Journal of Scheduling, 13(1): 17-38, 2010
DOI: 10.1007/s10951-009-0106-z

An improved constraint satisfaction adaptive neural network for job-shop scheduling

Shengxiang Yang · Dingwei Wang · Tianyou Chai · Graham Kendall

Received: 30 April 2007 / Accepted: 6 February 2009

Abstract The job-shop scheduling problem is one of the most difficult problems in scheduling. This paper presents an improved constraint satisfaction adaptive neural network for job-shop scheduling problems. The neural network is constructed based on constraint conditions of a job-shop scheduling problem. Its structure and neuron connections can change adaptively according to the real-time constraint satisfaction situations that arise during the solving process. Several heuristics are also integrated within the neural network to enhance its convergence, accelerate its convergence, and improve the quality of the solutions produced. An experimental study based on a set of benchmark job-shop scheduling problems shows that the improved constraint satisfaction adaptive neural network outperforms the original constraint satisfaction adaptive neural network in terms of computational time and the quality of schedules it produces. The neural network approach is also experimentally validated to outperform three classical heuristic algorithms that are widely used as the basis of many state-of-the-art scheduling systems. Hence, it may also be used to construct advanced job-shop scheduling systems.

Keywords Job-shop scheduling · Constraint satisfaction adaptive neural network · Heuristics · Active schedule · Non-delay schedule · Priority rule · Computational complexity

S. Yang
Department of Computer Science, University of Leicester
University Road, Leicester LE1 7RH, UK
E-mail: s.yang@mcs.le.ac.uk

D. Wang · T. Chai
Key Laboratory of Integrated Automation of Process Industry (Northeastern University), Ministry of Education
Northeastern University, Shenyang 110004, China

D. Wang
E-mail: dwwang@mail.neu.edu.cn

T. Chai
E-mail: tychai@mail.neu.edu.cn

G. Kendall
School of Computer Science, University of Nottingham
Jubilee Campus, Wollaton Road, Nottingham NG8 1BB, UK
E-mail: gxk@cs.nott.ac.uk

1 Introduction

The job-shop scheduling problem (JSSP) is one of the most difficult problems in scheduling. It aims to allocate a number of machines over time to perform a set of jobs with certain constraint conditions in order to optimize a certain criterion, e.g., minimizing the makespan (Baker 1974). The JSSP has been much studied in the academic literature due to its importance as a typical combinatorial optimization problem and its potential expansion to a wide range of industrial problems. Traditionally, there are three kinds of methods to solve JSSPs: priority rules, combinatorial optimization, and constraints analysis (Dubois et al. 1995). The first category has the merit of being computationally efficient and easy to implement on real cases, but there is no guarantee with respect to the quality of the solutions produced. Optimization methods are much more rigorous but are not tractable when the problem size is large and the optimal solution is required (Bellman et al. 1982). The third category, originated from Erschler *et al.* (1976), looks for a set of feasible solutions that meet several technical constraints from which the user may choose the final solution.

It has been demonstrated (Garey et al. 1976) that job-shop scheduling is usually an NP-complete prob-

lem. Because of the NP-complete characteristics of the JSSP, it is usually very hard to find its optimal solution. Fortunately, an optimal solution in the mathematical sense is not always necessary in practice. Hence, researchers have turned to search for near-optimal solutions to JSSPs, utilizing many different heuristic algorithms (French 1982) and the near-optimal solutions that are produced usually meet the requirements of practical problems. Several knowledge-based scheduling systems have been presented (Fox and Zweben 1993; Hentenryck 1989), which are much more general than traditional methods because they systematically use constraints, implement heuristic knowledge, and represent a framework for stating and solving combinatorial optimization problems.

Researchers have also investigated artificial intelligence methods for JSSPs. As one line of research on artificial intelligent methods for JSSPs, genetic algorithms (GAs) have been intensively investigated in the last decade (Bierwirth and Mattfeld 1999; Cheng et al. 1996, 1999; Fang et al. 1993; Hart et al. 2005). Nowadays, several state-of-the-art JSSP scheduling systems are based on GAs (Vázquez and Whitley 2000a provides a comparison of several GAs on JSSPs). As reviewed by Hart et al. (2005), many of the state-of-the-art GA based approaches for JSSPs make use of heuristics, such as the classical Giffler and Thompson's algorithms (Giffler and Thompson 1960), to generate schedules and/or guide genetic operators. Examples include the Heuristically-guided GA (HGA) (Hart and Ross 1998), the Order Based Giffler and Thompson (OBGT) algorithm (Vázquez and Whitley 2000b), and the GA with Time Horizon Exchange (THX) operators (Lin et al. 1997).

Another line of research on intelligent methods for JSSPs is to investigate artificial neural network based scheduling systems for JSSPs (Luh et al. 2000; Akyol and Bayhan 2007). Foo and Takefuji (1988a, 1988b) first used a neural network to solve JSSPs. Thereafter, several neural network architectures have been devised (Foo et al. 1994; Willems 1994; Willems and Brangts 1995; Yu 1997). Willems (1994) first proposed a constraint satisfaction neural network for traditional JSSPs without free operations[1]. Willems's neural network was extended in (Yu 1997; Yu and Liang 2001) by adding a job constraint block to deal with free operations and introduced the gradient optimization mechanism that can be combined with the neural network for JSSPs. The above mentioned neural networks are basically static, where the connection weights and biases of neurons must be first prescribed according to the constraints of a particular JSSP and then remain fixed during the solving process.

In (Yang and Wang 2000), a constraint satisfaction adaptive neural network (CSANN) was proposed for JSSPs. CSANN consists of a sequence constraint block that is constructed according to the sequence constraints of a JSSP and a resource constraint block that is constructed according to the resource constraints of the JSSP. CSANN works by iteratively removing the violation of the mapped constraints and can adaptively adjust the connection weights and biases of neurons according to the actual constraint violations during the solving process. In (Yang and Wang 2000, 2001), several heuristic algorithms were also proposed to improve the performance of CSANN and the quality of produced solutions. Recently, an improved model of CSANN (called *CSANN-II*), which simplifies the resource constraint block of CSANN, was proposed in (Yang 2005) with some preliminary experiments showing promising results. In CSANN-II, the resource constraint block is adaptively constructed from the actual resource constraint satisfaction situation during the solving process via a simple sorting algorithm. CSANN-II has reduced resource constraint neurons in the resource constraint block over the original CSANN model and hence has reduced computational complexity. In (Yang 2005), some heuristics were also integrated to improve the performance of CSANN-II. For example, an adaptive pre-processing method was proposed to determine the expected makespan value used for CSANNs and a schedule improvement method was applied to get an active schedule (the definition of an active schedule will be given later on in Sect. 2.2) from a feasible schedule produced by CSANNs.

This paper further investigates CSANN-II for JSSPs and provides a compilation and extended discussion of work reported previously in (Yang and Wang 2000, 2001; Yang 2005). Based on an extended set of benchmark JSSPs, experiments are carried out in this paper to verify the computational complexity comparisons of CSANN-II over CSANN reported previously in (Yang 2005), and new experiments are provided to investigate the impact of the initial expected makespan provided to CSANNs on the makespan of schedules produced by CSANNs. In this paper, the performance of CSANN-II is also experimentally validated in comparison with CSANN and three classical heuristics algorithms based on Giffler and Thompson's methods (Giffler and Thompson 1960) for JSSPs. The three classical heuristic algorithms studied in this paper are widely used as the fundamental tools for constructing advanced JSSP systems (Hart et al. 2005). Just like the three classical heuristic algorithms, CSANN-II can act as a

[1] A free operation of a job is one that has no sequence constraints with other operations of the job.

good fundamental tool for constructing advanced JSSP scheduling systems given its better performance.

The rest of this paper is organized as follows. The next section presents the basic concepts of classical job-shop scheduling, the mathematical formulation and the classical Giffler and Thompson methods for JSSPs (Giffler and Thompson 1960). Sect. 3 presents the original CSANN model proposed in (Yang and Wang 2000, 2001) for JSSPs. In Sect. 4, the improved model of CSANN-II is described in detail together with the heuristic algorithms that can be combined with CSANN and CSANN-II for a better performance. Sect. 5 presents the computer simulation study based on a set of benchmark JSSPs to investigate the computational complexity of CSANN-II over CSANN and show the performance of the combined approach of CSANN-II and heuristic algorithms for JSSPs. Finally, Sect. 6 concludes this paper with discussions on relevant future work.

2 The job-shop scheduling problem

2.1 Formulation of the JSSP

Traditionally, the JSSP can be stated as follows (Conway et al. 1967): Given n jobs to be processed on m machines in a prescribed order under certain restrictive assumptions, the objective is to optimally arrange the processing order and the start times of operations to optimize certain criteria. Generally speaking, for JSSPs there are two types of constraints: *sequence constraints* and *resource constraints*. The first type states that one operation of a job must finish before another operation starts. The second type states that no more than one job can be handled on a machine at the same time. Job-shop scheduling can be viewed as an optimization problem, bounded by both sequence and resource constraints. Traditionally for a JSSP, each job may consist of a different number of operations, which are subject to some precedence restrictions. Usually, the processing order of each job by all machines and the processing time of each operation are known and fixed. Operations cannot be interrupted once started (i.e., non-preemptive). This kind of scheduling is called deterministic and static. In this paper, we focus on the deterministic and static JSSP.

For the convenience of formulating the JSSP, some notations are defined as follows. Let $J = \{J_1, \ldots, J_n\}$ and $M = \{M_1, \ldots, M_m\}$ denote the job set and the machine set, respectively, where n and m are the numbers of jobs and machines, respectively. Let μ_i be the number of operations for job i. O_{ikq}, represents operation k of job i to be processed on machine q, T_{ikq} and P_{ikq} represent the start time and processing time of O_{ikq}, respectively, $T_{i\mu_i q}$ and $P_{i\mu_i q}$ represent the start time and processing time of the last operation of job i, respectively[2]. Denote r_i and d_i as the release date (the earliest start time) and deadline (the latest end time) of job i. Let S_i denote the set of operation pairs $[O_{ikp}, O_{ilq}]$ of job i, where O_{ikp} must be processed before O_{ilq}. Let R_q be the set of operations O_{ikq} to be processed on machine q.

Taking minimizing the makespan as the optimization criterion, the mathematical formulation of the JSSP considered in this paper can be presented as follows:

$$Minimize \;\; E = max_{i \in J}(T_{i\mu_i q} + P_{i\mu_i q}),$$

subject to

$$T_{ilq} - T_{ikp} \geq P_{ikp}, [O_{ikp}, O_{ilq}] \in S_i, \quad k, l \in \{1, \cdots, \mu_i\}, \; i \in J, \tag{1}$$

$$T_{jlq} - T_{ikq} \geq P_{ikq} \; or \; T_{ikq} - T_{jlq} \geq P_{jlq}, \quad O_{ikq}, O_{jlq} \in R_q, \; i, j \in J, \; q \in M, \tag{2}$$

$$r_i \leq T_{ijq} \leq d_i - P_{ijq}, \; i \in J, j \in \{1, \ldots, \mu_i\}, \; q \in M, \tag{3}$$

where the cost function E is the completion time of the latest operation. Minimizing the cost function means minimizing the makespan. Equation (1) represents the sequence constraint; (2) represents resource constraints in a disjunctive format; (3) represents the release date and deadline constraints. Note that the fact that jobs can have deadlines suggests that problems could be oversubscribed (i.e., there may be more jobs to do than available resources). In this paper, oversubscribed problems will not be considered. This paper will mainly concern JSSPs with no (tight) deadline constraints for jobs. In this paper, we also assume that a job is not allowed to be processed by a machine more than once.

Using the standard notation for scheduling systems of Graham et al. (1979), the above JSSP can be denoted by $Jm|n|C_{max}$, where J means job-shop scheduling and C_{max} means minimizing the maximal completion time, i.e., the makespan. For a $Jm|n|C_{max}$, there are at most $n(m-1)$ sequence constraint inequalities of (1), at most $mn(n-1)$ resource constraint inequalities of (2), at most mn start time constraint inequalities of (3), resulting in a total number of at most $n(mn+m-1)$ constraint inequalities. There are also at most mn number of variables T_{ikq}. The objective of job-shop scheduling

[2] In this paper, we deal with JSSPs where each operation of a job must be processed by one fixed machine. Hence, in the notation of O_{ikq} and other variables with three subscripts, the third subscript is determined by the first two subscripts. However, for the convenience of describing resource constraints and RC-block, the third subscript is used.

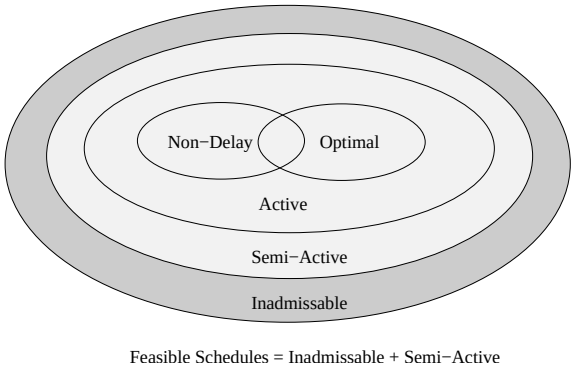

Fig. 1 Relationships of different kinds of feasible schedules.

is to find values for these variables so that they satisfy all the constraint inequalities while minimizing the makespan.

