Resource Management for Business Process Scheduling in the Presence of Availability Constraints

JIAJIE XU, Soochow University, Swinburne University of Technology
CHENGFEI LIU, Swinburne University of Technology
XIAOHUI ZHAO, University of Canberra
SIRA YONGCHAREON, Auckland University of Technology
ZHIMING DING, Chinese Academy of Sciences

In the context of business process management, the resources required by business processes, such as workshop staff, manufacturing machines, etc., tend to follow certain availability patterns, due to maintenance cycles, work shifts and other factors. Such availability patterns heavily influence the efficiency and effectiveness of enterprise resource management. Most existing process scheduling and resource management approaches tend to tune the process structure to seek better resource utilisation, yet neglect the constraints on resource availability. In this article, we investigate the scheduling of business process instances in accordance with resource availability patterns, to find out how enterprise resources can be rationally and sufficiently used. Three heuristic-based planning strategies are proposed to maximise the process instance throughput together with another strategy based on a genetic algorithm. The performance of these strategies has been evaluated by conducting experiments of different settings and analysing the strategy characteristics.

CCS Concepts: • **Information systems → Enterprise resource planning**; • **Applied computing → Enterprise resource planning**

Additional Key Words and Phrases: Resource management, resource planning, business process scheduling, business process management

ACM Reference Format:
Jiajie Xu, Chengfei Liu, Xiaohui Zhao, Sira Yongchareon, and Zhiming Ding. 2016. Resource management for business process scheduling in the presence of availability constraints. ACM Trans. Manag. Inf. Syst. 7, 3, Article 9 (October 2016), 26 pages.
DOI: http://dx.doi.org/10.1145/2990197

1. INTRODUCTION

Resource planning for business processes is a classical issue for enterprise operation management. The recent global financial crisis further urges enterprises to seek cost-effective utilization of resources including human resources and automated resources,

Note that the work reported in this article is based on our preliminary work on business process scheduling with resource availability constraints [Xu et al. 2010], with significant extensions and improvements on resource allocation model for business processes, extensive experimental studies on performance, and complexity analysis of the algorithms.

Authors' addresses: J. Xu, Department of Computer Science and Technology, Soochow University, Suzhou 215006, China; email: xujj@suda.edu.cn; C. Liu, Department of Computer Science and Software Engineering, Swinburne University of Technology, Hawthorn, VIC 3122, Australia; email: cliu@swin.edu.au; X. Zhao, School of Information Systems and Accounting, University of Canberra, Bruce, ACT 2617, Australia; email: xiaohui.zhao@canberra.edu.au; S. Yongchareon, Department of IT and Software Engineering, Auckland University of Technology, Auckland 1010, New Zealand; email: sira.yongchareon@aut.ac.nz; Z. Ding, NFS, Institute of Software, Chinese Academy of Sciences, Beijing 100864, China; email: zhiming@nfs.iscas.ac.cn.

DOI: http://dx.doi.org/10.1145/2990197

such as workers, machines, work centers in a manufacturing facility; doctors, nurses, special medical equipment in a hospital, etc. So far, many classical algorithms [Avanes and Freytag 2008; Decker and Schneider 2007; Iosup et al. 2007; Rood and Lewis 2008; Yarkhan and Dongarra 2002; Yu and Buyya 2006; Zomaya and Teh 2001] have been proposed, and most of them focus on run-time resource scheduling. Such algorithms can support rational resource utilization with real-time response, but they fail to guarantee timely completion of scalable process instances with complex dependencies and constraints. In reality, process instance completion in time is a crucial factor in enterprise decision making. Take a manufacturer for example, when some new customer orders come in, it is important for a manager to know whether all the orders can be fulfilled before deadlines under the current resource condition. If not, the manager has to seek further information to prioritize orders, adjust the original resource plan to cater for the business process instances for new orders, etc. Therefore, in the scenarios that business process instances and the needed resources are known before execution, it is crucial to investigate build-time resource planning for quality decision support.

Apart from the deadline constraint in classical resource management for business processes, a high level of process instance concurrency is sought after in most application scenarios. This is because parallel process instances lead to high instance throughput and efficient resource utilisation for resource management and process scheduling. However, it is not always realistic to increase the process instance concurrency by pushing all resources together to serve instances, because resources themselves are often available in certain time periods in practice, e.g., a worker is not supposed to work after hours. Thus, to comprehensively schedule business processes, factors of resource availability and capability, process task dependency, instance deadline, and inter-influence among them should all be taken into account. This article incorporates resource availability constraints and process structures into build-time process scheduling and devises a comprehensive framework for maximising process instance throughput with a set of strategies.

