequence of v_i tasks $(\theta_{i1}, \ldots, \theta_{iv_i})$. Each task θ_{il} has a single resource requirement $R_{\theta_{il}}$, an integer duration $p_{\theta_{il}}$ and a start time $st_{\theta_{il}}$ to be determined. A feasible schedule is a complete assignment of starting times to tasks that satisfies the following constraints: (i) the tasks of each job are sequentially scheduled, (ii) each machine can process at most one task at any time, (iii) no preemption is allowed. The objective is finding a feasible schedule that minimizes the completion time of all the tasks, i.e. the makespan.

In this framework, it is useful to represent the job-shop scheduling problem in terms of a disjunctive graph $G = (V, A, E)$ [2], where V is the set of nodes, A is the set of ordinary arcs (conjunctive) and E the set of disjunctive arcs. The nodes of G (V) correspond to operations, the directed arcs (A) to precedence relation,

and the disjunctive arcs (E) to operations to be performed on the same machine. A schedule on a disjunctive graph G consists on finding a set of orientations that minimizes the length of the longest path (critical path) in the resulting acyclic directed graph. Let consider the following example:

Table 3.1: Example of a job-shop problem

Job	**Processing cycle**
J_1	(1,10) ,(2,5), (3,6)
J_2	(2,5) ,(1,8)
J_3	(1,2) ,(3,10), (2,4)

The job-shop scheduling problem has three jobs, three machines and eight operations. J_1 consists of a sequence of three operations, J_2 consists of a sequence of two operations and J_3 consists of a sequence of three operations. The processing cycle for each job is a sequence of items ($R_{\theta_{il}}$, $p_{\theta_{il}}$) where $R_{\theta_{il}}$ denotes the machine and $p_{\theta_{il}}$ the processing time for the operation v_i, respectively. The disjunctive graph of the above problem is shown in figure 3.1.

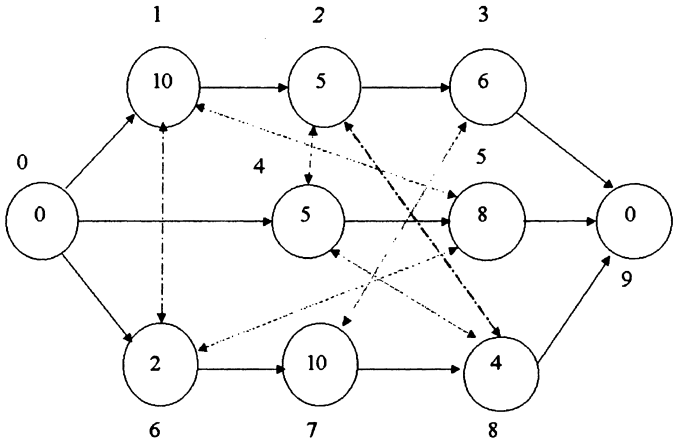

Figure 3.1: Disjunctive graph for a job-shop scheduling problem.

The number outside a node represents the number of operation $v \in \theta_{il}$, whereas the number inside a node is the processing time $p_{\theta_{il}}$. A feasible solution of the problem is represented by the directed graph shown in figure 3.2.

3.3 Techniques to solve Job-Shop Scheduling Problem

Many techniques have been developed to solve the job-shop scheduling problem. In this section some of these techniques are presented.

Branch and Bound (BB) use a dynamically constructed tree structure to represents the solution space of all feasible sequences. The search begins at the

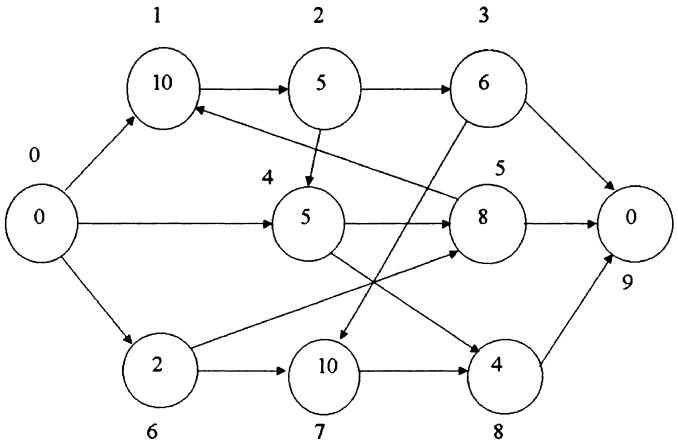

Figure 3.2: A feasible solution via a direct graph.

topmost node and a complete selection is achieved once the lowest level node has been evaluated. Each node at a level in the search tree represents a partial sequence of operations. As implied by their name a branching as well as a bounding scheme is applied to perform the search. From an unselected node the branching operation determines the next set of possible nodes from which the search could progress. The BB search technique was initially studied by [7] and [19]. Another example of such a BB algorithm is cited by [6] who construct a parallel version of edge finder. A depth first strategy is applied and the partial enumeration tree is represented by a collection of data structures held in shared memory.

Constraint Satisfaction techniques aim at reducing the effective size of the search space by applying constraints that restrict the order in which variables are selected and the sequence in which possible values are assigned to each variable. After a value is assigned to a variable any inconsistency arising is removed. The process of removing inconsistent values is called consistency checking, while the method of undoing previous assignments is referred to as backtracking. A backtrack search fixes an order on the variables and also determines a fixed ordering of the values of each domain. The Constraint Satisfaction Problem (CSP) is solved when a complete allocation of variables is specified that does not violate the constraints of the problem. Although considered within the domain of AI, many constraint based scheduling methods apply a systematic tree search and have close links with BB algorithms. [31] applies various propagation methods and operation selection heuristics in order to dynamically determine whether to schedule an operation first or last. [3] proposes a constraint-based model for the Job-Shop Scheduling Problem to be solved using local search techniques.

Neural Networks (NNs) are organised in a framework based on the brain struc-

ture of simple living entities. In these techniques information processing is carried out through a massively interconnected network of parallel processing units. Their simplicity, along with their capability to perform distributed computing, as well as their propensity to learn and generalise has made neural networks a popular methodology. [20] describes some of the main neural network architectures applied to solve scheduling problems.

Greedy Randomised Adaptive Search Procedure (GRASP) is a problem space based method that consists of a constructive and an iterative phase. In the construction phase the solution is built one element at a time. All possible elements which can be chosen next are ordered in a candidate list with respect to a greedy function. A number of the best candidates are then placed in a restricted candidate list (RCL). The adaptive nature of GRASP is derived from its ability to update the values associated with every element, at each iteration, based on the selection just made. While the probabilistic nature of the algorithm stems from the random selection of an element in the RCL. [34] present an application on presents an application of GRASP where the RCL consists of the operations that when sequenced next would result in the lowest overall completion time of the partial sequence.

Genetic Algorithms (GAs) are based on an abstract model of natural evolution, such that the quality of individuals builds to the highest level compatible with the environment. Genetic Algorithms are an optimization technique for functions defined over finite domains. They were first proposed in [18]. Their name refers to their imitation of natural evolutionary processes. In scheduling and ordering problems, encoding is usually such that the chromosomes are the permutation of a basic string. An example of representation is showed by [23] where a chromosome is a string of symbols of length n and each symbol identifies the operation to be processed on a machine. Another work where a GA are used to solve a JSP is presented in [15] the chromosome representation of the problem is based on random keys. The schedules are constructed using a priority rule in which the priorities are defined by the genetic algorithm and after a schedule is obtained a local search heuristic is applied to improve the solution.