2.2 Classification of schedules to JSSPs

Given a JSSP, any feasible solution to the above formulation is called a *feasible schedule*. Given a feasible schedule for a JSSP, if an operation can be left-shifted, i.e., started earlier, without altering the processing sequences, such a left-shift is called a *local left-shift*. If a left-shift of an operation alters the processing sequences but does not delay any other operations, it is called a *global left-shift*. Based on the concept of local and global left-shift, feasible schedules for a JSSP can be classified into four types: *inadmissible*, *semi-active*, *active* and *non-delay*. Inadmissible schedules are those that contain excess idle time and can be improved by local left-shift(s). Semi-active schedules are those that allow no local left-shift. Active schedules are those that allow neither local left-shift nor global left-shift. Non-delay schedules are those schedules in which no machine is kept idle when it could start processing some operation.

The relationships between the different kinds of feasible schedules are illustrated in Fig. 1. Non-delay schedules are necessarily active and hence also necessarily semi-active. An optimal schedule with respect to makespan is guaranteed to be an active one but not necessarily a non-delay one (Baker 1974). However, there is strong empirical evidence that non-delay schedules show a better mean solution quality than active ones. Nevertheless, scheduling algorithms typically search the space of active schedules in order to guarantee that the optimum is taken into consideration.

Algorithm 1 The GT-Act algorithm

1: Calculate the set D of all operations that can be scheduled next. If D is empty, stop.
2: Find an operation $O^* \in D$ (with ties broken randomly) that has the minimum earliest (possible) completion time. That is, $O^* := \arg\min\{EC(O)|O \in D\}$. Let M^* denote the machine that processes O^*.
3: Construct the *conflict set* C that contains those unscheduled operations in D that are processed on M^* and will overlap with O^* in time if they are scheduled at their earliest start times. That is, $C := \{O \in D \mid O \text{ on } M^*, ES(O) < EC(O^*)\}$.
4: Select an operation $O \in C$ *randomly* and schedule it on M^* with its completion time set to $EC(O)$.
5: Delete O from D and return to step 1.

Algorithm 2 The GT-ND algorithm

1: Calculate the set D of all operations that can be scheduled next. If D is empty, stop
2: Find an operation $O^* \in D$ (with ties broken randomly) that has the minimum earliest (possible) start time. That is, $O^* := \arg\min\{ES(O)|O \in D\}$. Let M^* denote the machine that processes O^*.
3: Construct the *conflict set* C that contains those unscheduled operations in D that are processed on M^* and will overlap with O^* in time if they are scheduled at their earliest start times. That is, $C := \{O \in D \mid O \text{ on } M^*, ES(O) < EC(O^*)\}$.
4: Select an operation $O \in C$ randomly and schedule it on M^* with its start time equal to $ES(O)$.
5: Delete O from D and return to step 1.

2.3 Giffler and Thompson heuristics for JSSPs

Giffler and Thompson (1960) first proposed a systematic method, denoted *GT-Act* in this paper, to generate any active schedules for JSSPs as described below. Let $ES(O)$ and $EC(O)$ denote the earliest (possible) start time and earliest (possible) completion time of an operation O respectively. Let D be a set of all operations that are not scheduled yet and can be scheduled next. Initially, D consists of the first operations of jobs. An active schedule is generated by iteratively updating D and selecting one operation from D until all operations are scheduled (i.e., D is empty) as shown in Algorithm 1.

Giffler and Thompson (1960) also developed an algorithm, henceforth denoted *GT-ND*, to generate any non-delay schedules for JSSPs iteratively as shown in Algorithm 2. GT-ND differs from GT-Act only in that, in step 2, GT-ND selects from D the operation with the minimum earliest (possible) start time to be O^* in order to decide the machine M^* to schedule an operation on.

In step 4 of the above GT-Act and GT-ND algorithms, an operation to be selected from the conflict set C has an important effect on the final active and non-delay schedule respectively. Researchers have developed a large number of heuristic priority rules to be

Table 1 A list of priority rules for dispatching an operation for JSSPs

Rule	Description
SPT	Select an operation with the shortest processing time
LPT	Select an operation with the longest processing time
MWR	Select an operation for the job with the most total remaining processing time
LWR	Select an operation for the job with the least total remaining processing time
MOR	Select an operation for the job with the most number of operations remaining
LOR	Select an operation for the job with the least number of operations remaining

Algorithm 3 The GT-Rule algorithm

1: Calculate the set D of all operations that can be scheduled next. If D is empty, stop.
2: Find an operation $O^* \in D$ (with ties broken randomly) that has the minimum earliest (possible) completion time. That is, $O^* := \arg\min\{EC(O)|O \in D\}$. Let M^* denote the machine that processes O^*.
3: Construct the *conflict set* C that contains those unscheduled operations in D that are processed on M^* and will overlap with O^* in time if they are scheduled at their earliest start times. That is, $C := \{O \in D \mid O \text{ on } M^*, ES(O) < EC(O^*)\}$.
4: Select an operation $O \in C$ according to a *priority rule* randomly selected from a set of priority rules and schedule it on M^* with its completion time equal to $EC(O)$.
5: Delete O from D and return to step 1.

used in the Giffler and Thompson algorithms to select an operation from the conflict set C to be scheduled next. An extensive summary and discussion on heuristic priority rules can be found in (Blackstone et al. 1982; Haupt 1989; Graves 1981). Table 1 lists some priority rules commonly used for scheduling an operation for JSSPs in practice.

It is also quite common that the priority rules can be combined into the Giffler and Thompson algorithm: when dispatching an operation, one priority rule is first randomly selected from a pre-defined set of priority rules and then is applied to select an operation. This hybrid method is denoted *GT-Rule* in this paper and is described in Algorithm 3.

The GT-Act, GT-ND, and GT-Rule algorithms have become the basis for many state-of-the-art hybrid scheduling systems for JSSPs (Hart et al. 2005). For example, in Hart and Ross's HGA (Hart and Ross 1998), each gene in a chromosome represents a pair of values (*method*, *heuristic*), where *method* denotes either GT-Act or GT-ND should be used at each iteration of a scheduling algorithm to calculate a set of schedulable operations, and *heuristic* represents the priority dispatch rule to be used to select an operation from that set. The priority dispatch rules used in HGA are similar to those shown in Table 1.

In this paper, the GT-Act, GT-ND, and GT-Rule algorithms will be used as peer algorithms for comparing

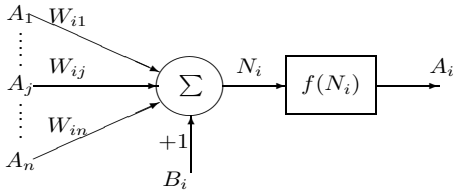

Fig. 2 General neuron model.

the performance of CSANN and CSANN-II for JSSPs. The GT-Rule algorithm studied in this paper uses the six rules in Table 1 as the set of priority rules. The following two sections will describe the details of the original CSANN model and the improved CSANN-II model, respectively.

3 The original CSANN model for JSSPs

Usually, a neural unit (or neuron), say neuron i, in a neural network consists of a linear summator and a nonlinear activation function $f(\cdot)$, which are serialized (Haykin 1999) as below:

$$A_i = f(N_i) = f(\sum_{j=1}^{n}(W_{ij} \times A_j) + B_i), \tag{4}$$

where the summator sums a bias B_i and received activations $A_j (j = 1, \ldots, n)$ from connected neurons with connection weight W_{ij} from neuron j to neuron i. The output of the summator is the net input N_i to neuron i, which is then passed to the activation function $f(\cdot)$ to obtain the activation A_i. Figure 2 shows the model of a general neuron in a neural network.

3.1 Neurons of CSANN

Based on the general neuron model, CSANN contains three kinds of neurons: *ST-neurons*, *SC-neurons* and *RC-neurons*. Each ST-neuron represents an operation with the activation representing the start time of the operation. Each SC-neuron or RC-neuron represents whether a relevant sequence constraint or resource constraint is satisfied, respectively.

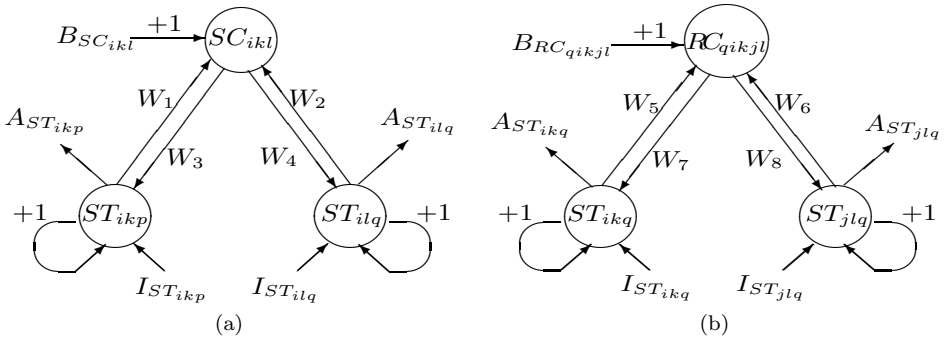

Fig. 3 (a) An SC-block unit SCB_{ikl} and (b) a RC-block unit RCB_{qikjl}.

The net input and activation functions of an ST-neuron, ST_i, are defined as:

$$\begin{aligned} N_{ST_i}(t+1) &= \textstyle\sum_j (W_{SC_j \to ST_i} \times A_{SC_j}(t)) \\ &\quad + \textstyle\sum_k (W_{RC_k \to ST_i} \times A_{RC_k}(t)) + A_{ST_i}(t) \end{aligned} \tag{5}$$

$$A_{ST_i}(t+1) = \begin{cases} r_i, & N_{ST_i}(t+1) < r_i, \\ N_{ST_i}(t+1), & r_i \le N_{ST_i}(t+1) \le d_i - P_{ST_i}, \\ d_i - P_{ST_i}, & N_{ST_i}(t+1) > d_i - P_{ST_i}, \end{cases} \tag{6}$$

where in (5) the net input of ST_i is summed from three parts. The first and second parts come from the activations of ST_i-relevant SC-neurons SC_j weighted by $W_{SC_j \to ST_i}$ and the activations of ST_i-relevant RC-neurons RC_k weighted by $W_{RC_k \to ST_i}$, which implement feedback adjustments due to sequence and resource constraint violations, respectively. The third part comes from the previous activation of neuron ST_i itself. The activation function in (6) is a linear-segmented one, where r_i and d_i are the release and deadline of job i to which the operation, corresponding to ST_i, belongs. P_{ST_i} is the processing time of the operation. This activation function implements the release and deadline constraints described by (3).

The net input and activation functions of an SC-neuron SC_i or a RC-neuron RC_i have the same definition as shown below:

$$\begin{aligned} N_{C_i}(t+1) &= W_{ST_1 \to C_i} \times A_{ST_1}(t) \\ &\quad + W_{ST_2 \to C_i} \times A_{ST_2}(t) + B_{C_i}, \end{aligned} \tag{7}$$

$$A_{C_i}(t+1) = \begin{cases} 0, & N_{C_i}(t+1) \ge 0, \\ -N_{C_i}(t+1), & N_{C_i}(t+1) < 0, \end{cases} \tag{8}$$

where C_i represents SC_i or RC_i and B_{C_i} is the bias, which equals the processing time of an operation corresponding to ST_1 or ST_2 (more details are explained in Sect. 3.2). The ST-neurons, ST_1 and ST_2, represent two operations of the same job for an SC-neuron, or two operations sharing the same machine for a RC-neuron. The activation function is linear-segmented. When the activation of an SC-neuron or RC-neuron is greater than zero, it means the relevant sequence constraint or resource constraint is violated and there will be feedback adjustments from C_i to relevant ST_1 and ST_2 with adaptive weights. In the following section, the connection weights and biases for SC-neurons and RC-neurons and feedback adjustments from an SC-neuron or RC-neuron to ST-neurons are explained.

3.2 Connection weights and biases

All neurons in CSANN are structured into two problem-specific constraint blocks: the sequence constraint block (*SC-block*) that deals with all sequence constraints of a given JSSP and the resource constraint block (*RC-block*) that deals with all resource constraints of a given JSSP. Each SC-block unit has two ST-neurons that represent two operations of a job and one SC-neuron that represents whether the relevant sequence constraint is satisfied, see Fig. 3(a). Similarly, each RC-block unit has two ST-neurons representing two operations on the same machine and one RC-neuron representing whether the relevant resource constraint is satisfied, see Fig. 3(b).