Figure 1 shows an example of the process scheduling scenario that can be found in many applications in business process management, such as product ordering in manufacturing companies, service booking in hospitals or government agencies, etc. Given a set of received orders from customers, an enterprise is expected to handle the orders according to a certain business process. Due to the limitation of resource capacity, only part of the process instances may be finally processed. From the view of decision making, it is essential for enterprises to plan the scheduling of process instances for optimal resource utilisation at build time, so that they can know how to use the available resources rationally to maximize the profit with the guarantee that the accepted orders can be indeed fulfilled. Some practical concerns must be considered in such planning on process instances: First, the processing of tasks in a process instance must comply with the temporal dependencies according to process structure. Second, process scheduling must follow the availability constraints of resources [Russell et al. 2005], e.g., work shifts of staff members or machine maintenance caused by down times. Third, the selection of resource for scheduling the business processes must follow such constraints as resource capability, privacy requirements, etc. Also, the completion time of each instance may be required to be in certain period due to constraints such as business schedule, payment arrangement from the customers. For example, in Figure 1, *instance1* can only start after time point *st1*, and the processing must be finished before deadline *et1* set by customers. The execution of tasks in *instance1* must follow its process structure, and each task can only be assigned to a capable resource according to its task type. Therefore, process scheduling is subject to all the above issues as well as their inter-influence. The problem is how to find out the optimal planning result efficiently (as shown in Figure 1) while satisfying all the constraints. Some heuristics may be used because this problem is computationally hard. Intuitively, a

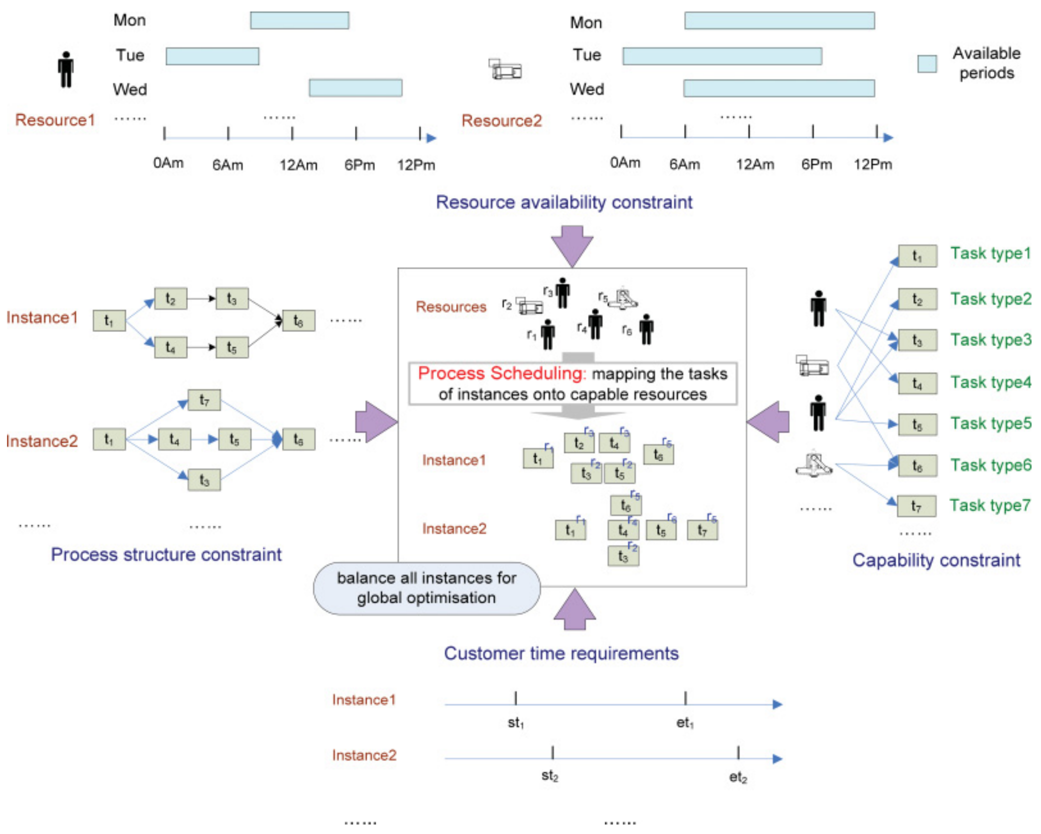

Resource1
Mon
Tue
Wed
0Am
6Am
12Am
6Pm
12Pm
Resource2
Available periods
Resource availability constraint
Instance1
Instance2
Process structure constraint
Resources
Process Scheduling: mapping the tasks of instances onto capable resources
balance all instances for global optimisation
Task type1
Task type2
Task type3
Task type4
Task type5
Task type6
Task type7
Capability constraint
Customer time requirements
st1
et1
st2
et2

Fig. 1. Process scheduling.

greedy strategy is to rescue the instances from the ones that are most likely to exceed their deadline. We can also plan the resource allocation for different instances in a balanced way, and the optimisation may be carried out in different criteria. Due to the limitation of heuristics, each strategy may have an advantage or a disadvantage in different scenarios. Thus, the evaluation of the proposed strategies must be applied in varied conditions for practical use.