The Tabu Search (TS) is a metaheuristic approach mainly used to find a near-optimal solution of combinatorial optimization problems. It was proposed and formalized by [12], [14] and [13]. The TS technique is based on an iterative procedure "neighbourhood search method" for finding, in a finite set T of feasible solutions, a solution $t \in T$ that minimizes a real-valued objective function f. [24] present some of the earliest TS approaches in scheduling. They create three tabu search strategies based on simple move definitions. [25] apply target analysis to these two works and indicate the inclusion of job transfers in addition to job swaps improves solution quality, reduces computing time and allows larger problems to be solved. In [32]

a new heuristics method based on a combination of a TS technique and the SBP has been proposed. The initial solution given by shifting bottleneck, the special structure of neighbourhood, and the proposed dynamic list allow to obtain interesting results.

3.4 Extension with operators

The job-shop scheduling problem (JSP) with operators is an extension of the classical job-shop scheduling problem. This extension, proposed by Agnetis et al. [1], is denoted $JSO(n, p)$ where n represents the number of jobs and p is the number of operators. The number of jobs n have to be greater than the number of operators p otherwise each operators could be assigned to each job and the remains as a standard JSP.

The extension to $JSO(n, p)$ produce some changes in the problem formulation:

- A new constraint is added: Each operation has to be assisted by one operator and each operator cannot assist more than one operation at the same time.

- The representation graph G has some changes: G was defined as (V, A, E) where V represents the nodes, A precedence relations and E the capacity constraints. The new graph representation is represented as $G = (V, A \cup E \cup I \cup O)$ [39] where V,A and E following represents the same. While O represents the operators arcs that includes three types of arc: one arc (u, v) for each pair of tasks of the problem, and arcs (O_i^{start}, u) and (u, O_i^{end}) for each operator node and task. The set I includes arcs connecting node *start* to each node O_i^{start} and arcs connecting each node O_i^{end} to node *end*. The arcs are weighted with the processing time of the task at the source node.

In [1] the authors made a thorough study of this problem and established the minimal NP-hard cases. Also, a number of exact and approximate algorithms to cope with this problem were proposed and evaluated on a set of instances generated from those minimal relevant cases. The results of the experimental study reported in [1] make it clear that instances of the $JSO(n, p)$ with 3 jobs, 3 machines, 2 operators and a number of 30 tasks per job may be hard to solve to optimality.

In [40] the authors propose an exact best-first search algorithm and experiment with new instances considering more than 3 jobs and 2 operators. Also, a genetic algorithm is proposed in [29] which reaches near optimal solutions for large instances. The $JSO(n, p)$ with total flow time minimization is considered in [39] where it is solved by means of an exact best first search algorithm and in [28] by means of an depth-first search algorithm. In both cases, some problem dependent heuristics and powerful pruning rules were used.

All the developed techniques are focused on obtaining optimized solutions according to makespan and total flow time. In this work is extended the objective for

searching robust solutions. It is well-known that real life scheduling problems are dynamic and incidences may occur so that an optimal solution remains unfeasible after the incidence. The main incidence that can occur in a $JSO(n,p)$ is that a task must be delayed due to problems with the associated machine or assigned operator. In this way, our main goal is to find robust and optimized solutions to these problems.

3.5 Robustness in Job-Shop Scheduling Problem

"*Robustness*" can be defined as the ability of a system to withstand stresses, pressures, perturbations, unpredictable changes or variations in its operating environment without loss of functionality. A system designed to perform in an expected environment is "*robust*" if it is able to maintain its functionality under a set of incidences. In our context a solution of a $JSO(n,p)$ is robust if no rescheduling is needed after small changes in the problem.

Intuitively, the notion of robustness is easy to define, but its formalization depends on the system, on its expected functionality and on the particular set of incidences to face up [35]. No general formal definition of robustness has been proposed, except few exceptions or particular cases. Particularly, Kitano [22] mathematically defines the robustness (R) of a system (SYS) with regard to its expected functionality (F) against a set of perturbations (Z), as (in a simplified way):

$$R_{F,Z}^{SYS} = \int_Z p(z) * F(z) dz \tag{3.1}$$

The application of robustness definitions is highly problem-dependent. Let's apply (3.1) to $JSO(n,p)$:

- SYS is a solution S of the $JSO(n,p)$, which we want to assess its robustness. Robustness is a concept related to $JSO(n,p)$ solutions, not to $JSO(n,p)$ itself.
- Z is the discrete set of unexpected incidences that are directly related to the start time or the duration of tasks.
- F is the expected functionality of the system. In $JSO(n,p)$, the expected functionality of a solution is its feasibility after the disruption. Here a solution is composed by the start times and duration of all tasks, plus the buffer times allocated between tasks.
- $p(z) = \frac{1}{|z|}, \forall z \in Z$. This is the probability for incidence $z \in Z$. All tasks have the same probability due to no information is given about incidences.

Therefore, the expression (3.1) becomes:

$$R_{F,Z}^{S} = \sum_Z p(z) * F(z) \tag{3.2}$$

Where function F is defined, in the case of a $JSO(n, p)$ as:

- $F(z) = 1$ iff only the affected task is modified by z. Thus the buffer assigned to this task can absorb the incidence.
- $F(z) = 0$, iff more tasks are modified by z. This means that the buffer assigned to this task cannot absorb the incidence and it is propagated to the rest of the schedule.

A robust solution is a solution that maintains its feasibility over the whole set of expected incidences. Thus, robustness in $JSO(n, p)$ implies that:

- If the duration of a task is greater than expected, then only its final time will be affected and no other tasks will be delayed.
- If the start time of a task is delayed, then its final time is also delayed but no other tasks will be affected.

Here, we focus our attention on searching for a robust solution with a minimal makespan. To do this, we will assign buffer times to tasks. This is an usual way for introducing robustness in scheduling problems. A buffer is an extra time that is given to a task to absorb small incidences. Due to the fact that the duration of a task is directly dependent of machine and operator involved in this task, the buffer assigned to this task can be used to absorb small incidences in these two components. However, there exists a trade-off between optimality and robustness so buffer times cannot be assigned to all tasks. If it is not known information about the probability to appear the possible interruptions the number buffers have to be as high as possible and these have to be good distributed.

Lemma 1. Let a robust schedule with a given makespan. A buffer can be assigned to a task iff this task is not involved in any critical path.

Proof by contradiction

$\rightarrow$ If a buffer is assigned to a task and it is involved in a critical path, then this buffer could be removed to reduce makespan. Contradiction: the initial schedule has minimum makespan.

$\leftarrow$ It is straightforward. If a task is not involved in a critical path and a buffer cannot be assigned, then this task takes part of another critical path. Contradiction: this task is not involved in any critical path.

Thus, we consider that tasks involved in a critical path will not be assigned buffers to avoid increasing the makespan. Thus our main goal is to assign buffers to all tasks that are not involved in critical paths, so that we could achieve the maximum robustness with a given optimality (makespan).