Figure 3(a) shows an example SC-block unit SCB_{ikl}. ST-neurons ST_{ikp} and ST_{ilq} represent two operations O_{ikp} and O_{ilq} of job i. Their activations $A_{ST_{ikp}}$ and $A_{ST_{ilq}}$ represent the start times T_{ikp} and T_{ilq}. The SC-neuron SC_{ikl} represents the sequence constraint of (1) between O_{ikp} and O_{ilq}, with $B_{SC_{ikl}}$ being its bias. $I_{ST_{ikp}}$ and $I_{ST_{ilq}}$ represent the initial value for T_{ikp} and T_{ilq}, which are taken as the initial net input to ST_{ikp} and ST_{ilq}, respectively. The weights and bias are valued as follows:

$$\begin{aligned} &W_1 = -1, W_2 = 1, W_3 = -W, \\ &W_4 = W, \;\; B_{SC_{ikl}} = -P_{ikp}, \end{aligned} \tag{9}$$

where W is a positive feedback adjustment factor (it is the same in other equations in this paper). At time t

during the run of CSANN, if the sequence constraint between O_{ikp} and O_{ilq} is satisfied, the activation $A_{SC_{ikl}}(t)$ of SC_{ikl} equals zero; otherwise, the activation of SC_{ikl} will be greater than zero and can be calculated by

$$\begin{aligned} A_{SC_{ikl}}(t+1) &= -N_{SC_{ikl}}(t+1) \\ &= A_{ST_{ikp}}(t) + P_{ikp} - A_{ST_{ilq}}(t) \\ &= T_{ikp}(t) + P_{ikp} - T_{ilq}(t). \end{aligned} \tag{10}$$

The feedback adjustments from SC_{ikl} to ST_{ikp} and ST_{ilq} are shown as follows:

$$\begin{aligned} A_{ST_{ikp}}(t+1) &= T_{ikp}(t+1) \\ &= T_{ikp}(t) - W \times A_{SC_{ikl}}(t), \end{aligned} \tag{11}$$

$$\begin{aligned} A_{ST_{ilq}}(t+1) &= T_{ilq}(t+1) \\ &= T_{ilq}(t) + W \times A_{SC_{ikl}}(t), \end{aligned} \tag{12}$$

where the feedback adjustments put backward the start time T_{ikp} of O_{ikp} and put forward the start time T_{ilq} of O_{ilq}. Thus, the sequence constraint violation between O_{ikp} and O_{ilq} may be solved.

Figure 3(b) shows an example RC-block unit RCB_{qikjl}, representing the resource constraint of (2) between O_{ikq} and O_{jlq} on machine q. At time t during the run of CSANN, the connection weights and bias are adaptively valued according to the following two cases.

Case 1: If $A_{ST_{ikq}}(t) \leq A_{ST_{jlq}}(t)$, i.e., $T_{ikq}(t) \leq T_{jlq}(t)$, (13) holds

$$\begin{aligned} &W_5 = -1, W_6 = 1, W_7 = -W, \\ &W_8 = W, \;\; B_{RC_{qikjl}} = -P_{ikq}, \end{aligned} \tag{13}$$

In this case, RCB_{qikjl} represents a sequence constraint described by the first disjunctive equation of (2). If violation exists, the activation of RC_{qikjl} and feedback adjustments from RC_{qikjl} to ST_{ikq} and ST_{jlq} are calculated by

$$\begin{aligned} A_{RC_{qikjl}}(t+1) &= A_{ST_{ikq}}(t) + P_{ikq} - A_{ST_{jlq}}(t) \\ &= T_{ikq}(t) + P_{ikq} - T_{jlq}(t), \end{aligned} \tag{14}$$

$$\begin{aligned} A_{ST_{ikq}}(t+1) &= T_{ikq}(t+1) \\ &= A_{ST_{ikq}}(t) + W_7 \times A_{RC_{qikjl}}(t) \\ &= T_{ikq}(t) - W \times A_{RC_{qikjl}}(t), \end{aligned} \tag{15}$$

$$\begin{aligned} A_{ST_{jlq}}(t+1) &= T_{jlq}(t+1) \\ &= A_{ST_{jlq}}(t) + W_8 \times A_{RC_{qikjl}}(t) \\ &= T_{jlq}(t) + W \times A_{RC_{qikjl}}(t). \end{aligned} \tag{16}$$

Case 2: If $A_{ST_{ikq}}(t) > A_{ST_{jlq}}(t)$, that is, $T_{ikq}(t) > T_{jlq}(t)$, (17) holds

$$\begin{aligned} &W_5 = 1, \quad W_6 = -1, W_7 = W, \\ &W_8 = -W, B_{RC_{qikjl}} = -P_{jlq}. \end{aligned} \tag{17}$$

In this case, RCB_{qikjl} represents a sequence constraint described by the second disjunctive equation of (2). If a violation exists, the activation of RC_{qikjl} and the feedback adjustments are calculated by

$$\begin{aligned} A_{RC_{qikjl}}(t+1) &= A_{ST_{jlq}}(t) + P_{jlq} - A_{ST_{ikq}}(t) \\ &= T_{jlq}(t) + P_{jlq} - T_{ikq}(t), \end{aligned} \tag{18}$$

$$\begin{aligned} A_{ST_{ikq}}(t+1) &= T_{ikq}(t+1) \\ &= A_{ST_{ikq}}(t) + W_7 \times A_{RC_{qikjl}}(t) \\ &= T_{ikq}(t) + W \times A_{RC_{qikjl}}(t), \end{aligned} \tag{19}$$

$$\begin{aligned} A_{ST_{jlq}}(t+1) &= T_{jlq}(t+1) \\ &= A_{ST_{jlq}}(t) + W_8 \times A_{RC_{qikjl}}(t) \\ &= T_{jlq}(t) - W \times A_{RC_{qikjl}}(t). \end{aligned} \tag{20}$$

3.3 Network complexity and running mechanism

The architecture of CSANN consists of two layers. The bottom layer consists of only ST-neurons. The top layer consists of SC-neurons and RC-neurons that are connected to ST-neurons at the bottom layer according to the exact sequence and resource constraints of a specific JSSP.

For a traditional JSSP with m machines and n jobs where each job goes through all machines once (i.e., $\mu_i = m$ for all $i \in J$) in a certain sequencing order, it requires mn ST-neurons representing the mn operations, $n(m-1)$ SC-neurons representing the $n(m-1)$ sequence constraints described by (1), and $mn(n-1)/2$ RC-neurons representing the $mn(n-1)/2$ resource constraints described by (2). In total, there are $n(0.5mn + 1.5m - 1)$ neurons for the whole CSANN, which is in the order of $O(mn^2)$.

Given a problem-specific CSANN, there are three running mechanisms to produce a schedule (Yang and Wang 2000). In the first mechanism, for each iteration, the activation of units is calculated in a fixed order: first calculating each ST-unit, then calculating each SC-unit, and finally calculating each RC-unit. This results in a deterministic unique schedule under the same initial conditions of ST-units. The second mechanism calculates the activation of units in a random order for each iteration, which results in non-deterministic schedules under the same initial conditions of ST-units. These two mechanisms are asynchronous. The third mechanism is synchronous, where the activation of units is calculated in a parallel manner. For each iteration, the activation of all units is calculated in a random or fixed order, but the newly calculated activation of a unit is not sent immediately to its connected units but stored until all

Algorithm 4 Produce a schedule by CSANN

1: Randomly initialize $T_{ikp}(0)$ for each operation O_{ikp}, and take it as the initial net input $I_{ST_{ikp}}$ to ST_{ikp}.
2: Run each SC-neuron SC_{ikl} of the SC-block: calculate its activation with (10). $A_{SC_{ikl}}(t) \neq 0$ means the violation of relevant sequence constraint, then adjust activations of related ST-neurons with (21) and (22) if Algorithm 5 is triggered, or with (11) and (12), otherwise.
3: Run each RC-neuron RC_{qikjl} of the RC-block: calculate its activation with (14) or (18). $A_{RC_{qikjl}}(t) \neq 0$ means the violation of relevant resource constraint, then adjust $A_{ST_{ikq}}(t+1)$ and $A_{ST_{jlq}}(t+1)$ with (23) and (24) if Algorithm 6 is triggered, or with (15) and (16) or (19) and (20), otherwise.
4: Repeat step 2 to 4 until all neurons become stable without changes, i.e., all sequence and resource constraints are satisfied and a feasible schedule is produced.
5: Produce a semi-active schedule from the feasible schedule produced by CSANN.

Algorithm 5 Swap two adjacent operations of a job

1: **if** $[O_{ikp}, O_{ilq}] \in S_i$ && $A_{ST_{ikp}}(t) > A_{ST_{ilq}}(t)$ (*i.e.*, $T_{ikp}(t) > T_{ilq}(t)$) **then**
2: exchange the order of O_{ikp} and O_{ilq} by exchanging their start times as follows:
$A_{ST_{ikp}}(t+1) := T_{ikp}(t+1) := T_{ilq}(t)$, (21)
$A_{ST_{ilq}}(t+1) := T_{ilq}(t+1) := T_{ikp}(t)$ (22)
3: **end if**

units have finished their calculations and stored their activations. In the next calculation cycle, the activation of a unit is calculated using the stored activations of the connected units.

In this paper, CSANN is run iteratively using the first asynchronous running mechanism. Each iteration first calculates each SC-block unit in the SC-block and then calculates each RC-block unit in the RC-block in a fixed order, e.g., starting from RC-block units corresponding to the first machine, to RC-block units corresponding to the second machine, and so on. This iteration continues until the activations of all SC-neurons and RC-neurons become zero. The final activations of ST-neurons form a feasible schedule to the given JSSP.

The procedure of running CSANN to produce a schedule is summarized in Algorithm 4. In order to enhance the performance of CSANN for JSSPs, several heuristic algorithms were developed in (Yang and Wang 2000, 2001). These heuristic algorithms, Algorithm 5 and Algorithm 6, are integrated in Algorithm 4. They are described in the following section.

3.4 Heuristic algorithms for CSANNs

3.4.1 Swap two adjacent operations of a job

During the running of CSANN, if two adjacent operations of the same job are placed in an order that violates the relevant JSSP sequence constraint, their start times are exchanged. Assuming $[O_{ikp}, O_{ilq}] \in S_i$, at time t during the running of CSANN, the heuristic algorithm works as shown in Algorithm 5.

This heuristic algorithm aims to accelerate the solving process. In fact, (21) and (22) form a more direct method of removing sequence constraint violations than the feedback adjustment scheme in CSANN. Hence, the adjustment time for removing sequence constraint violations is shortened and the solving process is speeded up.

Algorithm 6 Swap two adjacent operations on a machine

1: **if** $T_{qikjl}(t) \geq H$ (H is a prefixed threshold) **then**
2: swap the order of O_{ikq} and O_{jlq} on machine q:
$A_{ST_{ikq}}(t+1) := T_{ikq}(t+1) := T_{jlq}(t)$, (23)
$A_{ST_{jlq}}(t+1) := T_{jlq}(t+1) := T_{ikq}(t)$ (24)
3: $T_{qikjl}(t) := 0$
4: **end if**

3.4.2 Swap two adjacent operations on a machine

During the running of CSANN, due to conflicts resulting from sequence and resource constraint violation feedback adjustments, the phenomenon of *deadlock* may occur (Yang and Wang 2000). Deadlocks stop CSANNs from producing a feasible solution. A heuristic algorithm was proposed to break the deadlock (and hence make it possible to produce feasible schedules) by exchanging the order of two adjacent operations on the same machine via exchanging their start times under a certain condition.

The heuristic algorithm (i.e., Algorithm 6) works as follows. For each RC-block unit RCB_{qikjl}, a variable $T_{qikjl}(t)$ is defined to count, accumulated over iterations, the number of continuous and similar feedback adjustments from RC_{qikjl} to ST_{ikq} and ST_{jlq} due to the resource constraint violation between O_{ikq} and O_{jlq} on machine q. Two feedback adjustments are called *similar* if they have the same effect on ST_{ikq} and ST_{jlq}, e.g., both pushing T_{ikq} forward while pushing T_{jlq} backward. Whenever the resource constraint between O_{ikq} and O_{jlq} is satisfied or a different feedback adjustment occurs within RCB_{qikjl}, $T_{qikjl}(t)$ will be reset to zero. However, if a deadlock occurs, $T_{qikjl}(t)$ will increase over CSANN iterations because the feedback adjustments from RC_{qikjl} to ST_{ikq} and ST_{jlq} will remain similar due to the resource constraint violation between O_{ikq} and O_{jlq}. When $T_{qikjl}(t)$ reaches a prescribed threshold value H (e.g., $H = 5$), Algorithm 6 is triggered to swap the start times of O_{ikq} and O_{jlq} on machine q and reset $T_{qikjl}(t)$ to zero.