As a classical topic of enterprise management, process scheduling has drawn a lot of attention from the research community. A typical category of this field is job-shop scheduling, and so far has possessed many solutions, such as genetic algorithm [Jensen 2003; Wu et al. 2004; Yoo 2009], ant colony optimisation [Heinonen and Pettersson 2007; Seckiner and Kurt 2007], neural networks [Xie et al. 2005], and other heuristic approaches [Jin et al. 2009; Jose et al. 2008; Yarkhan and Dongarra 2002]. But these solutions cannot be used in business process management systems directly because the possible dependencies between different tasks are not considered. Another category is workflow scheduling/planning, which considers the workflow structure in resource allocation. Yu et al. in Yu and Buyya [2005] compared some significant workflow scheduling algorithms and systems [Ashraf and Erlebach 2010; Chen et al. 2010; Decker and Schneider 2007; Langguth and Schuldt 2010; Senkul and Toroslu 2005; Yarkhan and Dongarra 2002; Yu and Buyya 2006] under the Grid environment. These approaches mainly deal with the run-time workflow scheduling problems. However, they do not consider build-time workflow planning (where the resource situations tend to be stable and follow some pattern) that can provide essential information for decision making (e.g., production plan and negotiation with customers), because the patterns of resource availability are not fully considered. In our previous work [Xu et al. 2009], we assume that the availability of resources can be tailored to tasks. In this article, we particularly focus on the influence of resource availability on scheduling a large number of process instances at build time. Accordingly, the scheduling confronts a much larger search space, because assigning a task to a resource may correspond to a large number of possible time slots to place. To tackle this problem and further improve the performance of process scheduling, we propose a comprehensive scheduling framework, which deals with the scheduling of process instances of multiple business processes under resource availability constraints. This work contributes to the current process scheduling paradigm in the following aspects:

—Resource availability constraints are taken into account in scheduling process instances;
—Based on a genetic algorithm, an approach is proposed to plan process instances at build time for maximising the number of instances to be successfully scheduled;
—Three heuristic based methods are proposed to enhance the efficiency of process instance scheduling in different criteria;
—The performance of the proposed strategies is evaluated with an experimental study, and the influences of performance from different resources and process instance settings are analysed.

The rest of this article is organised as follows: Section 2 introduces a model that includes the key notions for characterising resources and process instance scheduling, and formally defines our problem. Three strategies for process scheduling are proposed and discussed in Section 3. Section 4 introduces a process scheduling strategy using genetic algorithm that can find near optimal results. Section 5 presents an experimental study to evaluate and compare these three strategies. Section 6 reviews the related work and discusses the advantages of our approach. Lastly, concluding remarks and future works are given in Section 7.

2. MODEL AND PROBLEM DEFINITION

In this section, we first present a model comprising resources, tasks, process instances and resource allocation. Based on the definitions of these concepts, we formally define the problem to solve as follows:

Definition 1 (*Resource*). Resource is used to perform tasks defined in a business process. A resource r satisfies time availability constraints defined by a sequence of available time periods denoted as AP. For each available time period $ap = (t_s, t_e) \in AP$, resource r is available for allocation from time t_s to t_e.

Definition 2 (*Task*). A task in a business process can be executed by a set of resources denoted as $Res(t)$. For each resource $r \in Res(t)$, the time for executing a task t may be different. We define $time(t, r)$ as the time duration required by a resource r to perform a task t.

Definition 3 (*Resource Slot*). Resource slot measures a time duration (within the available time periods) of a particular resource. Given a slot slt of a resource r in a time duration from st to et, then there exists available time period $ap = (t_s, t_e) \in r.AP$ such that $t_s \leq st$ and $t_e \geq et$. The resource of this slot may be *available* for use in this duration (from st to et), or has been *assigned* to perform a task for a business process.

Definition 4 (*Process Instance*). A process instance ins has a task set T and an edge set E, which defines the dependency between tasks. An edge $e(t_i, t_j) \in E$ indicates that a task t_j can only start after a task t_i finishes. An instance ins is also required to be

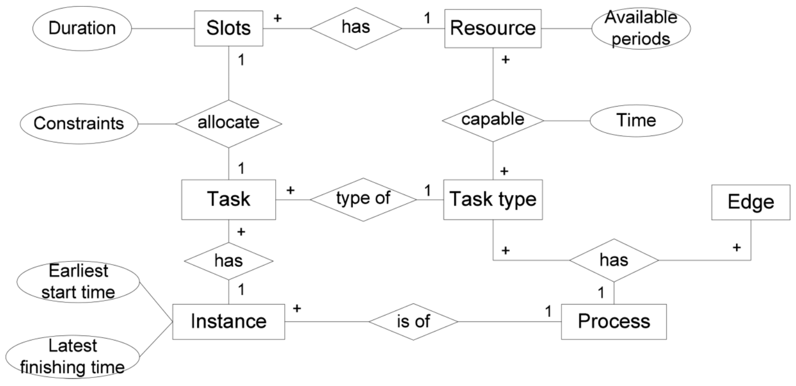


Fig. 2. Process scheduling model.

executed within a time range $[D_s(ins), D_f(ins)]$, where $D_s(ins)$ and $D_f(ins)$ refers to the earliest start time and the latest finishing time (deadline), respectively. For any two tasks $t_a, t_f \in T$, the following two relationships may exist: (1) a sequential relationship $t_a \prec t_f$ if there exists a path $e(t_a, t_b) \cup e(t_b, t_c) \cup \ldots \cup e(t_d, t_e) \cup e(t_e, t_f)$ from a task t_a to a task t_f, or (2) a parallel relationship $t_a \mathbin{//} t_f$, otherwise.