Chapter 4

Modelling $JSO(n, p)$ as a CSP

4.1 Introduction

In this chapter $JSO(n, p)$ is modelled as a CSP and solved using a look-ahead algorithm (Forward-checking). By using this algorithm, a solution can be obtained, However the main aim of this problem is not to obtain a solution but to obtain an optimized solution. To this end a branch and bound technique has been developed. When a solution is found the algorithm remains looking for more solutions that improve the obtained solutions. When the objective is not only to get a solution of the problem but an optimized solution it is named Constraint Satisfaction and Optimization Problem (CSOP).

4.2 A modelling Phase

In this approach, we proposed to model the $JSO(n, p)$ as a Constraint Satisfaction and Optimization Problem (CSOP) [4].

The CSOP model for a $JSO(n, p)$ is characterized by the following elements:

- A set of variables $x_1, ..., x_n$ associated with the start time of tasks and with the operator responsible for carrying out each task. These variables take values in finite domains $D_1, ..., D_n$ that may be constrained by unary constraints over each variable. In these problems, time is usually assumed discrete, with a problem-dependent granularity.
- A set of constraints $c_1, ..., c_m$ among variables defined on the Cartesian product $D_i \times ... \times D_j$ and restrict the variable domains.
- The objective function is to minimize the makespan.

Three main constraints appear in this kind of job-shop problems:

1. Precedence constraints: The tasks θ_{ij} of each job J_i must be scheduled according to precedence constraints, i.e., there exists a partial ordering among

the tasks of each job and may be represented by a precedence graph or tree-like structure [38].

2. Capacity constraints: Resources cannot be used simultaneously by more than one task. Thus, two different tasks θ_{ij} and θ_{ik} cannot overlap unless they use different resources.

3. Operator constraints: An operator can not handle more than one task at a time.

To modeling the $JSO(n, p)$ as a CSOP we have used the syntax XCSP [36]. The Extensible Markup Language presents a simple and flexible text format and it gives the facility to use some functions and structures defined in Abscon [26]. In the figure 4.1, it can be seen an example of a representation of a job-shop partial instance represented in XCSP.

Figure 4.1: Partial instance of a job-shop problem represented in XCSP.

In the modeling phase, we have applied two different filtering techniques to reduce the variable domains. The first technique developed (Algorithm 1) calculates initials values of each tasks and reduces the domain size of the involved variables. These techniques are similar to the filtering techniques presented in chapter 2 (node consistency, arc-consistency).Thus, a solution can be found more efficiently. This

algorithm calculates the maximum time interval in which each task can be scheduled. On the one hand, given a θ_{ij} task, the lowest value of its $st_{\theta_{ij}}$ is the sum of the processing times of the tasks that has to be scheduled before θ_{ij} from the same job i ($cumulative_{ij}$), subject to the precedence constraints. On the other hand, the highest value of all the domains for $st_{\theta_{ij}}$ is the sum of all processing times ($maxTime$), since this value represents the end of the schedule where the tasks are scheduled linearly. Due to the fact that at least the following tasks from the same job must be scheduled before $maxTime$, the highest value for the domain of each task θ_{ij} can be reduced by subtracting the duration of all the tasks from the same job i that have to be scheduled after θ_{ij} (including θ_{ij}) to $maxTime$.

Algorithm 1: Calculate initial values to reduce domain

Data: J: set of jobs;
Result: Relative starts to each task and lineal maxtime
$maxTime := 0$;
$cumulative_{ij} := 0,\ \forall \theta_{ij} \in \theta$;
foreach $i \in J$ **do**
    $cumulativeJob \leftarrow \{0\}$;
    foreach $\theta_{ij} \in \theta$ **do**
        $maxTime := maxTime + p_t$;
        $cumulative_{ij} \leftarrow cumulativeJob$;
        $cumulativeJob \leftarrow cumulativeJob \cup \{p_{\theta_{ij}}\}$;
return $cumulative, maxTime$;

The values $cumulative_{ij}$ and $maxTime$ obtained in Algorithm 1 are used to filter the domains by removing values that cannot take part of feasible solutions.

In the second technique to reduce the variable domains (Algorithm 2), the values that cannot be possible are calculated. $st_{\theta_{ij}}$ only should get values that represent the sum of the tasks that can be executed before θ_{ij}.

Algorithm 2: Calculate possible values

Data: All tasks
Result: New domain of all θ
$pValues_{jt} \leftarrow \emptyset,\ \forall \theta_{ij} \in \theta$;
foreach $i \in J$ **do**
    foreach $\theta_{ij} \in \theta$ **do**
        $tasksBefore \leftarrow$ `tasksCanBeBefore`(θ_{ij})
        $pValues_{ij} \leftarrow$ `combineDurTasks`$(tasksBefore)$
return $pValues$

For example, if θ_{ij} is the first task of its job, the possible values are 0 and a set of the combination of the processing times of all the tasks T that can be scheduled before θ_{ij}. In this case, these tasks T are all the tasks of the other jobs. For the following task (θ_{ij+1}), its possible values $st_{\theta_{ij+1}}$ are the same as θ_{ij} plus the processing time of θ_{ij}. In the Algorithm 2, *tasksCanBeBefore* function returns the tasks that can be executed before a given task θ_{ij}; and, *combineDurTask* function

calculates all the possible values for $st_{\theta_{ij}}$ following the precedence constraints. Thus, by applying these filtering techniques, the variable domains are reduced and the solving phase can be executed in a more efficient way.

4.3 Solving Phase

Once a CSOP has been modeled and the domains filtered, it is solved by using a modified algorithm of Forward Checking (FC) (Algorithm 3). When a solution has been found, the algorithm tries to found more solutions with a Branch and Bound technique. Filtering techniques are also used but prune is only used when the set of not analysed solutions cannot improve the best obtained one. For this reason no solution is lost. Instead of applying Maintaining Arc-Consistency, this algorithm performs a filtering procedure that removes the domains in blocks, i.e., it removes several values at a time. This is due to the fact that if a task θ_{ij} must be scheduled after task θ_{kl}, the $st_{\theta_{ij}}$ can take neither the value of the $st_{\theta_{kl}}$ nor all successive values until the ending time of θ_{kl}. Thus, all these values can be removed as a block of values. The values to delete depend on the type of constraint. If this is a precedence constraint, all the values before θ_{kl} plus its processing time will be deleted. Otherwise, if it is a capacity constraint, the values between $st_{\theta_{kl}}$ (included) and its processing time will be deleted.

Algorithm 3: Select-value-forward-checking with deletion block

```
while D'_i ≠ ∅ do
    select first element a ∈ D'_i, and remove a from D_i
    forall k, i < k ≤ |D'| do
        if constraint(i,k) = MachineOrJobConstraint then
            if constraint(i,k)= JobConstraint then
                remove values if(val_k < a + dur_i) from D'_k
            else
                remove values if(val_k + dur_k ≥ a ∧ val_k < a + dur_i) from D'_k
        else
            forall b ∈ D'_k do
                if not CONSISTENT (a_{i-1}, x_i := a, x_k := b) then
                    remove b from D'_k
        if D'_k = ∅ then
            reset each D'_k, i < k ≤ n to value before a was selected
        else
            return a
return null
```

Some heuristic techniques have been applied to improve the speed and efficiently solving the problem. Variable selection and value selection are two critical tasks, and with a good heuristic technique can improve the results. The variable selection is really important because a good solution can be obtained early and it makes more efficient the prune. Also the value selection is important because the

values of temporal variables are selected in increase order and the smallest value for each temporal variable is selected. This is due to the fact that if a task can start in a defined domain of time interval, the earliest value will be probably the value that minimizes the makespan. The variable selection heuristic that has given better results is to select the firsts tasks of each job.