Figure 4 illustrates how a deadlock may happen and how Algorithm 6 solves the deadlock problem on

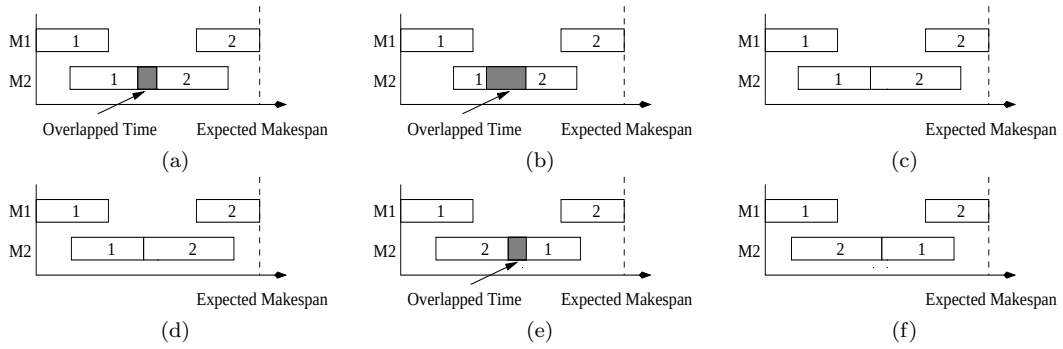

Fig. 4 Illustration of applying Algorithm 6 to solve the deadlock problem on a $J2|2|C_{max}$ JSSP: (a) an initial infeasible schedule with $H_{21221}(0) = 0$, (b) after the sequence constraint feedback adjustments in the first iteration, (c) after the resource constraint feedback adjustment in the first iteration and $H_{21221}(1) = 1$, (d) deadlock happens till the fifth iteration where $H_{21221}(5) = H = 5$ and Algorithm 6 is triggered to swap the start times of O_{122} and O_{212} on machine 2, (e) after the swapping and $H_{21221}(6) = 0$, and (f) another resource constraint feedback adjustment produces a feasible schedule.

a $J2|2|C_{max}$ JSSP where the sequence constraints between operations of the two jobs are given by $[O_{111}, O_{122}] \in S_1$ and $[O_{222}, O_{211}] \in S_2$. Suppose the start times of operations and the expected makespan are initialized as shown in Fig. 4(a) and we have $H_{21221}(0) = 0$. In the first iteration, the sequence constraint feedback adjustments will push the start times of O_{122} and O_{212} toward the right and left, respectively, giving Fig. 4(b). Then the resource constraint feedback adjustment from RC_{21221} to ST_{122} and ST_{212} in the first iteration will result in Fig. 4(c) with $H_{21221}(1) = 1$. It can be imagined that a deadlock will happen in the following iterations because the sequence constraint feedback adjustments will push T_{122} and T_{212} toward the right and left, respectively, and the resource constraint feedback will be similar over iterations, separating O_{122} and O_{212} on machine 2. This procedure continues till the fifth iteration, as shown in Fig. 4(d). Now $H_{21221}(5)$ reaches the threshold $H = 5$ and hence triggers Algorithm 6 to swap the start times of O_{122} and O_{212}, resulting in Fig. 4(e) and $H_{21221}(6)$ is reset to 0. Thereafter, another resource constraint feedback adjustment from RC_{22112} to ST_{212} and ST_{122} in the next iteration will produce a feasible schedule, as shown in Fig. 4(f).

From the above description, it can be seen that swapping only happens between two adjacent operations on a machine because, for two non-adjacent operations O_{ikq} and O_{jlq}, their corresponding resource constraint will be satisfied and hence $T_{qikjl}(t)$ will be zero.

3.4.3 Improving the quality of schedules

Unlike the schedules produced by the Giffler and Thompson algorithms, the feasible schedules produced by CSANN are usually inadmissible, where there may exist many idle times for each machine while some operations are available to be processed. The schedules can be improved by compacting away these idle times. In (Yang and Wang 2001), an algorithm is used to produce a semi-active schedule from the schedule produced by CSANN. It first sorts all the operations in a non-decreasing order of their start times, and then moves each operation from the first to the last in the ordered operation list to its earliest possible start time by local left-shift(s). This semi-active algorithm is applied as step 5 in Algorithm 4.

4 The improved CSANN-II model for JSSPs

4.1 Simplifying the RC-block for CSANN

In the original CSANN model, the major network complexity (and hence computational complexity) lies in the RC-block. In CSANN, any combination of two operations to be processed on a machine corresponds to one RC-block unit. Assuming each of the n jobs of a JSSP passes through all machines once, there will be $n(n-1)/2$ RC-block units for each machine. This is illustrated in Fig. 5 (a), where each ST-neuron ST_i ($i = 1, \ldots, n$) represents one operation of a job. This gives the whole RC-block a network complexity of $O(mn^2)$, a magnitude n larger than the network complexity of the SC-block, which is $O(mn)$. Hence, any simplification in the RC-block will further improve the performance of CSANN.

Fortunately, when we further consider the running of the RC-block, a potential improvement can be obtained. When we run CSANN for a JSSP, during each iteration of the RC-block, in fact usually only a part of the $n(n-1)/2$ resource constraints with respect to one machine are violated and hence are *relevant* to our

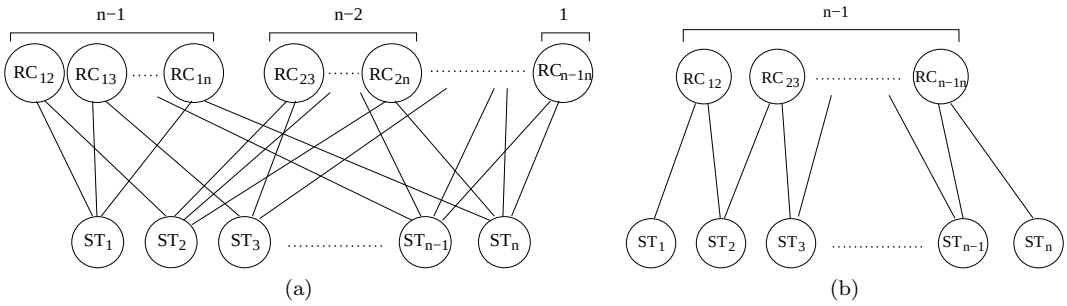

Fig. 5 Structure of the RC-block regarding one machine in (a) CSANN and (b) CSANN-II for a JSSP where each of the n jobs passes through all machines once.

Algorithm 7 Construct the RC-block adaptively

1: Before each iteration of the RC-block, sort the ST-neurons related to each machine according to their activations (i.e., present start times of relevant operations to be processed on the machine) in a non-decreasing order.
2: From the first to the last in the ordered ST-neuron list, construct one RC-block unit for two adjacent ST-neurons.

concern toward a feasible schedule. The ratio of relevant resource constraints on the average decreases with the solving progress of CSANN. In other words, with the solving progress of CSANN, quite a lot of RC-block units are, in fact, redundant in terms of solving resource constraint violations during each iteration since they do not involve violated resource constraints on a machine. This thinking leads to the following mechanism of constructing a dynamic RC-block adaptively during each iteration of CSANN instead of the original static RC-block, see Algorithm 7.

The new CSANN model with the above adaptive RC-block construction mechanism is named *CSANN-II*. In CSANN-II, the number of RC-block units on each machine is greatly reduced. Assuming each job of a JSSP passes through all machines once, the number of RC-block units regarding each machine will be reduced to $n-1$, as illustrated in Fig. 5(b). In Fig. 5(b), the n ST-neurons on a machine are sorted according to their activations in a non-decreasing order from left to right, i.e., $A_{ST_1} \leq A_{ST_2} \leq \cdots \leq A_{ST_n}$. The $n-1$ RC-neurons are constructed as follows: $RC_{i(i+1)}$ is connected to adjacent ST-neurons ST_i and ST_{i+1} $(i = 1, \ldots, n-1)$.

4.2 Comparison of network and computational complexity

For CSANN-II, for a traditional JSSP with m machines and n jobs where each job passes through all machines in a certain sequencing order, it requires mn ST-neurons, $n(m-1)$ SC-neurons, and $m(n-1)$ RC-neurons. In total, CSANN-II consists of $3mn-m-n$ neurons, which is in the order of $O(mn)$ instead of in the order of $O(mn^2)$ for CSANN. This is a reduction of magnitude n regarding the network complexity.

For each CSANN-II iteration, sorting the ST-neurons for each machine requires $O(n \log n)$ calculations by a quick sort algorithm (Cormen et al. 1990). It also requires $n(m-1)$ SC-neuron calculations and $m(n-1)$ RC-neuron calculations, resulting in a computational complexity of $O(mn \log n)$. In contrast, each iteration of CSANN requires $n(m-1)$ SC-neuron calculations and $mn(n-1)/2$ RC-neuron calculations, which is in the order of $O(mn^2)$. Hence, for each iteration of the neural network, CSANN-II achieves a reduction of magnitude $O(n/\log n)$ over CSANN with respect to the computational complexity.

4.3 New heuristic algorithms for CSANNs

For CSANN-II, Algorithm 5 and Algorithm 6 can also be used to speed up the solving process and enhance the convergence. In addition to these algorithms, this paper presents two other heuristics to be combined with CSANNs (both CSANN and CSANN-II) to achieve even better performance. They are described as follows.

4.3.1 Producing a proper expected makespan

For a JSSP without deadline constraints, which is the concern of this paper, we need to set an expected makespan as the common deadline of all jobs for CSANNs to run. The expected makespan can be taken as what a scheduler wants to achieve. The value of the expected makespan greatly affects the performance of CSANNs. If it is set too large, the quality of schedules produced by CSANNs will be low. Decreasing the expected makespan, the quality of schedules produced by CSANNs will increase. The computational time rises but is still reasonable (i.e., increasing slowly with the decrease of the expected makespan) if the value of expected makespan

Algorithm 8 Produce a proper expected makespan

1: Initialize the pre-processing stage cycle counter $k := 0$ and tightness factor $\gamma(0) := 0.5$
2: For cycle k of the pre-processing stage, run CSANN or CSANN-II for τ times with the expected makespan set to $\gamma(k) \times \sum P$, where $\sum P$ is the total processing time of all operations
3: Calculate $\bar{I} := (\sum_{i=1}^{i=\tau} I_i(k))/\tau$, where $I_i(k)$ is the number of iterations used by CSANN or CSANN-II for a schedule in the ith run of the kth cycle of the pre-processing stage
4: **if** $\bar{I} < \rho \times \sum O$ **then**
5: $\gamma(k+1) := \gamma(k) - \delta$, where δ is a preset decreasing factor for γ
6: k++
7: Go to step 2
8: **end if**
9: Return $\gamma(k) \times \sum P$ as the final expected makespan

is still above a certain level. However, when the value of the expected makespan is decreased to a certain value, called the *threshold value*, it will take CSANNs too long to produce a feasible schedule or even make it impossible to produce a feasible schedule. This qualifies the importance of setting a proper expected makespan for CSANNs to run.

In this paper, an adaptive heuristic algorithm, as shown in Algorithm 8, is proposed to produce a proper expected makespan by adding a pre-processing stage. Let $\sum P$ denote the total processing time of all operations of a JSSP and $\sum O$ denote the total number of operations, i.e., $\sum O = \sum_{i=1}^{i=n} \mu_i$. The expected makespan can be represented by $\gamma \times \sum P$ where γ is the *tightness factor*. Algorithm 8 works by cyclically decreasing the tightness factor γ from 0.5 until it reaches a value that requires CSANNs to use a number of iterations that is above a prefixed value (and results in CSANNs running for too long) to produce a schedule.

In Algorithm 8, τ is the total number of runs for each cycle of the pre-processing stage, $I_i(k)$ is the number of iterations that CSANN or CSANN-II uses to produce a schedule in the ith run of the kth cycle of the pre-processing stage, $\bar{I}(k)$ is the mean number of iterations that CSANN or CSANN-II uses to produce a schedule in the kth cycle of the pre-processing stage, ρ is a coefficient that roughly determines the mean number of iterations required by CSANNs to produce a schedule after the pre-processing stage, and δ is the decreasing factor for the tightness factor γ.

The design of Algorithm 8 is based on our experience of running CSANNs. A key factor relevant to the computational cost is the number of iterations that CSANNs require to produce a schedule. During our preliminary experiments, it seems that, if the expected makespan is properly set, the number of iterations that CSANNs require to produce a schedule has an approximately linear relationship with the total number of operations in the JSSP. When the number of iterations required by CSANNs to produce a schedule is approximately linear with the total number of operations, the computation time for a schedule is always reasonable. Hence, Algorithm 8 is designed to produce a proper expected makespan that makes the mean number of iterations that CSANN or CSANN-II requires to produce a schedule to be approximately linear with $\sum O$. This is realized by cyclical trials. For each cycle, we try to reduce the factor γ (and hence tighten the expected makespan) while ensuring that the mean number of iterations required to produce a schedule is still within a linear relation with $\sum O$ approximately.

4.3.2 Improving the quality of solutions

Just like CSANN, the feasible schedules produced by CSANN-II are usually inadmissible. In this paper, a heuristic algorithm is proposed to generate an active schedule from the feasible schedule produced by CSANN-II as follows: first, sort all operations in a non-decreasing order of their start times; then, from the first to the last in the ordered operation list, each operation is moved forward to its earliest start time as follows: if possible, performing a global left-shift; otherwise, if possible, performing a local left-shift. The details are shown in Algorithm 9.

The adjustments of the start times of all operations in Algorithm 9 are dynamic. That is, the adjusted start time of an operation takes effect in adjusting the latter operations. For example, supposing that $T_{i(k-1)p}$ has already been adjusted, when computing T_{ikq} of operation O_{ikq} which is just subsequent to $O_{i(k-1)p}$ of the same job i, the adjusted $T_{i(k-1)p}$ is used in (25) or (26) instead the original $T_{i(k-1)p}$. Hence, each operation needs only one adjustment to produce an active schedule. Fig. 6 illustrates the use of Algorithm 9 for an active schedule from an inadmissible schedule produced by CSANN-II on a $J2|2|C_{max}$ JSSP.