Note that, we simplify a loop structure that may appear in a process instance as a sequential execution of expected number of iterations of the tasks executed within the loop. A historical log data can be statistically analysed to estimate the execution time duration of each task and the required number of loops. If the time is underestimated, run-time resource scheduling algorithms can be used to re-plan resources for a given instance.

Definition 5 (*Allocation*). Allocation is the resource assignment to a task *t* of a process instance *ins*. An allocation $<ins, t, r, st, et>$ indicates that an available slot *slt* of resource *r* is allocated for executing task *t* of instance *ins* from time *st* to *et*. Such allocation may result in adjustment on those slots of resource *r* before or after available slot *slt*

Figure 2 illustrates the model for business process scheduling particularly on resource availability constraints. Enterprise resources may have multiple available time periods. Each instance has its own earliest start time, latest finishing time and set of tasks to be executed. Each task of the instance has a task type in accordance with the business process of the instance. Dependencies between tasks can also be derived from the edges defined in the process structure. Each resource may be capable of executing tasks of multiple types. Also, each type of task may be performed by several capable resources, and the time required for the execution of a task is determined by both the task type and the capable resource. Each task of the instance is scheduled onto a slot belonging to a particular resource, and this resource is obligated to execute this task in a time duration indicated by the assigned slot. This duration of resource slot must satisfy a set of constraints on both the resource side and the instance side. Note that, BPM resource planning faces a capacity limit (e.g., a resource can serve no more than five tasks at one time). To this end, the model can be extended to be capacity limit aware: if a resource is allowed to concurrently process a number of *n* tasks, it is converted to *n* identical resource slots for each available time period for scheduling. Based on this model, the problem can be defined as below:

Problem Statement
Given a set of resources *R*, and a number of instances *I*, we seek to find a scheduling scheme *S* (which consists of a set of allocations) to schedule instances in *I*

using resources in R such that maximal number of instances can be scheduled. During scheduling process, the following four constraints must be satisfied:

(C1) *Customer time requirement* – each instance *ins* must be executed between a required duration within a specified time range $[D_s(ins), D_f(ins)]$;

(C2) Availability constraint of resource – for each allocation $<ins, t, r, st, et> \in S$, $ap.t_s \leq st < et \leq ap.t_e$, where $ap = (t_s, t_e) \in r.AP$;

(C3) *Process structural constraints* – if a sequential relationship $t_1 \prec t_2$ is defined then a task t_2 cannot start before a task t_1 finishes;

(C4) *Conflict free* – at one time, one resource can only be used for executing one task, i.e., given any two allocations $<ins_1, t_1, r, st_1, et_1>$ and $<ins_2, t_2, r, st_2, et_2>$ on resource r, time periods $tp_1 = (st_1, et_1)$ and $tp_2 = (st_2, et_2)$ do not overlap.

3. HEURISTIC-BASED SCHEDULING STRATEGIES

Business process and workflow scheduling are known as classical NP-complete problems [Yu and Buyya 2006; Fechner et al. 2008; Liu et al. 2011]. When resource availability constraints need to be additionally considered, finding the optimal solution to the problem defined above is even computationally harder. As such, near-optimal strategies based on reasonable heuristic rules are sought after. In this article, we propose three heuristic-based algorithms to find reasonable scheduling results in an efficient way.

The basic idea of the three algorithms is to save "dangerous" instances and urgent tasks, which are subject to the time gap between optimistic finishing time and deadline. We first apply a so-called optimistic pre-allocation scheme to all instances. This scheme only satisfies constraints C1, C2, and C3 while ignoring constraint C4 at the beginning. For each allocation of the optimistic scheme, the most efficient resource is used, i.e., the one that enables each task to finish at the earliest opportunity. This pre-allocation sets a basis for the following process scheduling. After the pre-allocation, for all instances, we can initialize an important feature called the *time gap*, denoted as $g(ins)$ for instance *ins* defined in Formula 1. The gap is calculated as the difference between $D_f(ins)$ (i.e., the deadline of *ins*) and the optimistic finishing time of *ins* in the optimistic allocation. We know that this initial time gap is obtained by allowing conflict resource allocation to occur. The scheduling process then is to re-allocate tasks such that constraint C4 can also be satisfied.

$$g(ins) = Df(ins) - max(et| < ins, t, r, st, et > \in \text{optimistic allocation of } ins) \tag{1}$$

The time gap is an important indicator for the priority of instance allocation. An instance *ins* with a smaller time gap is considered to be more dangerous, and it cannot be scheduled if a gap $g(ins)$ is negative because the most efficient resources are used. Our first scheduling strategy is based on the rule of iteratively saving the most dangerous instance *ins* which owns the minimum value in $g(ins)$. This strategy operates in a depth-first manner and falls into the category of greedy algorithm since only local optimisation is applied to one instance (i.e., the most dangerous one) at a time. Sometimes, this strategy is practical because it guarantees that an instance is scheduled once it is processed. However, allowing one instance to go through may be at the cost of sacrificing other instances.