Chapter 5

Heuristic Algorithms for Finding Robust Solutions in Job-Shop Scheduling Problem with Operators

5.1 Introduction

In chapter 4 a model to solve $JSO(n, p)$ with a CSP implementation with the objective to get an optimal solution of the problem has been presented. It is well-known the trade-off between robustness and optimality. This aim would be used to calculate a robust solution without lose a big part of optimality. But this job is really difficult to archive, so a new technique has been developed and it is presented in this chapter. Our technique has been classified in three steps [8]. At the first step, the job shop problem without taking account operators is solved, so that an optimized solution minimizing the makespan is obtained. In the second step this solution is modified to take into account the operators, modifying the problem to a $JSO(n, p)$, but now the aim is to get a solution that present a good trade-off between makespan and robustness. In the third step the solution is modified to redistribute the buffers but maintaining the same makespan.

5.2 First Step: Modeling and Solving a Constraint Satisfaction and Optimization Problem

In this approach, we proposed to model the JSP, in the first phase, as a Constraint Satisfaction and Optimization Problem (CSOP) [4]. Due to the post-process performed, the optimal solution is not necessary to archive, since this solution will be changed. However, it is still necessary an optimized solution to try to minimize the makespan. Therefore, a near-optimal solution is looked for by our CSOP solver.

Nevertheless any solver can be used applied to obtain an optimized solution. Our CSOP solver is an any-time solver that provides a set of solutions. Each new solution always improves the previous one, until an optimal or a time out is reached.

The phases of modeling and solving are similar to those described in Chapter 4 but without take into account the operators. Only precedence and capacity constraints are take into account. To model the problem Algorithm 1 and 2 have been used to calculate the initial values of each tasks and reduce the domain. There techniques can be used in a JSP because they do not take into account operators.

In the solving phase the techniques presented in Chapter 4 are also used. The problem is solved using the modified FC (Algorithm 3) explained in Chapter 4.

5.3 Second Step: A Post-process Procedure

Once the CSOP has been solved and an optimized solution to the job-shop scheduling problem have been obtained, this solution is used to allocate the required number of operators in order to solve the $JSO(n, p)$. This problem consists on finding a feasible schedule by minimizing makespan and maximizing number of buffers (N_{buf}) to guarantee a certain level of robustness. Note that a feasible schedule for $JSO(n, p)$ is also feasible for the standard job-shop problem and satisfies the restriction that at most p machines work simultaneously. Therefore, significant cases are those in which $p < min(n, m)$, otherwise our problem becomes a standard job-shop problem [1].

The aim of the Algorithm 4 is to convert a solution without operators in one where the operator constraints are considered. The idea is to set a number of machines ($remainingMachines$) equal to the number of operators p and try to reschedule the tasks of the other machines ($machinesFewerTasks$) within the $remainingMachines$. The tasks in $machinesFewerTasks$ must be sorted by their st ($tasksToPut$). Each θ_{ij} in $tasksToPut$ is allocated in the first available gap between two tasks of each machine in $remainingMachines$. For each machine, the search starts from the previous state ($savedState$). There are cases where θ_{ij} must be allocated without a gap between two tasks due to the precedence constraints. For instance, if we found a θ_{ik} of the same job as θ_{ij} and θ_{ik} must be scheduled after θ_{ij} according to the precedence constraints ($k > j$), θ_{ij} is allocated just before θ_{ik}, being delayed θ_{ik}. When a task is delayed, other tasks may be also delayed. The computational cost of the algorithm is $O(tasksToPut * |remainingMachines|)$.

The best state to allocate θ_{ij} is the state that maximizes the function $\frac{N_{buf}}{C_{max}}$. This function could be adjusted depending on the requirements of the user, e.g. either only minimizing the makespan or maximizing the number of buffers generated.

Algorithm 4: Post-process

Data: S: Solution without operators; m: machines; p: operators;
Result: A solution considering operators

Order the machines by their number of tasks;
$machinesFewerTasks \leftarrow$ set of $m - p$ machines with fewer tasks;
$tasksToPut \leftarrow$ tasks of the machines $machinesFewerTasks$;
$remainingMachines \leftarrow m - machinesFewerTasks$;
Order $tasksToPut$ by Starting Times;

$actualState \leftarrow S$;
foreach $\theta_i \in tasksToPut$ **do**
    $savedState \leftarrow actualState$;
    $states \leftarrow \{\}$;
    for $r \in remainingMachines$ **do**
        $foundGap :=$ `IsThereAGap`(r);
        if $not\ foundGap$ **then**
            insert θ_i before the next task according to its Job;
        else
            insert θ_i in this gap;
        Delay the needed tasks according to the restrictions among them;
        $states \leftarrow states \cup \{actualState\}$;
        $actualState \leftarrow savedState$;
    $actualState \leftarrow$ `chooseBestState`$(states)$;

return $actualState$;

5.4 Third Step: Distributing Buffer Algorithm

The previous step gives us an optimized solution that satisfies all constraints of the $JSO(n, p)$. This solution is an optimized solution in terms of minimizing makespan and maximizing the number of buffers. However, the main goal for a robust solution, in a scheduling problem where no information about incidences is given, is to distribute the amount of available buffers among as many tasks as possible. It is well-known that all tasks involved in a critical path have not any associate buffer, because it will affect the makespan. The rest of tasks can be reassigned to generate a buffer after their ending time. The main goal of this algorithm is to distribute the amount of buffers without affecting this makespan. Thus, we can maximize the number. In this way, the obtained solution is considered more robust due to more tasks have buffer times to absorb small incidences. Figure 5.1(b) shows the solution obtained in the second step and all critical paths. It can be observed that 10 buffers were generated, meanwhile the distributing buffer algorithm was able to find 14 buffers. We remark that our goal is to obtain the maximum number of buffers since no information is given about incidences so that all tasks have the same probability for delaying.

This third step is presented in Algorithm 5. This algorithm looks for the tasks θ_{ij} allocated just before each buffer generated in Algorithm 4, and tries to set them back. A task θ_{ij} is only set back if it generates a new buffer. In case a new buffer

Algorithm 5: Distributing buffers

Data: Sch: Schedule; $buffers$;
Result: New Schedule

foreach $b \in buffers$ **do**
    $sizeB$:=size of the buffer b;
    repeat
        $continue := false$;
        θ_{ij} := task allocated before b;
        Set back θ_{ij};
        if *this movement generates another buffer* nb **then**
            $continue := true$;
            Update schedule Sch;
    until $continue$;
return Sch;

nb is generated, this process is repeated with the task just before nb. The computational cost of the algorithm is $O(\text{tasks in non-critical path})$, because tasks in the non-critical path are the only ones that can be moved to distribute the buffers.

5.5 An example

Figure 5.1 shows different schedules obtained by the CP Optimizer solver and our technique for a given instance of the $JSO(n, p)$. This instance represents a scheduling problem with 3 jobs, each with 10 tasks, and 2 operators. Each rectangle represents a task whose length corresponds to its processing time. Inside the rectangle, the job, task, machine and operator are showed. The dotted lines showed in these schedules represent the critical path.