4.4 Hybrid approach of CSANNs and heuristics for JSSPs

The proposed heuristics can be combined with CSANN and CSANN-II for JSSPs. In practice,we can execute the hybrid approach that combines heuristics with CSANN or CSANN-II a number of times to produce a number of schedules and select the best one as the final schedule. The running strategy is shown in Algorithm 10, where some preliminary experiments may be carried out to set the proper values for W, H, τ, ρ, and δ. The procedure of running CSANN-II for one schedule is summarized in Algorithm 11.

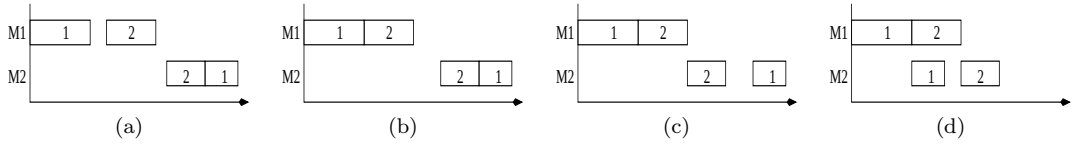

Fig. 6 Illustration of applying Algorithm 9 for an active schedule on a $J2|2|C_{max}$ JSSP: (a) an inadmissible schedule, (b) after a local left-shift for O_{211}, (c) after a local left-shift for O_{222}, and (d) after a global left-shift for O_{122}.

Algorithm 9 Produce an active schedule

Given a feasible schedule $\{T_{ikp}, i \in N, k \in \{1, \ldots, \mu_i\}, p \in M\}$ produced by CSANN-II, sort all operations in a non-decreasing order of their start times.
for each operation O_{ikp} from the first to the last in the ordered operation list **do**
  Denote the already orderly adjusted r operations on machine p by a list $L_p = \{O_1, \cdots, O_r\}$. If there is no operation already adjusted on machine p (i.e.,, $r = 0$), then L_p is empty.
  Denote the start and precessing time of $O_j \in L_p$ by T_{O_j} and P_{O_j}, respectively.
  For the convenience of description, define dummy operations O_{i0*} $(* \in M)$ and O_0 with the start time and precessing time being zero, i.e., $T_{i0*} = P_{i0*} = 0$ and $T_{O_0} = P_{O_0} = 0$.
  Declare two variables $Done = false$ and $j = 1$.
  while $(Done == false \;\&\&\; j \leq r)$ **do**
    if $(T_{O_j} - \max\{T_{O_{(j-1)}} + P_{O_{(j-1)}}, T_{i(k-1)q} + P_{i(k-1)q}\} \geq P_{ikp})$ **then**
      Perform a global left shift as follows:
      $T_{ikp} = \max\{T_{O_{(j-1)}} + P_{O_{(j-1)}}, T_{i(k-1)q} + P_{i(k-1)q}\}$ (25)
      $\{O_{(j-1)}$ is either O_0 or the operation on machine p that has been adjusted just before O_j, and $T_{i(k-1)q}$ is the start time of $O_{i(k-1)q}$, which is either the dummy operation O_{i0q} or the operation of job i that precedes $O_{ikp}\}$
      $Done = true$
    else
      $j = j + 1$
    end if
  end while
  if $(Done == false)$ **then**
    if $(T_{ikp} > \max\{T_{i(k-1)q} + P_{i(k-1)q}, T_{O_r} + P_{O_r}\})$ **then**
      Perform a local left shift as follows:
      $T_{ikp} = \max\{T_{i(k-1)q} + P_{i(k-1)q}, T_{O_r} + P_{O_r}\}$ (26)
    end if
  end if
end for

Algorithm 10 The running strategy of the hybrid approach

1: Construct a JSSP-specific CSANN or CSANN-II
2: Set the maximum number of schedules $MaxSched$ to be produced and values for W, H, τ, ρ, and δ according to some preliminary experiments
3: Apply Algorithm 8 to produce a proper expected makespan
4: Run CSANN or CSANN-II to produce one schedule with the produced expected makespan
5: If $MaxSched$ is reached, stop; otherwise, go to step 4

Algorithm 11 Produce a schedule by CSANN-II

1: Randomly initialize $T_{ikp}(0)$ for each operation O_{ikp}, and take it as the initial net input $I_{ST_{ikp}}$ to ST_{ikp}.
2: Run each SC-neuron SC_{ikl} of the SC-block: calculate its activation with (10). If $A_{SC_{ikl}}(t) \neq 0$, adjust activations of related ST-neurons with (21) and (22) if Algorithm 5 is triggered, or with (11) and (12), otherwise.
3: Construct the RC-block by Algorithm 7.
4: Run each RC-neuron RC_{qikjl} of the RC-block: calculate its activation with (14) or (18). If $A_{RC_{qikjl}}(t) \neq 0$, adjust $A_{ST_{ikq}}(t+1)$ and $A_{ST_{jlq}}(t+1)$ with (23) and (24) if Algorithm 6 is triggered, or with (15) and (16) or (19) and (20), otherwise.
5: Repeat step 2 to 4 until a feasible schedule is produced.
6: Use Algorithm 9 to produce an active schedule from the feasible schedule produced by CSANN-II.

Table 2 Benchmark test JSSPs, where "B/O" means the best-known or optimal makespan value

JSSP	LA01	LA06	LA11	LA16	LA21	LA26
$n \times m$	10×5	15×5	20×5	10×10	15×10	20×10
B/O	**666**	**926**	**1222**	945	**1046**	1218

5 Experimental study

5.1 Experimental setting

The experimental study was executed on a PC with 2.2Ghz AMD Opteron 848 CPU using the GNU C++ programming environment under the Linux system. It has three purposes: validating the computational complexity of CSANN-II over CSANN for JSSPs, comparing the performance of CSANN and CSANN-II over the three classical heuristics for JSSPs described in Sect. 2.3, and analyzing the effect of some key component algorithms proposed for CSANN-II. Three sets of experiments were carried out regarding the three purposes, respectively. For CSANNs, the parameters W and H are set as in (Yang and Wang 2001): $W = 0.5$ and $H = 5$.

In the experimental study we select the benchmark JSSPs LA01, LA06, LA11, LA16, LA21, and LA26 from Lawrence (1984) as the test bed. Table 2 shows the problem sizes and the best known or optimal values (B/O in short) of the makespan of these test JSSPs, where the optimal values are shown in the bold font. The source data of these JSSPs can be found in (Beasley

Table 3 The settings of the tightness factor γ for each JSSP

JSSP	γ										
LA01	0.5	0.45	0.4	0.35	0.3						
LA06	0.5	0.45	0.4	0.35	0.3	0.28					
LA11	0.5	0.45	0.4	0.35	0.3	0.28					
LA16	0.5	0.45	0.4	0.35	0.3	0.28	0.26	0.24			
LA21	0.5	0.45	0.4	0.35	0.3	0.28	0.26	0.24	0.22	0.2	
LA26	0.5	0.45	0.4	0.35	0.3	0.28	0.26	0.24	0.22	0.2	0.18

1990). These JSSPs form a good test bed because we can study the effect of not only increasing n while fixing m, but also doubling m while fixing n on the performance of CSANNs.

5.2 Experiments on the computational complexity

In the first set of experiments, we focus on the study of the computational complexity of CSANN-II over the original CSANN based on the six test JSSPs. In this set of experiments, CSANN uses Algorithms 4, 5 and 6, and CSANN-II uses Algorithms 5, 6, 7, 9, and 11. Algorithm 8 is switched off for both CSANN and CSANN-II and the expected makespan value is manually set to $\gamma \times \sum P$ where $\gamma \leq 0.5$. In order to test the effect of the expected makespan value on the performance of CSANNs, γ needs to be set to different values. In order to properly study the effect of different settings of γ on the performance of CSANNs, we carried out some preliminary experiments. In the preliminary experiments, for each JSSP problem, CSANN and CSANN-II were run a few times with γ manually decreased from 0.5 to a value that becomes unreasonable. A value of γ is said *unreasonable* if it is too tight such that it takes CSANNs too long to produce a feasible schedule or it is impossible for CSANNs to produce a feasible schedule.

According to our preliminary experiments, the settings of γ shown in Table 3 basically capture the time growth feature of CSANNs for each test JSSP and hence were used in the following experiments. That is, γ is decreasingly set to the values from 0.5 to 0.3 with a step of 0.05 for all JSSPs and is then reduced from 0.3 with a step 0.02 till it becomes unreasonable for each test JSSP. For each value of γ, CSANN or CSANN-II was run 10000 times with different random seeds to produce 10000 schedules for each test JSSP and the time used for all runs was recorded. For each run, the number of iterations of CSANN or CSANN-II was also recorded. The experimental results regarding the best and mean makespan produced by CSANN and CSANN-II over the 10000 runs against different settings of γ on the JSSPs are plotted in Fig. 7. The experimental results regarding the computational complexity of CSANNs, including the required number of iterations per schedule (IPS), the run time per schedule (TPS), and the run time per iteration (TPI) against different values of γ on the JSSPs are plotted in Fig. 8. In Fig. 8, the data were averaged over the 10000 runs, the computational complexity measures are log-scaled, and TPS and TPI are plotted in the unit of millisecond and microsecond respectively.

From Figs. 7 and 8, several results can be observed and are analyzed as follows. First, CSANN-II greatly outperforms CSANN with respect to the quality of schedules produced, either the best makespan or the mean makespan, under almost all values of γ on all the test JSSPs (see Fig. 7). On LA21 and LA26, when γ is set to loose values, e.g., bigger than 0.3, even the mean makespan achieved by CSANN-II is better than the best makespan achieved by CSANN. Under almost all settings of γ, CSANN-II has reached the optimal schedule for JSSPs LA01, LA06, and LA11.

Second, with respect to the computational complexity of CSANNs, it can be seen that CSANN-II greatly (note that the y-axis is log-scaled in Fig. 8) shortens the mean run time used to produce a schedule over CSANN under all values of γ on all the test JSSPs with only one exception occurred on LA16 with $\gamma = 0.24$ where CSANN-II is beaten by CSANN[3].

One thing to notice is that on the average, CSANN-II requires slightly more iterations for a schedule on all JSSPs, see the IPS lines in Fig. 8. This happens because the dynamic RC-block in CSANN-II is not totally equivalent to the static RC-block in CSANN. For example, in Fig. 5(a), except for the $n-1$ RC-neurons that correspond to the $n-1$ RC-neurons in Fig. 5(b), not all the other $n(n-1)/2-(n-1) = (n-1)(n-2)/2$ RC-neurons are irrelevant. Among them, some relevant RC-neurons will help solve the resource constraint violations during each iteration. This slightly helps shorten the total number of iterations to reach a feasible schedule. However, this benefit of slightly reduced number of iterations in CSANN is rather small in comparison with the benefit of reduced computational cost per iteration due to greatly reduced size of the RC-block in CSANN-II. The overall effect is that on the average the TPS required by CSANN-II is significantly shorter than the TPS required by CSANN.

In Sect. 4.2, we have analyzed that for each iteration of the neural network, CSANN-II achieves a re-

[3] Careful scrutinization reveals that under this exceptional case, CSANN-II has used an extremely large number of iterations to produce a schedule for a small part of the 10000 runs. This results in that CSANN-II uses a much bigger mean time per schedule than CSANN does.

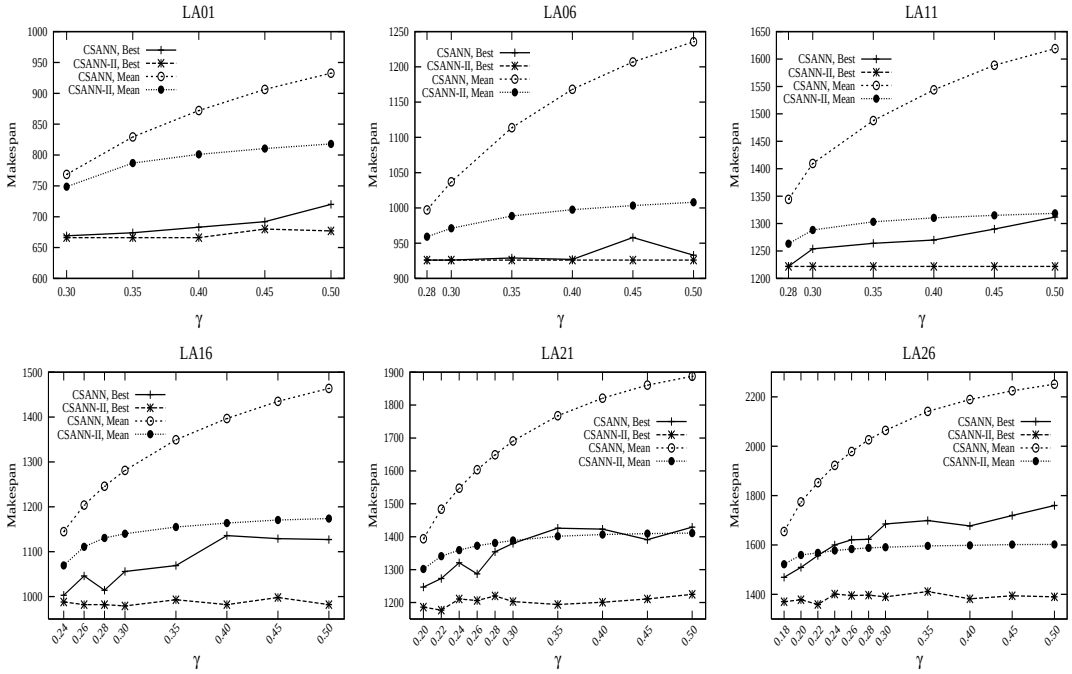

Fig. 7 Experimental results on the best and mean makespan produced by CSANNs against different values of γ on the JSSPs.