Given the limitation of the first strategy which is based on local optimisation, we propose some holistic strategies that consider global optimisation. A holistic approach operates in a breadth-first manner. Instead of scheduling one instance at a time, it allocates resources to a task of an instance at a time. Compared with the greedy strategy, it focuses more on dependencies among instances, and gives chances to all instances. Thus, process instances can be scheduled in a more balanced way. In most cases, a balanced scheduling approach makes more instances schedulable. However, when

Table I. Scheduling Strategies

Strategy	Priority
DM – Depth first/Min gap	Most dangerous instance (instance with the minimum time gap)
BL – Breadth first/Dynamic local optimisation	Most urgent task (mainly determined by the task in the most dangerous instance)
BG – Breadth first/Dynamic Global optimisation	Least penalised task (the allocation of the task that results in the minimum time gap increases of all instances and other factors)

available resources are limited (i.e., not sufficient to schedule the given instances), a holistic approach may cause more non-schedulable instances compared with the greedy strategy. In the holistic approach, we would like to allocate a resource to the current task at an instance at a time. In order to decide the instance to be selected, we design two strategies. The first strategy is to select the most dangerous instance according to some criteria, including whether it owns the minimum time gap, the number of unscheduled tasks, etc. We call the current task in such an instance the most urgent task. This strategy is different from the greedy strategy because after the allocation, the time gap for other instances will be adjusted to use the remaining available resources in an optimistic way, and the instance with minimum time gap may be changed to another instance after the adjustment. However, it does bear similarity with the greedy strategy so we call it a *dynamic local optimization strategy*. The second strategy is *dynamic global optimization*. A set of holistic rules are defined based on the penalty calculated from all instances, including the summation of the gap increases of all instances. This strategy chooses the instance with the minimum penalty to schedule. Table I summarises the three strategies introduced. We provide the details of each of them in Sections 3.2, 3.3, and 3.4, respectively.

3.1. Resource Data Structure

Before we discuss the three scheduling strategies, we first describe the resource data structure. Each resource r has a set of available time periods in AP. Each time period $ap \in AP$ has a set SLT, which consists of a set of resource slots within the available period. Slots are created to store the usage information of a resource in particular time duration.

Initially, each available period $ap = (t_s, t_e) \in AP$ has a single time slot slt, with start time $slt.st = t_s$, end time $slt.et = t_e$, and the initial status $slt.status =$ ‘available’. After a few allocations, slt may be spitted into several slots ($slt_1, \ldots, slt_{i\text{-}1}, slt_i, slt_{i+1}, \ldots$). When resource allocation is made on slt_i for the time period (st, et), slt_i may be partitioned into more slots. A new slot slt_{i1} is created to replace slt_i, with $slt_{i1}.st = st$, $slt_{i1}.et = et$, and $slt_{i1}.status =$ ‘assigned’. The allocating information on the instance and the task is also recorded in slt_{i1}. If $slt_i.st < st$, we do a leftward split by adding a new *available* slot slt_{i2} for the period ($slt_i.st$, st) in ap. If slt_i has a left adjacent slot $slt_{i\text{-}1}$ which is also available, slt_{i2} and $slt_{i\text{-}1}$ will be merged into a single available slot. Similarly, a rightward split may be made. Sometimes we need to re-schedule previous allocations to optimise the use of resource. Re-scheduling requires frequently retrieving left and right adjacent slots. Adjacent slots can be retrieved by function $prev(slt)$ for left adjacent slot and $next(slt)$ for right adjacent slot of slot slt, respectively.

3.2. DM Scheduling Strategy – Depth-First/Min Gap

This strategy is based on a greedy algorithm for process scheduling. Resource allocation is applied to the most dangerous instance one by one. The allocating sequence is in a descending order of instance time gap. Given a set of instances, an instance *ins* with the minimal time gap has the least room to delay. If the resources are allocated to other instances first, this instance is most likely to be affected, i.e., re-allocation even using the best available resources may cause it to exceed its deadline D_f (*ins*). Therefore, we set the highest priority to this instance. The algorithm works as follows: First, it pre-allocates all instances in the optimistic way (satisfying C2 and C3), and each instance has an initial time gap to deadline; Then it selects the instance with minimum time gap and keeps the optimistic allocations for the instance; Once an instance is scheduled, as resources allocated to the instance may be requested by other instances, it has to adjust those affected allocations of other instances by using the remaining resources to guarantee C4; It does this adjustment also in an optimistic way; After that, it continues to select the instance with minimum time gap among the un-scheduled instances; The selection and scheduling of the instance with the minimum time gap and allocation adjustment are repeated until none of the remaining instances are schedulable; Finally, it attempts to re-schedule tasks in the scheduled instances if there is room for the purpose of rescuing other un-schedulable instances. The detail of the algorithm is given below.