The first schedule Figure 5.1(a) represents the distribution of tasks of the optimal solution obtained by CP Optimizer (only minimizing the makespan) according to the operators. It can be observed that the obtained makespan was 57 but only 1 buffer was generated due to the fact that only task $\theta_{1,10}$ was not involved in any critical path (green and red lines in Figure 5.1(a)). Taking into consideration the robustness within the objective function, the Figure 5.1(b) represents the solution obtained by applying step1 + step2 of our algorithm, where all tasks were distributed by operators.

Finally, Figure 5.1(c) represents the schedule obtained by the third step of our algorithm. Although the makespan was increased up to 67, it can be seen that the buffers (black boxes) were distributed between all tasks that they did not take part of any critical path being increased the robustness of this schedule. These buffers can be used to absorb incidences or delays from the previous task. For instance, if the resource assigned to the tasks θ_{23} suffers a small failure, the solution could not be affected.

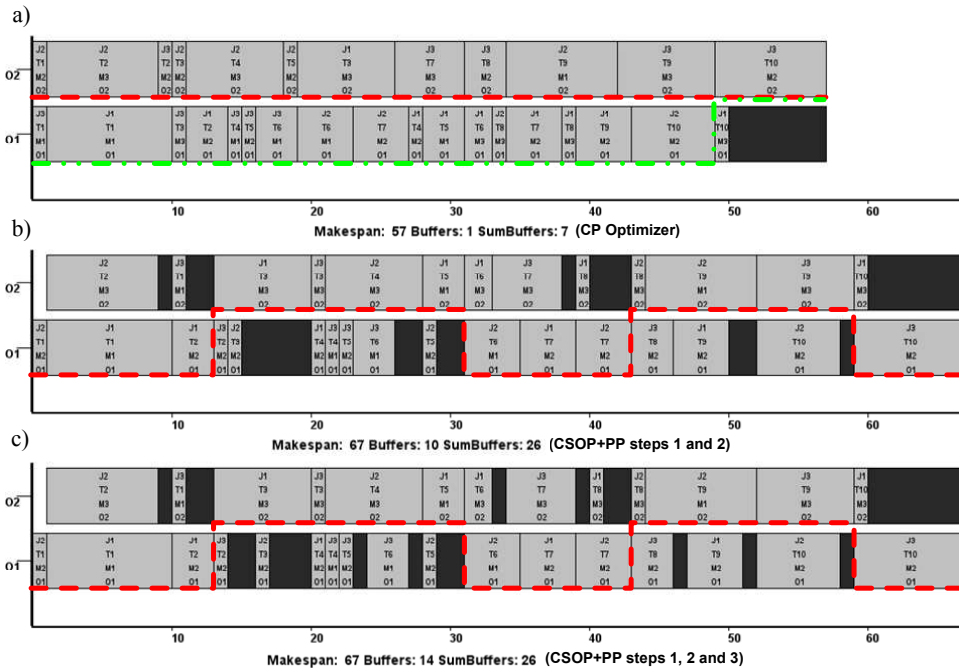

Figure 5.1: A scheduling problem: an optimal and a robust solution.

Chapter 6

Evaluation

6.1 $JSO(n, p)$ CSP model Evaluation

We have experimented across the benchmarks proposed in [1], where all instances have $n = 3$ and $p = 2$ and are characterized by the number of machines (m), the maximum number of tasks per job (v_{max}) and the range of processing times (p_i). A set of instances was generated combining three values of each parameter: $m = 3, 5, 7$; $v_{max} = 5, 7, 10$ and $p_i = [1, 10], [1, 50], [1, 100]$. In all cases, 10 instances were considered from each combination and the average results are shown in the next tables. The sets of instances are identified by the tuple: $< m\_v_{max}\_p_i >$.

The set of problems were solved with the algorithm explained in chapter 4. The aim of the algorithm is found a solution that minimizes the makespan. The tables 6.1, 6.2 and 6.3 show the results for the instances with 10, 50 and 100 of processing time. The column $\mathbf{C_{max}}$ represents the average obtained makespan of ten instances, the column time represent the computational time (in seconds) to find these solutions with a maximum time of 300 seconds, the column Optimal $\mathbf{C_{max}}$ represents the average optimal makespan for 10 instances and the column Diff $\mathbf{C_{max}}$ represents the difference between the obtained makespan and the optimal.

The computational time to solve a the instance was variable because a solution for a specific instance could be found in 10 seconds and no more solutions were found in the 300 seconds so the time was 10 seconds but this solution was not a good solution. Otherwise for other instance a solution was found in 10 seconds again but it was improved in 250 seconds by the optimal solution so in this case the computational time was 250 but this solution was better.

For the instances where $p_i = [1, 10]$ the obtained makespan was near to the optimal because the temporal domain was smaller and so the problem was smaller. The average difference makespan for $p_i = [1, 10]$ was 1.73 moreover for instance with $p_i = [1, 50]$ was 20.41 and for the instance with $p_i = [1, 100]$ was 41.81. But such is not the same 5 units of time (u/t) plus for a solution with a makespan 50 than 500, because for the first makespan represents an increase of 10% but for the second an increase of 1%. For this reason percentage for the increment of makespan

Table 6.1: Avg. Makespan and Computational time ($p_i = [1, 10]$)

Instance	**CSP with Operators**			
	$\mathbf{C_{max}}$	Time	Optimal $\mathbf{C_{max}}$	Diff $\mathbf{C_{max}}$
3_5_10	44,20	46,35	42,70	1,50
3_7_10	60,90	63,90	57,90	3,00
3_10_10	68,70	24,56	65,20	3,50
5_5_10	40,30	43,84	39,40	0,90
5_7_10	55,10	70,00	53,90	1,20
5_10_10	63,30	15,83	60,60	2,70
7_5_10	32,10	3,00	31,00	1,10
7_7_10	46,30	11,92	44,70	1,60
7_10_10	72,20	22,18	70,40	1,80

Table 6.2: Avg. Makespan and Computational time ($p_i = [1, 50]$)

Instance	**CSP with Operators**			
	$\mathbf{C_{max}}$	Time	Optimal $\mathbf{C_{max}}$	Diff $\mathbf{C_{max}}$
3_5_50	218,90	49,29	201,10	17,80
3_7_50	296,10	4,68	267,20	28,90
3_10_50	385,80	7,72	353,30	32,50
5_5_50	200,20	22,28	184,30	15,90
5_7_50	274,00	36,93	253,50	20,50
5_10_50	395,40	18,55	363,00	32,40
7_5_50	186,00	1,77	177,10	8,90
7_7_50	275,10	19,36	252,00	23,10
7_10_50	388,00	12,46	363,90	24,10

Table 6.3: Avg. Makespan and Computational time ($p_i = [1, 100]$)

Instance	**CSP with Operators**			
	$\mathbf{C_{max}}$	Time	Optimal $\mathbf{C_{max}}$	Diff $\mathbf{C_{max}}$
3_5_100	412,80	36,94	389,00	23,80
3_7_100	617,80	3,93	561,40	56,40
3_10_100	858,80	37,45	768,70	90,10
5_5_100	391,60	9,42	349,00	42,60
5_7_100	536,40	53,43	495,00	41,40
5_10_100	771,60	55,84	698,20	73,40
7_5_100	411,90	33,70	391,40	20,50
7_7_100	533,40	12,80	516,00	17,40
7_10_100	797,30	24,99	744,80	52,50

were calculated for the three tables and the results were 3.27% , 7.53% and 7.46% correspondingly. The domain size was bigger in the temporal variables when the duration of the task was bigger. These results showed that the result obtained in the instances with a short domain were simpler to solve than the instances with a bigger domain.