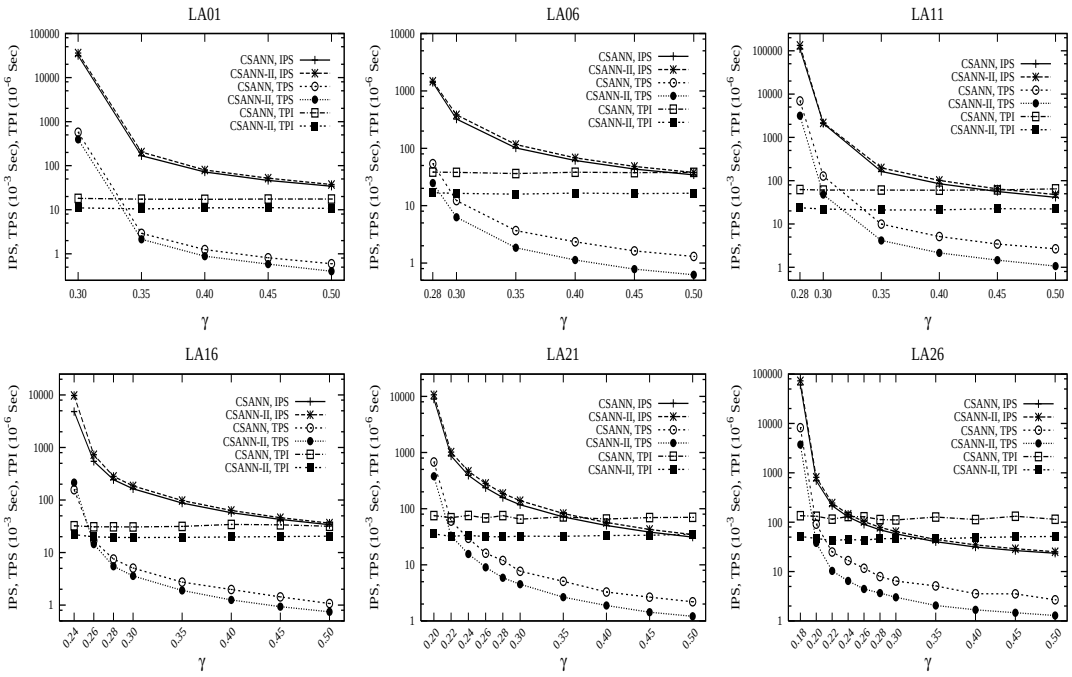

Fig. 8 Experimental results on the mean iterations per schedule (IPS), mean time per schedule (TPS), and mean time per iteration (TPI) used by CSANNs against different values of γ on the JSSPs. The y-axis is log-scaled and the unit for TPS and TPI is 10^{-3} second and 10^{-6} second, respectively.

duction in the order $O(n/\log n)$ over CSANN regarding the computational complexity. In order to see whether this is the case in the experiments, we calculate the weighted ratio of the TPI required by CSANN to the TPI required by CSANN-II for each value of γ and on each JSSP using the following formula:

$$R_{TPI} = \frac{TPI(\text{CSANN})}{TPI(\text{CSANN-II}) \times (n/\log n)}, \tag{27}$$

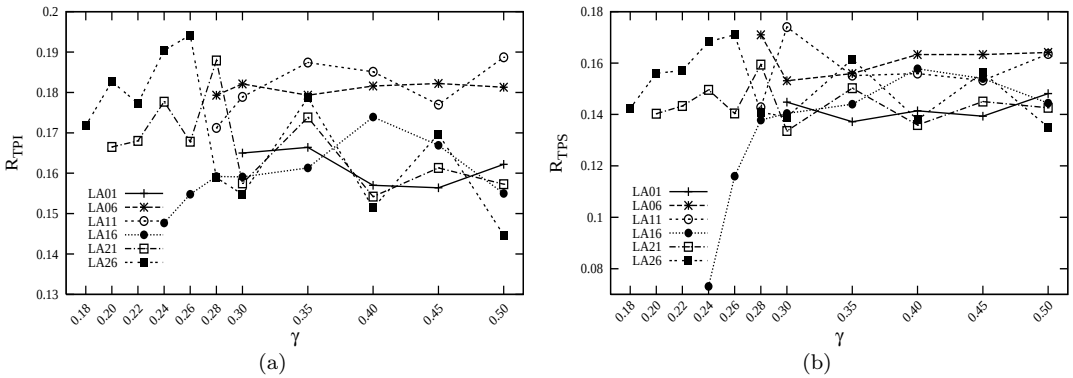

Fig. 9 Results of (a) R_{TPI} and (b) R_{TPS} against different γ on the JSSPs.

where $\log n$ is 10 based. We also calculate the weighted ratio of the TPS by CSANN to the TPS by CSANN-II using the following formula:

$$R_{TPS} = \frac{TPS(\text{CSANN})}{TPS(\text{CSANN-II}) \times (n/\log n)}. \quad (28)$$

The calculated results regarding R_{TPI} and R_{TPS} are plotted in Fig. 9(a) and (b), respectively. From Fig. 9(a), it can be seen that R_{TPI} is roughly bounded in the range of $[0.14, 0.2]$ over all values of γ for CSANNs and on all the JSSPs. In other words, we have the following relationship between *TPI*(CSANN) and *TPI*(CSANN-II):

$$TPI(\text{CSANN}) = 0.14 \sim 0.2(n/\log n)TPI(\text{CSANN-II}). \quad (29)$$

This result confirms that CSANN-II really achieves a reduction in the order $O(n/\log n)$ over CSANN on the test JSSPs with respect to the TPI.

From Fig. 9(b), it can be seen that R_{TPS} is roughly bounded in the range of $[0.13, 0.18]$ over almost all γ settings and test JSSPs with only two exceptions on LA16 when $\gamma = 0.24$ and 0.26. The upper and lower bounds for R_{TPS} are slightly lower than those for R_{TPI}. This is because, as analyzed before, CSANN-II requires slightly more iterations per schedule than CSANN. However, the effect is rather small with a factor of about 10% (i.e., $max\{(0.14 - 0.13)/0.14, (0.2 - 0.18)/0.2\} \approx 10\%$). Roughly, we can say that CSANN-II achieves a reduction in the same order $O(n/\log n)$ over CSANN on the test JSSPs regarding the TPS measure.

Regarding the effect of m on the performance of CSANNs, from Fig. 8 it can be seen that when m is doubled, the TPI of CSANNs is approximately doubled. And from Fig. 9, it can also be seen that the value of m has no clear effect on the value of R_{TPI} (and R_{TPS}). This result is consistent with our analysis in Sect. 4.2: the computational complexity per iteration for CSANN and CSANN-II is linear with m while the improvement of CSANN-II over CSANN is of the order $O(n/\log n)$ without an explicit impact from m.

Third, regarding the effect of the tightness factor γ on the performance of CSANNs, from Fig. 7 it can be seen that the value of γ affects the best schedule produced by CSANN, but it shows no clear effect on the best schedule produced by CSANN-II. However, γ has a significant effect on the mean makespan produced by both CSANN and CSANN-II. The smaller (and hence the tighter) the value of γ, the better the schedule produced on the average. The trade-off for this gain is, not surprisingly, the computational cost.

From Fig. 8, it can be seen that when γ becomes tighter, the mean IPS and hence the mean TPS required by CSANNs increases. And for each JSSP when γ is decreased from 0.5, the IPS and TPS by CSANNs increase slowly at the early stage. When γ is decreased below a certain value (i.e., the threshold value), they increase significantly in a power law. For example, on LA21, the mean IPS of CSANN-II increases approximately linearly from 34.43 to 138.80 when γ decreases from 0.5 to 0.3. Then, the IPS increases from 186.54 to 1019.40 approximately in a power law when γ decreases from 0.28 to 0.22. Finally, it increases exponentially from 1019.40 to 10672.38 when γ decreases from 0.22 to 0.2. The threshold value varies with the JSSP. The larger the problem size, the smaller the threshold value.

The above two aspects together validate the importance of selecting a proper γ, i.e., a proper expected makespan, for CSANNs to run. There is a big trade-off between the quality of the schedules and the computational cost. Ideally, γ should be set at the end of the linear growth region for a good schedule quality and reasonable run time of CSANNs. This is the aim of the proposed Algorithm 8, which tries to produce a proper expected makespan by iterative trials. CSANNs with Algorithm 8 are the concern of the second set of experiments to be presented in the next section.

Table 4 Experimental results of CSANNs and classical heuristics over 50 runs

	Best Makespan Produced (min/ave/std)		
JSSP	LA01	LA06	LA11
CSANN-II	666/666.2/0.9	926/926.0/0.0	1222/1222.0/0.0
CSANN	666/677.6/7.1	926/926.2/1.0	1222/1247.1/12.9
GT-Act	666/698.3/10.5	926/934.9/7.6	1225/1247.1/9.2
GT-ND	668/687.0/9.7	926/929.7/5.3	1222/1238.6/9.5
GT-Rule	674/696.1/10.4	926/939.8/9.4	1236/1253.3/8.0
JSSP	LA16	LA21	LA26
CSANN-II	962/978.5/3.3	1162/1186.1/8.3	1336/1361.3/10.9
CSANN	1011/1053.4/14.0	1280/1318.6/21.3	1481/1530.3/22.4
GT-Act	1012/1045.1/11.8	1253/1295.1/17.4	1469/1529.9/19.0
GT-ND	1008/1051.2/16.9	1287/1317.1/14.7	1483/1513.5/13.5
GT-Rule	1020/1047.4/12.4	1243/1301.3/18.0	1508/1540.7/15.0
	Time Used in Seconds (min/ave/std)		
JSSP	LA01	LA06	LA11
CSANN-II	47/60.5/8.5	91/133.8/20.8	131/203.9/51.0
CSANN	69/91.8/13.6	134/237.4/49.5	290/558.8/125.7
GT-Act	9/15.6/1.3	35/35.1/0.3	32/33.1/0.6
GT-ND	8/8.0/0.1	18/18.3/0.4	33/34.1/0.5
GT-Rule	8/8.9/0.2	19/19.6/0.5	35/36.0/0.3
JSSP	LA16	LA21	LA26
CSANN-II	170/336.9/110.9	269/543.4/136.6	589/1238.9/487.0
CSANN	214/326.4/55.5	501/1044.7/243.3	1057/3089.9/1341.7
GT-Act	15/15.6/0.5	33/35.2/0.6	64/65.4/0.6
GT-ND	15/15.5/0.5	34/34.7/0.5	64/64.2/0.4
GT-Rule	16/16.8/0.4	36/37.7/0.5	78/121.3/10.6

5.3 Experiments on comparisons with classical heuristics

In the second set of experiments, we focus on the performance comparisons of CSANNs and three classical heuristics, GT-Act, GT-ND, and GT-Rule, based on the test JSSPs listed in Table 2. In this set of experiments, CSANN uses Algorithm 10 with Algorithms 4, 5, and 6, and CSANN-II uses Algorithm 10 with Algorithms 11, 5, 6, and 9. Algorithm 8 is now switched on for both CSANN and CSANN-II with $\tau = 10$, $\rho = 1$, and $\delta = 0.01$. Here, parameters τ, ρ, and δ were set according to some preliminary experiments without systematic tuning.

For each run of a method on a test JSSP, 10^5 schedules[4] were calculated with the intermediate best-so-far schedule recorded every 100 schedules. And for each run, the final best schedule and time used were also recorded. In order to avoid the effect a random seed may have, 50 runs with different random seeds were carried out for each method on each test problem and the mean results over 50 runs are reported.

[4] For CSANNs the schedules calculated during the preprocessing stage were also counted into the total 10^5 schedules.

The experimental results regarding the makespan of final best schedule and time used in seconds are given in Table 4, where *min/ave/std* means minimum, average and standard deviation over 50 runs of algorithms, respectively. The statistical comparison of methods regarding the best makespan produced and the run time used over 50 runs by one-tailed t-test with 98 degrees of freedom at a 0.05 significance level is given in Table 5, In Table 5, if the t-test value regarding Method 1 – Method 2 is negative or positive, it means that Method 1 is better than or worse than Method 2, respectively, regarding the corresponding performance measure. If the absolute t-test value is greater than 1.660, the performance difference between Method 1 and Method 2 is significant. Those t-test values that mean significant differences of performance between methods are shown in bold font in Table 5. The experimental results regarding best-so-far makespan produced by different methods against schedules are plotted in Fig. 10, where the data were averaged over 50 runs. From Table 4, Table 5, and Fig. 10, several results can be observed.