3.2.1. Initial Optimistic Allocation. In Algorithm 1, Lines 1–3 pre-allocate all instances and calculate their time gap to deadline before allocation, without considering the issue of resource conflict (constraint C4). It provides the best case scenarios for all the instances. Given an instance set I and a resource set R, pre-allocation is based on the function *optimisticAlloc*(*ins*) for each instance *ins* in I (Lines 19–31), where constraints C2 and C3 are satisfied. It first finds the current task t (or the next task to schedule) of *ins* by calling function *getTsk*(*ins.T*) in Line 22, and then calculates the minimal end time of t with available resources using function *getMin*(t, R) (Line 23). If the end time of t exceeds deadline (violating constraint C1), this instance is dropped out as it is definitely non-schedulable (Line 25). Otherwise, t is allocated with the most efficient resource (Line 27). This pre-allocation procedure continues until all instances have been processed.

3.2.2. Resource Allocation. Based on the time gap of un-scheduled instances, resource allocation becomes not too difficult. Basically, we use the criterion of time gap to deadline g(*ins*) to evaluate the priority of instances. The less the time gap between the finishing time to deadline is, the more dangerous this instance is. Hence, the resource allocation gets a higher priority. The most dangerous instance i_s can be detected by function *ergt*(I) (Line 5), and then the resource allocation table is updated using the resource data structure described in 3.1 (Line 6), and this instance is successfully scheduled. However, other instances may conflict with i_s (constraint C4 violated). Therefore, we need to conduct necessary adjustment to the optimistic allocation of its conflicting instances (Lines 7–14) before removing this allocated instance from the scheduling list (Line 15).

3.2.3. Optimistic Allocation Adjustment. After allocating instance *ins*, the optimistic allocation of other instances must be adjusted due to the change of resource availability. We only need to adjust the optimal allocations of some instances, which use any resource slot occupied by instance *ins*. The adjustment can be made by selecting the most efficient slot from the remaining available resources. Optimistic adjustment is made in Lines 7–14 of Algorithm 1. For each remaining instance *ins'*, set T of the tasks conflicting with previous allocation (violating constraint C4) is generated in Line 8, and these tasks need to be re-allocated. For each task t in T, the new finishing time of optimistic allocation is re-calculated in Line 10. If the updated finishing time is within the deadline, optimistic allocation is re-applied for t of *ins'* using function *OptUpd*(*ins'*, t). Otherwise, this instance is dropped out (Line 11). Steps in 3.2.2 and 3.2.3 are repeated until all instances are processed, i.e., either scheduled or dropped out (the while loop in Lines 4–16).

ALGORITHM 1: DM Strategy

Input:	I	-	instance set
	R	-	resource set
Output:	*allocT*	-	allocation table

```
for each $ins \in I$
   $optimisticAlloc(ins)$;
end for;
while($I \neq \phi$)
   $ins_s = ergt(I)$;
   $allocT = allocT + ins_s.Optalloc$;
   for each $ins' \in I$
      $T = conf(ins', ins_s.Optalloc)$;
      for each $t \in T$
         $t.minet = getMin(t, R)$;
         if ($t.minet \leq D_f(ins')$) then $OptUpd(ins', t)$ else $drop(ins')$ end if;
         $T = T - \{t\}$;
      end for;
   end for;
   $I = I - \{ ins_s \}$;
end while;
$alloc\_reschedule(I)$;
return $allocT$;
function $optimisticAlloc(ins)$
   $T_1 = ins.T$;
   while ($T_1 \neq \phi$)
      $t_1 = getTsk(T_1)$;
      $t_1.minet = getMin(t_1, R)$;
      if ($t_1.minet > D_f(ins)$)
         $drop(ins)$;
      else
         $<t_1, \mathrm{s}, tp> \rightarrow Optalloc(ins)$;
      end if;
      $T_1 = T_1 - \{ t_1 \}$;
   end while;
end function;
function $alloc\_reschedule(I)$
   for each $ins = getIns(I)$;
      $minet = MinEt\_reschedule(ins)$;
      if ($minet \leq D_f(ins)$)
         $R = reschedule(R, ins)$;
         $allocT = allocT + ins_s.Optalloc$;
      else
         $drop(ins)$;
      end if;
   end for;
end function;
```

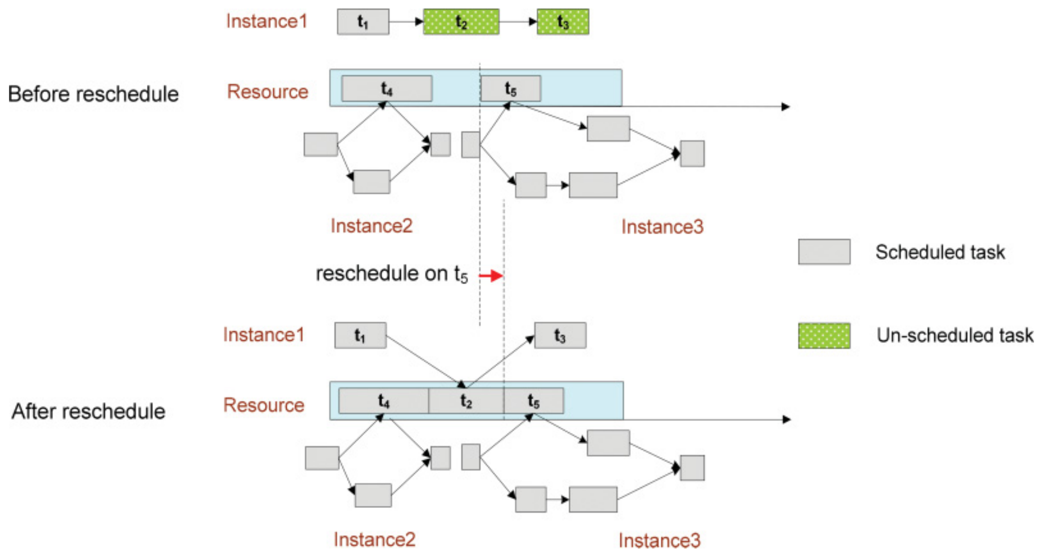


Fig. 3. Task re-scheduling.