Another thing to note is that the results improved when the number of machines increase and this was because if the problem was more restricted then search space was reduced. This can be observed in table 6.3 when the lower value of Diff $\mathbf{C_{max}}$ was obtained in the instances with 7 machines comparing between the same value of $v_{max} = 5, 7, 10$.

6.2 Evaluation of heuristic algorithm for robust solutions in $JSO(n, p)$

The purpose of this experimental study is to assess our proposal CSOP+ Post procedures (PP) and to compare it with the IBM ILOG CPLEX CP Optimizer tool (CP). In CP, the p operators were modeled as a nonrenewable cumulative resource of capacity p. Also, the CP was set to exploit constraint propagation on no overlap (*NoOverlap*) and cumulative function (*CumulFunction*) constraints to extended level. The search strategy used was Depth First Search with restarts (default configuration).

We have experimented with the same benchmark of the previous section, where all instances have $n = 3$ and $p = 2$ and are characterized by the number of machines (m), the maximum number of tasks per job (v_{max}) and the range of processing times (p_i). A set of instances was generated combining three values of each parameter: $m = 3, 5, 7$; $v_{max} = 5, 7, 10$ and $p_i = [1, 10], [1, 50], [1, 100]$.

In Figure 6.1, the solutions showed are the schedules for an instance $<$ 3_5_50 $>$ obtained by the CSOP+PP after both the step 2 (CSOP Step2) and step 3 (CSOP Step3). The smoothed curve of CSOP Step2 and CSOP Step3 are represented by dotted lines; by the CP Optimizer without taking into account the operators and applying steps 2 (CP Step2) and 3 (CP Step3); and, by the CP Optimizer taking into consideration operators (CP operators). Since two objective functions are considered in this problem, the solutions that are not dominated by any other solution are marked with a black point (Non-Dom Sols). It can be observed that keeping the same makespan (C_{max}), solutions given by the step 3 always outperform the ones obtained by the step 2 according to N_{buf}. It is important to note that in order to achieve the minimal C_{max}, there have to be few N_{buf}; and vice versa, to obtain more N_{buf}, it is needed to increase the C_{max}. Among the non-dominate solutions obtained, there is no optimal schedule with the lowest C_{max} and the maximum N_{buf}, therefore, the users should choose among them according to their necessities. For instance, let the solutions space is subdivided in four squares according to whether they minimize or maximize each objective. If the user just needs maximizing the N_{buf} (achieving better robustness), the solutions needed are from the

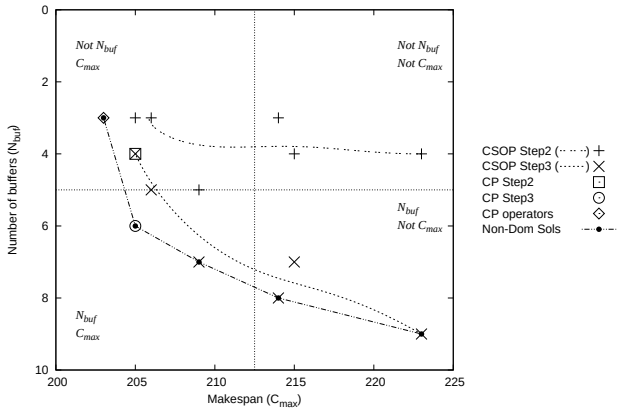

Figure 6.1: Set of solutions for an instance given

right-down square.

In the next experiment the first solution given by the CSOP has been chosen to apply the post-procedure mentioned above because it is the solution that gives the opportunity to get more N_{buf}. This first solution is compared against the optimal solution. In all cases, 10 instances were considered from each combination and the average results are shown in the next tables. The sets of instances are identified by the tuple: $< m\_v_{max}\_p_i >$. The incidences (Z) to assess the robustness were modeled as a delay in a random task θ_{ij} from the schedule. For each instance, a set of 100 incidences were generated with a delay (d) that follows a uniform distribution between 1 and a 10% of p_i.

Tables 6.4(a), 6.4(b) and 6.4(c) show the performance of both techniques to absorb incidences. For each technique, we report the C_{max}, the number of buffers generated (N_{buf}) in step 3, and the robustness (R) for instances for each m, v_{max} and p_i.

Following Lemma 1, the number of buffers showed in these tables is a lower bound of the number of tasks that are not involved in any critical path. For example, in instances $< 3\_10\_10 >$ the optimal solution had an average of 1.2 buffers in the 10 instances evaluated, meanwhile our technique obtained an average of 10.90 buffers in the same instances evaluated. This indicates that in average 10.90 out of 30 tasks were not involved in any critical path and a disruption in one or some of them could be absorbed and the rest of the tasks would not be involved in the disruption.

In all instances the average number of buffers obtained by CSOP+PP was bigger than the ones obtained by CP Optimizer. According to the robustness measure and the number of buffers generated, CSOP+PP procedure always outperformed the solutions given by the CP Optimizer, although the makespan turned out to be increased. For instance, in Table 6.4(a) the instances $< 3\_10\_10 >$ increased up to 32.6% the number of incidences absorbed.

Table 6.4: Avg. Makespan and Robustness

(a) Maximum delay 1 time units

Instance	CP Optimizer			CSOP+PP		
	$\mathbf{C_{max}}$	$\mathbf{N_{buf}}$	**R (%)**	$\mathbf{C_{max}}$	$\mathbf{N_{buf}}$	**R (%)**
3_5_10	42.70	2.00	12.50	55.50	4.40	29.20
3_7_10	57.90	1.10	6.80	71.20	7.80	38.60
3_10_10	65.20	1.20	4.60	87.00	10.90	32.60
5_5_10	39.40	0.20	1.20	51.20	4.90	32.90
5_7_10	53.90	0.70	3.10	71.50	7.50	35.90
5_10_10	60.60	0.60	1.70	78.60	8.10	25.50
7_5_10	31.00	0.90	5.40	42.60	5.20	31.10
7_7_10	44.70	0.50	2.30	61.44	5.60	29.40
7_10_10	70.40	0.50	1.50	99.00	8.20	26.90

(b) Maximum delay 5 time units

Instance	CP Optimizer			CSOP+PP		
	$\mathbf{C_{max}}$	$\mathbf{N_{buf}}$	**R (%)**	$\mathbf{C_{max}}$	$\mathbf{N_{buf}}$	**R (%)**
3_5_50	201.10	2.20	12.90	232.50	6.60	39.80
3_7_50	267.20	2.20	7.20	317.30	8.80	34.00
3_10_50	353.30	3.50	8.60	450.00	12.60	35.30
5_5_50	184.30	1.20	2.70	252.20	5.70	27.70
5_7_50	253.50	1.10	1.60	340.20	8.30	31.80
5_10_50	363.00	0.90	0.30	467.10	11.60	31.70
7_5_50	177.10	1.10	2.50	237.80	5.80	27.90
7_7_50	252.00	0.40	0.40	380.20	6.20	29.10
7_10_50	363.90	1.00	1.00	512.60	10.40	28.50