First, CSANN-II significantly outperforms CSANN with respect to both the quality of produced schedules and time used on almost all test JSSPs, as indicated

Table 5 The t-test results of comparing CSANNs and classical heuristics over 50 runs

JSSP	LA01	LA06	LA11	LA16	LA21	LA26
t-Test Result Regarding the Best Makespan Produced						
CSANN-II – CSANN	**-11.30**	-1.66	**-13.75**	**-36.83**	**-40.92**	**-48.04**
CSANN-II – GT-Act	**-21.64**	**-8.35**	**-19.36**	**-38.51**	**-39.92**	**-54.50**
CSANN-II – GT-ND	**-15.13**	**-4.96**	**-12.30**	**-29.84**	**-54.86**	**-62.13**
CSANN-II – GT-Rule	**-20.40**	**-10.41**	**-27.80**	**-38.02**	**-41.02**	**-68.43**
t-Test Result Regarding the Time Used						
CSANN-II – CSANN	**-13.83**	**-13.64**	**-18.50**	0.60	**-12.70**	**-9.17**
CSANN-II – GT-Act	**36.96**	**33.53**	**23.68**	**20.48**	**26.31**	**17.04**
CSANN-II – GT-ND	**43.67**	**39.26**	**23.55**	**20.49**	**26.33**	**17.06**
CSANN-II – GT-Rule	**42.89**	**38.80**	**23.28**	**20.41**	**26.18**	**16.22**

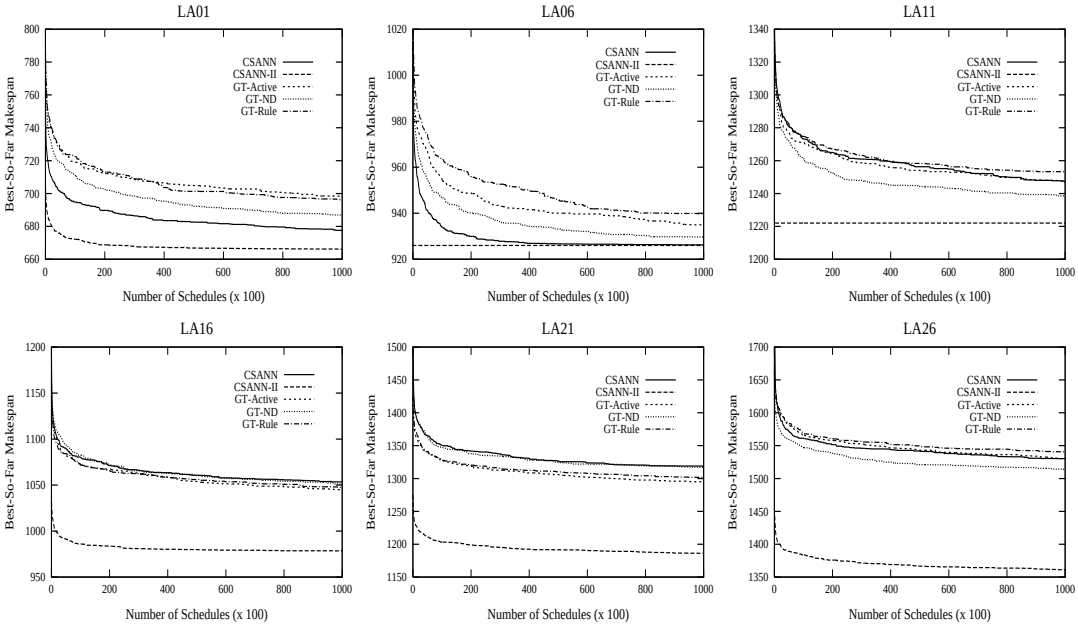

Fig. 10 The dynamic performance of methods regarding the best-so-far makespan against schedules on the JSSPs. The data were averaged over 50 runs.

in the t-test results regarding CSANN-II – CSANN. This result is consistent with our first set of experiments. In order to see whether Algorithm 8 for CSANNs works, we also recorded the tightness factor γ achieved after the pre-processing stage of each run and report the results in Table 6, where *min/ave/std* means minimum, average and standard deviation over 50 runs of CSANNs, respectively. From Table 6 and Fig. 7, it can be seen that Algorithm 8 achieved a slightly loose value of γ for LA01, LA06 and LA11 and a good value of γ on larger JSSPs, LA16, LA21 and LA26. Here, a "loose" or "good" value of γ is relative to the "threshold value". For example, for LA01, it is "good" to set γ to the range of $[0.35, 0.4]$ since in this range TPS is low but the solution quality is high, see Fig. 6 and 7. Algorithm 8 reaches an average value of 0.46 for CSANN-II for LA01 and 0.45 for CSANN for LA01, respectively, which is "loose" in comparison to the range $[0.35, 0.4]$. Generally speaking, Algorithm 8 works reasonably well for CSANNs under the given parameters though proper tuning may further improve the perfor-

Table 6 The tightness factor γ produced by Algorithm 8

Method	γ (min/ave/std)		
JSSP	LA01	LA06	LA11
CSANN-II	0.42/0.46/0.020	0.36/0.40/0.021	0.36/0.42/0.026
CSANN	0.40/0.45/0.020	0.34/0.39/0.028	0.36/0.39/0.024
JSSP	LA16	LA21	LA26
CSANN-II	0.32/0.36/0.020	0.26/0.30/0.021	0.20/0.22/0.010
CSANN	0.32/0.34/0.015	0.26/0.28/0.018	0.20/0.22/0.012

Table 7 Experimental results regarding the best makespan produced over 50 runs of CSANNs and classic heuristics with fixed run time, where the data are shown in the format of min/ave/std

JSSP	LA01	LA06	LA11
Run Time (Seconds)	250	375	500
CSANN-II	666/666.0/0.0	926/926.0/0.0	1222/1222.0/0.0
CSANN	666/672.0/4.9	926/926.1/0.52	1224/1249.0/12.5
GT-Act	666/673.3/5.4	926/926.0/0.1	1222/1230.9/6.3
GT-ND	666/667.0/2.0	926/926.0/0.0	1222/1222.9/2.5
GT-Rule	666/674.5/5.8	926/927.0/1.9	1222/1235.5/7.5
JSSP	LA16	LA21	LA26
Run Time (Seconds)	500	750	1000
CSANN-II	956/976.8/5.1	1162/1183.1/7.2	1342/1363.7/9.9
CSANN	1011/1050.8/13.7	1281/1324.0/18.8	1481/1544.5/22.7
GT-Act	995/1016.5/8.5	1233/1258.8/9.5	1463/1497.8/13.4
GT-ND	961/1012.4/13.3	1241/1278.7/14.1	1423/1472.3/16.6
GT-Rule	995/1019.4/8.6	1226/1267.0/12.2	1483/1508.6/11.3

Table 8 The t-test results of comparing CSANNs and classic heuristics with fixed run time regarding the best makespan produced over 50 runs

JSSP	LA01	LA06	LA11	LA16	LA21	LA26
Run Time (Seconds)	250	375	500	500	750	1000
CSANN-II – CSANN	**-8.65**	-1.63	**-15.27**	**-35.88**	**-49.46**	**-51.54**
CSANN-II – GT-Act	**-9.63**	-1.00	**-10.05**	**-28.32**	**-45.02**	**-56.93**
CSANN-II – GT-ND	**-3.41**	0.0	**-2.64**	**-17.70**	**-42.74**	**-39.75**
CSANN-II – GT-Rule	**-10.40**	**-3.54**	**-12.70**	**-30.09**	**-41.92**	**-67.97**
GT-ND – GT-Act	**-7.84**	-1.00	**-8.38**	**-1.84**	**8.32**	**-8.48**
GT-ND – GT-Rule	**-8.73**	**-3.54**	**-11.25**	**-3.16**	**4.44**	**-12.80**

mance of CSANNs. On average, Algorithm 8 achieved slightly tighter value of γ for CSANN than for CSANN-II. This is because, as analyzed before, CSANN-II requires slightly more iterations per schedule than CSANN does.

Second, comparing the performance of CSANNs with the three heuristic algorithms regarding the quality of produced schedules, it can be seen that CSANN-II also significantly outperforms GT-Act, GT-ND and GT-Rule on all test JSSPs and that CSANN outperforms them on LA01 and LA06 while performs similarly as or is beaten by them on the other JSSPs. This result can be more clearly viewed from the dynamic performance of methods in Fig. 10. Fig. 10 shows that CSANN-II performs much better than the other methods on the JSSPs. On LA06 and LA11, CSANN-II even achieved the optimal solution within 100 schedules on the average.

Third, when considering the computational cost of different methods, both CSANN and CSANN-II spend significantly more time than GT-Act, GT-ND, and GT-Rule. This is easy to understand because CSANNs, under the values of γ produced by Algorithm 8, need tens or hundreds of iterations for one schedule while GT-Act, GT-ND and GT-Rule produce one schedule in just one iteration.

In order to carry out a fairer comparison among different methods regarding the computational time, further experiments were carried out to run each method on each test JSSP for a certain fixed time. We also carried out 50 runs for each method on a JSSP. For each run of a method on a JSSP, the run time was limited to $5 \times \sum O$ seconds, where $\sum O$ is the total number of operations. For each run, the best schedule produced and the total number of schedules produced were recorded. The experimental results regarding the best makespan produced by CSANNs and classic heuristics and the corresponding t-test results of comparing them over 50 runs are shown in Table 7 and Table 8 respectively. The average number of schedules produced by CSANNs and classic heuristics over 50 runs is shown in Table 9.

Table 7 shows that CSANN is now beaten by the three heuristic methods on large JSSPs. However, CSANN-II still significantly outperforms CSANN as well as GT-

Table 9 The average number of schedules produced by CSANNs and classic heuristics with fixed run time over 50 runs

JSSP	LA01	LA06	LA11	LA16	LA21	LA26
Run Time (Seconds)	250	375	500	500	750	1000
CSANN-II	435464.3	306350.6	234634.4	227509.8	147369.5	88392.6
CSANN	322730.8	153152.2	92711.4	132779.9	63429.8	36318.4
GT-Act	3129798.9	2065252.2	1492171.4	3258000.8	2153771.9	1545604.7
GT-ND	3101206.7	1961333.2	1480097.7	3250826.2	2157310.0	1543490.5
GT-Rule	2845412.3	1903676.1	1428637.2	3096360.8	1811451.4	1511530.7

Table 10 Experimental results of analyzing the component algorithms of CSANNs regarding the best makespan produced over 50 runs, where the data are shown in the format of min/ave/std

JSSP	LA01	LA06	LA11	LA16	LA21	LA26
CSANN-Act	666/666.5/1.2	926/926.0/0.0	1222/1222.0/0.0	956/980.3/5.3	1173/1193.6/9.4	1338/1374.8/11.8
CSANN-AP	666/666.2/0.7	926/926.2/0.0	1222/1222.0/0.0	959/978.7/3.1	1160/1181.6/8.9	1334/1361.6/8.8

Act, GT-ND, and GT-Rule on nearly all JSSPs, as indicated in the t-test results in Table 8. Among the three heuristic algorithms, it seems that GT-ND outperforms GT-Act and GT-Rule on most JSSPs, as indicated in the t-test results regarding GT-ND – GT-Act and GT-ND – GT-Rule respectively. This result is consistent with other research work in the literature that non-delay schedules show a better mean solution quality than active ones (Bierwirth and Mattfeld 1999).

Next, let us consider the average number of schedules produced by different methods within the fixed time on each JSSP. From Table 9, it can be seen that GT-Act and GT-ND produce a similar number of schedules while GT-Rule generates fewer schedules. This is natural due to their similar computational complexity. CSANN-II generates about 6 to 18 times fewer schedules than GT-Act and GT-ND on the JSSPs while dominating them in mean solution quality. In other words, the peak quality per schedule (QPS) generated by CSANN-II is much higher than other methods. This is an interesting result because both CSANN-II and GT-Act aim at active schedules with uncertainty: CSANN-II starts from random initialized start times of operations while GT-Act schedules a random operation from the conflict set C iteratively. The difference lies in that GT-Act searches in the whole domain of active schedules randomly while CSANN-II searches from the larger domain of feasible schedules, which are properly filtered by the expected makespan imposed (the produced schedules are then mapped to active ones by Algorithm 9). This gives CSANN-II an advantage over GT-Act (and GT-ND and GT-Rule similarly). The result that CSANN-II has a much higher QPS is also important since for many advanced scheduling systems, the QPS is a key issue. For these systems, CSANN-II appears better suited than some widely applied heuristics like GT-Act, GT-ND, and GT-Rule.