3.2.4. Allocation Adjustment of Scheduled Instances. Previously, most efficient available resources are used in each scheduled instance by optimistic pre-allocation and optimistic allocation adjustment. In this step, we check allocations of scheduled instances to see if there is room for re-scheduling these allocations in order to rescue some un-scheduled instances. The precondition of allocation re-scheduling is that it will not affect the allocation. This is done by function *alloc_reschedule*(I) (Lines 32–42 in Algorithm 1). For example, in Figure 3, initially task t_2 of *instance1* is un-schedulable because the only capable resource is assigned to execute t_4 of *instance*2 and t_5 of *instance*3. However, after re-scheduling on t_5, *instance*1 can be finally saved. Among un-schedulable instances, an instance *ins* with the minimum exceeded distance *ed*(*ins*) from deadline to current finishing time is most likely to be saved. Thus, we iteratively select an instance *ins* with the minimum value of *ed*(*ins*) to rescue using function *getIns*(I) (Line 33).

Next, we explore how to save an instance by re-scheduling allocated tasks: For each task t of instance *ins* in execution order, from the perspective of constraint C2, we seek to make task t to finish the earliest through re-scheduling. First, among the capable slot candidates prior to current best slot of t while keeping dependencies relevant to t are satisfied, we select: (1) top 10 in time duration (more likely to be large enough after re-scheduling); (2) top 10 in availability (t is likely to finish earlier). For each slot candidate *slt* in the time sequence, we check if the adjacent slots can be moved around to enlarge *slt* until t can be executed by this slot. To reduce computational complexity, re-scheduling is not permitted to conflict with the allocated instances according to constraint C4. Technically, cascading leftward (rightward) moves may occur to those adjacent slots on the left (right) side for finding a sufficient slot. If t can be eventually executed by *slt*, we record the re-scheduling operations (not apply them yet). When all tasks of *ins* have been checked (i.e., C1 and C4 can be satisfied), we update the minimal finishing time of *ins* according to the new plan after re-scheduling (Line 34). If the deadline can be reached, we apply the recorded re-scheduling operations (re-scheduling operations on *ins* and relative slots) to save *ins* using function *reschedule*(R, *ins*) in Line 36. Otherwise, this instance is un-schedulable and dropped out (Line 39).

After the allocation adjustment of scheduled instances, the total number of scheduled instances can be obtained from the allocation table and the success rate can be computed and returned together with the allocation table (Line 18).

3.3. BL Scheduling Strategy – Breadth-First/Dynamic Local Optimisation

BL strategy attempts to plan resources for instances in a holistic way yet using dynamic local optimisation. As discussed, DM strategy allocates resources to one instance at a time. On the other hand, BL strategy allocates resources to one task of one instance at a time. In this way, it balances all instances by giving them the same chance for occupying resources. Among those current tasks competing for a resource, the allocation will be only made to schedule the most urgent task, and task urgency can be evaluated by certain criteria on the instance to which this task belongs, as shown in Formula 2. Specifically, in each round, we select the next available resource slot for allocation, and this resource is supposed to be used for the most urgent task according to a set of heuristic rules for local optimisation. The strategy is illustrated in Algorithm 2. Initially, Lines 1–3 use the result of pre-allocation from function *optimisticAlloc*(ins) introduced in Section 3.2.1. The function guarantees that constraints C1, C2, and C3 can be satisfied. Then, instance scheduling is performed based on an iterative approach using three steps (discussed in Sections 3.3.1, 3.3.2, and 3.3.3) until no remaining instance to be scheduled.

3.3.1. Selecting Resource and Generating Candidate Task Set. We select a resource slot *slt* that is the first in use among all available resource slots. This optimistically allocated resource slot may conflict with more than one current task of other different instances. Only one of them can be allocated with the slot (i.e., satisfying constraint C4 for conflict-free allocation). In this step, we first find a conflicting task set on this resource slot. Assume task t is the earliest one using *slt*, the time period *tp* of the allocation on t is derived (Line 5). A conflicting task set T includes all un-scheduled tasks using resource slot *slt* during a period overlapping with *tp* (Line 6). In Section 3.3.2, we select the most urgent task from T based on a set of rules.

3.3.2. Resource Allocation. Given resource slot *slt* and conflicting task set T from the step described in Section 3.3.1, this step is to choose the most urgent task. In the BL strategy, the urgency of a task is determined by a set of heuristic rules about the instance containing the task.