(c) Maximum delay 10 time units

Instance	CP Optimizer			CSOP+PP		
	$\mathbf{C_{max}}$	$\mathbf{N_{buf}}$	**R (%)**	$\mathbf{C_{max}}$	$\mathbf{N_{buf}}$	**R (%)**
3_5_100	389.00	2.20	12.70	459.00	7.40	41.00
3_7_100	561.40	4.40	16.70	680.90	9.70	38.50
3_10_100	768.70	2.90	4.40	948.90	12.70	34.60
5_5_100	349.00	1.10	2.70	438.90	6.40	34.20
5_7_100	495.00	0.80	0.40	643.90	8.30	33.90
5_10_100	698.20	0.70	1.00	932.10	12.40	35.00
7_5_100	391.40	1.00	2.30	500.80	6.30	37.10
7_7_100	516.00	0.40	2.00	695.80	6.70	27.80
7_10_100	744.80	0.70	0.50	1043.20	10.20	30.00

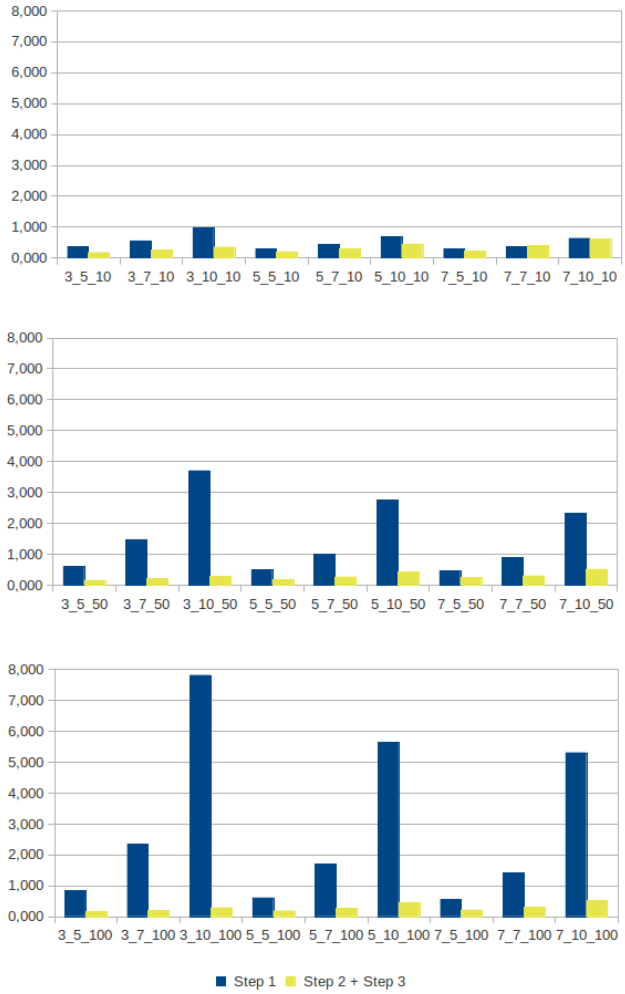

Figure 6.2: Computational time to calculate the step 1 and the step 2 + step 3

It can be seen that for CSOP+PP the greater p_i, the greater robustness values since the buffers generated could have bigger sizes, e.g., the instances with $m = 5$ and $v_{max} = 10$ increased their robustness degree obtaining an average of 25.5% for $p_i = 10$; 31.7% for $p_i = 50$; and 35% for $p_i = 100$.

In the figure 6.2 are presented the computational times to calculate the step 1, CSOP in this case, and the computational time to calculate the post procedure step 2 and step 3. The axis y represent the time in second and the axis x represent each set of instances. In the graphics can be observed that the computational time of the step 1 is dependent of the durations of task and the number task by job. But for the post procedures the computational time is only dependent of the number of task

because the computational time for the post procedures remains stable in the three graphics.

Table 6.5 presents how large delays in the incidences affect the schedules. As d increases, the average of incidences absorbed are reduced, reaching the case that the CP Optimizer obtained an average robustness about 0% for most instances with $p_i = 100$. Even, CP Optimizer was unable to absorb any incidence in instances of $<$ 7_10_100 $>$, whereas the CSOP+PP obtained an average of 6.9% the number of incidences absorbed for large delays.

Table 6.5: Avg. Robustness with large incidences ($p_i = [1, 100]$)

Instance	**CP Optimizer**			**CSOP+PP**		
	d [1,20]	**d [1,50]**	**d [1,100]**	**d [1,20]**	**d [1,50]**	**d [1,100]**
3_5_100	9.90	6.10	3.10	35.20	20.80	11.90
3_7_100	8.90	6.40	3.30	34.90	22.40	12.00
3_10_100	3.80	1.90	1.30	29.10	17.10	9.50
5_5_100	1.90	1.40	0.50	26.90	12.00	8.30
5_7_100	0.20	0.00	0.00	28.60	14.80	8.50
5_10_100	1.00	0.50	0.30	26.70	16.80	10.80
7_5_100	1.90	0.50	0.40	25.00	11.30	6.70
7_7_100	0.80	0.10	0.10	20.10	10.60	5.60
7_10_100	0.00	0.00	0.00	23.50	13.80	6.90

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Job-shop scheduling problem is a representation of a hypothetical problem of scheduling, the extension with operators give more difficulty to the problem but $JSO(n, p)$ can be closer to the reality. Most of the job-shop solving techniques try to find the optimality of the problem for minimizing the makespan, minimizing tardiness, minimizing flow-time, etc. But with the developed techniques, the solution also try to improve the robustness of the solution. A robust solution can resist small modifications and remains being correct. The developed technique tries to obtain a solution that take into account both makespan and robustness.

The JSP with operators is really hard to solve efficiently. At the beginning of this work, the objective was to obtain a solution to the problem with operators by minimizing the makespan, then to obtain a robust solution from the previous optimized solution. A technique for solving the problem was developed but the results were not so goods. For this reason, some techniques were developed to improve the solver and reduce the difficulty of the problem but the problem remained intractable. In this way, the idea of solving the problem in several parts appeared.

The technique is composed into three steps. The first step consists in solving a problem without take into account the operators, this step can be done for any job-shop problem solving technique. In this case in the step one the problem has been modelled as a CSP, the output of this step is a correct solution of the problem minimizing the makespan. The advantage if the technique is separated into three steps is that the first step can be only calculated one time. This can be an advantage if the plan of some problems is going to use periodically. The first step can be calculated only one time where the makespan is minimized and in the second step can be controlled the trade-off between robustness and optimality desired by the client. In the second step the problem is modified to take into account operators and the solution is obtained desiring to improve the robustness. In the third step the robustness is increased because the number of buffers is also increased. The computational time cost to execute the second and third step is low comparing the

time needed to execute the first step. For this reason this steps can be calculated repeatedly without causing a very high computational cost and the most expensive step is calculated only once.

7.2 Future work

The previous section shows techniques to obtain solutions that take into account robustness to absorb incidences. Related to this robustness concept, a study of the different robustness measures could be carried out.