5.4 Experiments on analyzing the component algorithms proposed for CSANN-II

There are several component algorithms proposed in CSANN-II, i.e., Algorithms 7, 8 and 9. In Sect. 5.2, we have shown that Algorithm 7 improves the computational complexity of CSANN-II over CSANN and that Algorithm 8 eliminates the need of setting up the tightness factor for CSANNs. In this set of experiments, we further analyze the effect of Algorithms 8 and 9 on the schedule quality improvement of CSANNs. Two CSANN variants were investigated on the test JSSPs listed in Table 2: one is CSANN without Algorithm 8 but with Algorithm 9 used in Step 5 of Algorithm 4 to produce an active schedule instead of a semi-active one, which is denoted *CSANN-Act*. The other is CSANN with both Algorithm 8 and Algorithm 9 switched on, which is denoted *CSANN-AP*. For CSANN-Act, the tightness parameter is set to $\gamma = 0.5$. For CSANN-AP, the parameters in Algorithm 8 are set the same as in Sect. 5.2 with $\tau = 10$, $\rho = 1$, and $\delta = 0.01$.

For each CSANN on a JSSP, 50 runs were carried out with 10^5 schedules produced per run and the intermediate best-so-far schedule recorded every 100 schedules. The experimental results regarding the best makespan produced by CSANN-Act and CSANN-AP and the corresponding t-test results of comparing them and CSANN and CSANN-II are shown in Table 10 and Table 11 respectively.

From Table 10 and Table 4, it can be seen that CSANN-AP significantly outperforms CSANN on almost all test JSSPs, as indicated in the t-test results in

Table 11 The t-test results of comparing CSANNs with different component algorithms over 50 runs

JSSP	LA01	LA06	LA11	LA16	LA21	LA26
CSANN-AP – CSANN	**-11.33**	-1.66	**-13.75**	**-36.83**	**-41.93**	**-49.65**
CSANN-AP – CSANN-Act	-1.61	0.0	0.0	**-1.83**	**-6.52**	**-6.32**
CSANN-II – CSANN-AP	0.0	0.0	0.0	-0.35	**2.64**	-0.16

Table 11. That is, Algorithm 9 significantly improves the quality of solutions produced. Another observation is that CSANN-AP outperforms CSANN-Act, especially on large JSSPs. This result indicates that Algorithm 8 also contributes toward a better quality of solutions produced by CSANN. The third result is that CSANN-II performs statistically equivalent or similar to CSANN-AP regarding the solution quality, except on LA21 where CSANN-II is significantly beaten by CSANN-AP.

6 Conclusions and future work

This paper investigates an improved constraint satisfaction adaptive neural network, CSANN-II, for the JSSP. In CSANN-II, the topology corresponding to the resource constraints is simplified according to the online resource constraint satisfaction situation when it is running via a simple sorting algorithm. Consequently, CSANN-II's computational time per schedule is reduced over the original CSANN model. Some heuristics are also proposed to improve the performance of CSANN and CSANN-II, including producing a proper expected makespan and improving the quality of produced schedules.

Based on a set of benchmark JSSPs, we experimentally studied the computational complexity of CSANN-II over CSANN and compared the performance of CSANN-II over CSANN and three classical heuristics. From the experimental results, we analyze the strength and weakness of CSANNs for JSSPs. According to the experimental results and analysis, the following conclusions can be drawn on the test JSSPs.

First, CSANN-II speeds up in the order $O(n/\log n)$ over CSANN, including the time per iteration and time per schedule measures. CSANN-II outperforms CSANN with respect to the quality of solutions primarily due to its use of Algorithm 9, which is better than the semi-active algorithm used in CSANN.

Second, the proposed adaptive scheme for producing a proper expected makespan works well for CSANNs. The experimental results show that reasonable values can be found for the expected makespan automatically. This eliminates the need and difficulty of manually setting the value for CSANNs to solve JSSPs since setting a reasonably good value for the expected makespan can usually lead to good schedules. Of course, we may still manually tune the value with human expertise for optimal results.

Third, CSANN-II outperforms three classical heuristic algorithms with respect to the quality of solutions but requires more time for a schedule. When the run time is fixed, CSANN-II can reach better solutions than the three classical heuristic algorithms with fewer schedules. Hence, CSANN-II has a much higher QPS, which is an important feature.

There are several avenues to pursue for future work regarding CSANN-II. First, just as the three classical heuristic algorithms studied in this paper, CSANN-II is a working tool for JSSPs. Whenever the initial start times for operations are given, CSANN-II will return a deterministic schedule. It is important to see that the QPS generated by CSANN-II is much higher than the three classical heuristic algorithms studied, which are widely used as the fundamental tools for advanced JSSP systems (Hart et al. 2005). Just as the three classical heuristic algorithms, CSANN-II can surely act as a good fundamental tool for constructing advanced hybrid intelligent systems for JSSPs. For example, combining it with a simple local search scheme has shown some promising results for JSSPs (Yang 2006). Combining it with other meta-heuristic methods may also produce promising results. For example, we may integrate CSANN-II into GAs and compare the performance with other state-of-the-art GA based scheduling systems for the JSSP. This is an interesting work now under investigation.

Second, CSANN-II itself may be further improved. The knowledge that becomes available during the solving process can be further integrated to improve its performance. For example, with the solving process of CSANN-II, more and more neuron units will correspond to already satisfied sequence constraints and resources constraints and hence become irrelevant in terms of solving constraint violations for a feasible schedule. This knowledge may be used to avoid calculating such irrelevant units and further reduce the computational cost.

Finally, the JSSP studied in this paper is basically an academic problem in scheduling. For the sake of simplicity and clarity, we have focused on JSSPs that are not oversubscribed, have no (tight) deadlines for jobs,

and do not allow one job to be processed more than once on a machine. A modest future effort will extend the CSANN approach to practical JSSP-type situations. The application of CSANN-II can also be extended to stochastic and dynamic JSSPs, which are closer to real-world scheduling problems. To address these JSSPs, the structure of CSANN-II may be more flexible and change over time according to the stochastic or dynamic conditions of the JSSP.

Acknowledgments The authors would like to thank the anonymous associate editor and reviewers for their thoughtful comments and constructive suggestions. This work was supported in part by the Engineering and Physical Sciences Research Council (EPSRC) of UK under Grant EP/E060722/01 and in part by the National Nature Science Fundation of China under Grant 60821063 and National Basic Research Program of China under Grant 2009CB320601.

References

Akyol, D. E., & Bayhan, G. M. (2007). A review on evolution of production scheduling with neural networks. *Computers & Industrial Engineering*, 53(1), 95–122.

Baker, K. R. (1974). *Introduction to Sequence and Scheduling.* New York: Wiley.

Bellman, R. E., Esogbue, A. O., & Nabeshima, I. (1982). *Mathematical Aspects of Scheduling and Applications.* Oxford: Pergamon.

Beasley, J. E. (1990). OR-Library: Distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11), 1069–1072.

Bierwirth, C., & Mattfeld, D. C. (1999). Production scheduling and rescheduling with genetic algorithms. *Evolutionary Computation*, 7(1), 1–17.

Blackstone, J., Phillips, D., & Hogg, G. (1982). A state-of-the-art survey of dispatching rules for manufacturing job shop operations. *International Journal of Production Research*, 20, 27–45.

Cheng, R., Gen, M., & Tsujimura, Y. (1996). A tutorial survey of job-shop scheduling problems using genetic algorithms, I. representation. *Computers & Industrial Engineering*, 30(4), 983–997.

Cheng, R., Gen, M., & Tsujimura, Y. (1999). A tutorial survey of job-shop scheduling problems using genetic algorithms, part II: hybrid genetic search strategies. *Computers & Industrial Engineering*, 36(2), 343–364.

Conway, R. W., Maxwell, W. L., & Miller, L. W. (1967). *Theory of scheduling.* Reading: Addison-Wesley.

Cormen, T. H., Leiserson, C. E., & Rivest, R. L. (1990). *Introduction to Algorithms.* Cambridge: MIT Press.

Dubois, D., Fargier, H., & Prade, H. (1995). Fuzzy constraints in job-shop scheduling. *Journal of Intelligent Manufacturing*, 6, 215–234.

Erschler, J., Roubellat, F., & Vernhes, J. P. (1976). Finding some essential characteristics of the feasible solutions for a scheduling problem. *Operations Research*, 24(4), 774–783.

Fang, H.-L., Ross, P., & Corne, D. (1993). A promising genetic algorithm approach to job-shop scheduling, rescheduling and open-shop scheduling problems. In *Proceedings of the 5th international conference on genetic algorithms* (pp. 375–382).

Foo, S. Y., & Takefuji, Y. (1988a). Neural networks for solving job-shop scheduling: part 1. problem representation. In *Proceedings of the 2nd IEEE international joint conference on neural networks* (Vol. 2, pp. 275–282).

Foo, S. Y., & Takefuji, Y. (1988b). Neural networks for solving job-shop scheduling: part 2. architecture and simulations. In *Proceedings of the 2nd IEEE international joint conference on neural networks* (Vol. 2, pp. 283–290).

Foo, S. Y., Takefuji, Y., & Szu, H. (1994). Job-shop scheduling based on modified Tank-Hopfield linear programming networks. *Engineering Application of Artificial Intelligence*, 7(3), 321–327.

Fox, M. S., & Zweben, M. (1993). *Knowledge-based scheduling.* San Manteo: Morgan Kaufmann.

French, S. (9182). *Sequencing and scheduling: An introduction to the mathematics of the job-shop.* New York: Wiley, 1982.

Garey, M. R., Johnson, D. S., & Sethi, R. (1976). The complexity of flowshop and job-shop scheduling. *Mathematics of Operational Research*, 1(2), 117–129.

Giffler, B., & Thompson, G. (1960). Algorithms for solving production scheduling problems. *Operations Research*, 8, 487–503.

Graham, R. L., Lawler, E. L., Lenstra, J. K., Rinnooy Kan, A. H. G. (1979). Optimization and approximation in deterministic sequencing and scheduling: a survey, *Annals of Discrete Mathematics*, 4, 287–326.

Graves, S. C. (1981). A review of production scheduling. *Operations Research*, 29(24), 646-675.

Hart, E., & Ross, P. (1998). A heuristic combination method for solving job-shop scheduling problems. In *Lecture notes in computer science: Vol. 1498. Proceedings of the 5th international conference on parallel problem solving from nature (PPSN V)* (pp. 845–854). Berlin: Springer.

Hart, E., Ross, P., & Corne, D. (2005). Evolutionary scheduling: a review. *Genetic Programming and Evolvable Machines*, 6, 191–220.

Haupt, R. (1989). A survey of priority-rule based scheduling problem, *OR Spektrum*, 11, 3–16.

Haykin, S. (1999). *Neural Networks A Comprehensive Foundation (2nd ed.).* London: Prentice Hall International.

Hentenryck, P. V. (1989). *Constraint Satisfaction and Logic Programming.* Cambridge: MIT Press.

Lawrence, S. (1984). Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques. Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, Pennsylvania, PA.

Lin, S.-C., Goodman, E. D., & Punch, W. F. (1997). A genetic algorithm approach to dynamic job-shop scheduling problems. In *Proceedings of the 7th International Conference on Genetic Algorithms* (pp. 481-489).

Luh, P. B., Zhao, X., Wang, Y., & Thakur, L. S. (2000). Lagrangian relaxation neural networks for job shop scheduling, *IEEE Transactions on Robotics and Automation*, 16(1), 78–88.

Vázquez, M., & Whitley, L. D. (2000a). A comparison of genetic algorithms for the static job shop scheduling problem. In *Proceedings of the 6th International Conference on parallel problem solving from nature (PPSN VI)* (pp. 303–312).

Vázquez, M., & Whitley, L. D. (2000b). A comparison of genetic algorithms for the dynamic job shop scheduling problem. In *Proceedings of the 2000 genetic and evolutionary computation conference* (pp. 1011–1018).

Willems, T. M. (1994). Neural networks for job-shop scheduling. *Control Engineering Practice*, 2(1), 31-39.

Willems, T. M., & Brandts, L. E. M. W. (1995). Implementing heuristics as an optimization criterion in neural networks for job-shop scheduling. *Journal of Intelligent Manufacturing*, 6, 377–387.

Yang, S., & Wang, D. (2000). Constraint satisfaction adaptive neural network and heuristics combined approaches for generalized job-shop scheduling. *IEEE Transactions on Neural Networks*, 11(2), 474–486.

Yang, S., & Wang, D. (2001). A new adaptive neural network and heuristics hybrid approach for job-shop scheduling. *Computers & Operation Research*, 28(10), 955–971.

Yang, S. (2005). An improved adaptive neural network for job-shop scheduling. In *Proceedings of the 2005 IEEE international conference on systems, man and cybernetics* (Vol. 2, pp. 1200-1205).

Yang, S. (2006). Job-shop scheduling with an adaptive neural network and local search hybrid approach. In *Proceedings of the 2006 IEEE international joint conference on neural networks* (pp. 2720–2727).

Yu, H. (1997). *Research of intelligent production scheduling methods and their applications*. PhD Thesis, Northeastern University, China.

Yu, H., & Liang, W. (2001). Neural network and genetic algorithm-based hybrid approach to expanded job-shop scheduling. *Computers & Industrial Engineering*, 39, 337–356.