These robust solutions have the property to absorb incidences but losing optimality against optimal solution. However a good point to take into account is to analyze the trade-off between optimality and robustness. Therefore, a good line of future work is, given an optimal solution, to provide different robust solutions according to the optimality level that the client is willing to lose from the optimal solution given by the first step.

The technique developed in this work is specific for this problem ($JSO(n, p)$) because between the step one and two, there is a change in the problem due to the introduction of a new resource: the operators. This idea of this Master Thesis about simplifying the problem to model and solve it easier and more efficient can be used in other problems in the real world. Therefore, another future work could be designing a general technique to apply it for more schedule problems.

7.3 Related publications

In this section, it is shown a list of published publications to conferences.

- Carlos Mencía, María R. Sierra, Miguel A. Salido, Joan Escamilla and Ramiro Varela.
  Combining Global Pruning Rules with Depth-First Search for the Job Shop Scheduling Problem with Operators.
  In 19th International Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion, RCRA 2012.

- Joan Escamilla, Mario Rodriguez-Molins, Miguel A. Salido, María R. Sierra, Carlos Mencía and Federico Barber.
  Robust Solution to Job-Shop Scheduling Problems with Operators.
  In 24th International Conference on Tools with Artificial Intelligence, ICTAI 2102 (CORE B).

Bibliography

[1] A. Agnetis, M. Flamini, G. Nicosia, and A. Pacifici. A job-shop problem with one additional resource type. *Journal of Scheduling*, 14(3):225–237, 2011.

[2] E. Balas. Machine sequencing via disjunctive graphs: an implicit enumeration algorithm. *Operations research*, pages 941–957, 1969.

[3] I. Barba, C. Del Valle, and D. Borrego. A constraint-based job-shop scheduling model for software development planning. *Actas de los Talleres de las Jornadas de Ingeniería del Software y Bases de Datos*, 3(1), 2009.

[4] J.C. Beck. *A schema for constraint relaxation with instantiations for partial constraint satisfaction and schedule optimization*. PhD thesis, University of Toronto, 1994.

[5] J.R. Bitner and E.M. Reingold. Backtrack programming techniques. *Communications of the ACM*, 18(11):651–656, 1975.

[6] EA Boyd and R. Burlingame. A parallel algorithm for solving difficult job-shop scheduling problems. *Operations Research Working Paper, Department of Industrial Engineering, Texas A&M University, College Station, Texas*, pages 77843–3131, 1996.

[7] APG Brown and ZA Lomnicki. Some applications of the" branch-and-bound" algorithm to the machine scheduling problem. *Operations Research*, pages 173–186, 1966.

[8] J. Escamilla, M. Rodriguez-Molins, M. A. Salido, M. R. Sierra, C. Mencía, and F. Barber. Robust solution to job-shop scheduling problems with operators. In *24th IEEE International Conference on Tools with Artificial Intelligence, 2012.(ICTAI 2012)*. IEEE.

[9] D. Frost and R. Dechter. Dead-end driven learning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 294–294. JOHN WILEY & SONS LTD, 1994.

[10] J. Gaschig. Performance measurement and analysis of certain search algorithms. Technical report, DTIC Document, 1979.

[11] J. Gaschnig. A general backtrack algorithm that eliminates most redundant tests. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 457–457, 1977.

[12] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5):533–549, 1986.

[13] F. Glover. Tabu search-part ii. *ORSA Journal on computing*, 2(1):4–32, 1990.

[14] F. Glover et al. Tabu search-part i. *ORSA Journal on computing*, 1(3):190–206, 1989.

[15] J.F. Gonçalves, J.J. de Magalhães Mendes, and M.G.C. Resende. A hybrid genetic algorithm for the job shop scheduling problem. *European journal of operational research*, 167(1):77–95, 2005.

[16] R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3):263–313, 1980.

[17] E. Hebrard, B. Hnich, and T. Walsh. Super csps. Technical report, Technical report, 2003.

[18] J.H. Holland. *Adaptation in natural and artificial systems*. Number 53. University of Michigan press, 1975.

[19] E. Ignall and L. Schrage. Application of the branch and bound technique to some flow-shop scheduling problems. *Operations Research*, pages 400–412, 1965.

[20] A.S. Jain and S. Meeran. Job-shop scheduling using neural networks. *International Journal of Production Research*, 36(5):1249–1272, 1998.

[21] E. Jen. Stable or robust? whats the difference? *Complexity*, 8(3):12–18, 2003.

[22] H. Kitano. Towards a theory of biological robustness. *Molecular Systems Biology*, 3(137), 2007.

[23] S. Kobayashi, I. Ono, M. Yamamura, et al. An efficient genetic algorithm for job shop scheduling problems. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 506–511. Morgan Kaufmann, San Francisco. CA, 1995.

[24] M. Laguna, J.W. Barnes, and F.W. Glover. Tabu search methods for a single machine scheduling problem. *Journal of Intelligent Manufacturing*, 2(2):63–73, 1991.

[25] M. Laguna and F. Glover. Integrating target analysis and tabu search for improved scheduling systems. *Expert Systems with Applications*, 6(3):287–297, 1993.

[26] C. Lecoutre and S. Tabary. Abscon 112 toward more robustness. 2008.

[27] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[28] C. Mencía, M. R. Sierra, M. A. Salido, J. Escamilla, and R. Varela. Combining global pruning rules with depth-first search for the job shop scheduling problem with operators. In *19th RCRA International Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion*, 2012.

[29] R. Mencía, M. R. Sierra, C. Mencía, and R. Varela. Genetic algorithm for job-shop scheduling with operators. In *Proceedings of IWINAC 2011(2). LNCS 6687*, pages 305–314. Springer, 2011.

[30] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information sciences*, 7:95–132, 1974.

[31] W. Nuijten and C. Le Pape. Constraint-based job shop scheduling with iilog scheduler. *Journal of Heuristics*, 3(4):271–286, 1998.

[32] F. Pezzella and E. Merelli. A tabu search method guided by shifting bottleneck for the job shop scheduling problem. *European Journal of Operational Research*, 120(2):297–310, 2000.

[33] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational intelligence*, 9(3):268–299, 1993.

[34] M.G.C. Resende. A grasp for job shop scheduling. In *INFORMS Spring Meeting*, 1997.

[35] A. Rizk, G. Batt, F. Fages, and S. Solima. A general computational method for robustness analysis with applications to synthetic gene networks. *Bioinformatics*, 25(12):168–179, 2009.

[36] O. Roussel and C. Lecoutre. Xml representation of constraint networks: Format xcsp 2.1. 2009.

[37] Z. Ruttkay. Constraint satisfaction-a survey. *CWI Quarterly*, 11(2-3):163–214, 1998.

[38] N. Sadeh and M.S. Fox. Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence*, 86(1):1–41, 1996.

[39] M. Sierra, C. Mencía, and R. Varela. Optimally scheduling a job-shop with operators and total flow time minimization. In *Advances in Artificial Intelligence: 14th Conf. of the Spanish Association for Artificial Intelligence, Caepia 2011*, LNAI 7023, pages 193–202. Springer, 2011.

[40] M. R. Sierra, C. Mencía, and R. Varela. Searching for optimal schedules to the job-shop problem with operators. *Technical report. Computing Technologies Group. University of Oviedo*, 2011.

[41] E. Szathmáry. A robust approach. *Nature*, 439(7072):19–20, 2006.