Rule 1. The urgency of a task t for allocation is influenced by the time gap of the instance that t belongs to (from constraint C1's view). The smaller the value of the time gap, the more urgent the t for allocation becomes because the instance is more likely to exceed the deadline.

Rule 2. The number of alternative resource slots to resource slot *slt* influences the urgency of a task t. Alternative resource slots are the capable resources that t can be re-allocated while not affecting the current allocation of the instance that t belongs to. If t has many alternative resource slots, it means t has abundant allocation choices, and hence it may not be urgent for t to be allocated using the slot *slt*.

Rule 3. If a task t is not allocated and there is no alternative resource slot for t, the time gap of the instance *ins* that t belongs to may reduce from $g(ins)$ to $g(ins)'$. An instance with a higher ratio $= g(ins)'/g(ins)$ is more likely to exceed deadline if the resource is not allocated to it, and hence it has higher priority to be scheduled immediately.

For each task t of instance *ins* in conflicting task set T, we generate the alternative resource set S_a according to Rule 2 (Line 8). Based on the heuristic rules, the priority of each task for requesting this resource is calculated by function $u(t)$ (Lines 9–15). Assume $x = 1 + |S_a|$ and $r = g(ins)'/g(ins)$ for task t, its urgency of being selected is computed as follows:

$$u(t) = {}^{r}\!/_{g(ins) \times \sqrt{x}} \, (g(ins) \neq 0) \tag{2}$$

In Formula 2, the urgency of task t is inverse proportion to $g(ins) \times \sqrt{x}$, because resources tend to save tasks in dangerous instances according to Rule 1. Importantly, the task with less alternative resources has a higher priority, because it is less likely to be scheduled by alternative resources without affecting optimistic allocation (Rule 2). As the number of alternative resources may vary greatly, a square root of x is used because we seek to reduce the effect of x. In contrast, the urgency of task is in proportion to ratio r. If a missed allocation of a task to this resource can cause a dramatic decrease of the time gap from $g(ins)$ to $g(ins)'$, this task has a high value of r and is more urgent to be allocated (Rule 3). Within task set, T we select a task t_s using function $maxW(T)$ (Line 20), which returns the most urgent task with maximum value of $u(t_s)$. Then, t_s can be scheduled as optimistic allocation (Line 21).

ALGORITHM 2: BL and BG Strategies

Input:	*I*	-	instance set
	R	-	resource set
	Strg	-	strategy BL or BG
Output:	*allocT*	-	allocation table

```
1   for each $ins \in I$
2     $optimisticAlloc(ins)$;
3   end for;
4   while( $nexttsk(I) \neq \phi$ )
5     $< slt, tp> = nextslts(S)$;
6     $T = cap(slt)$;
7     for each $t[i_t] \in T$
8       $S_a = altnRS(S, t, i_t)$;
9       if ($strg$== 'BL strategy')
10        if ($S_a \neq \phi$)
11          $u(t) = fomula\ 2\ (g(ins), |S_a|, 1)$;
12        else
13          $ratio = g(ins)'/g(ins)$;
14          $u(t) = fomula\ 2\ (g(ins), 0, ratio)$;
15        end if;
16      else    //* BG strategy
17          $p(t) = fomula\ 3\ (t)$;
18      end if;
19    end for;
20    if ($strg$== 'BL strategy')   $t_s = maxW(T)$;
21      schedule $t_s$ using optimal allocation;
22    else $t_s = minP(T)$;      //* BG strategy
23      $st = getSt(t_s)$;
24      $et = getET(t_s)$;
25      $r = getRes(slt)$
26      $allocT = allocate < i_t, t, r, st, et>$;
27    end if;
28    $updOpt(I)$;
29  end while;
30  $alloc\_reschedule(I)$;
32  return $allocT$;
```

3.3.3. Allocation Adjustment. After the resource allocation in the step discussed in Section 3.3.2, allocations of some instances may be affected and required to change. Similar to the DM strategy, we adjust the affected allocations by using function *updOpt*(I) (Line 28, for C4). But if an instance becomes un-schedulable with remaining resources, its occupied resources are released. After optimistic pre-allocation, all the steps are repeated until no remaining instance is schedulable. Lastly, we try to re-schedule the allocated tasks to save other instances by using function *alloc_reschedule*(*ins*) (Line 29). The allocation table is finally returned (Line 30).

Note that business process scheduling may have conflict-of-interest constraints. As we focus on the complexity caused by the availability constraints, this article adopts a simplified resource model. To consider conflict of interest, we need to consider some changes on our two operations. The first change is on the initial optimistic allocation. Given a business process instance, we can traverse the conflict free mappings from tasks to resource slots, rather than all, to select out the best allocation. The other one is on the optimistic allocation adjustment, we can use the similar approach to select the optimal conflict-free resource slot for assignment.

3.4. BG Scheduling Strategy – Breadth-First/Dynamic Global Optimisation

The BG strategy uses a different optimisation criterion for holistic process scheduling compared with the BL strategy. Though it schedules processes for one task of one instance in each round and schedules instances in a balanced way, this strategy targets global optimisation for every allocation. Resources are used to schedule the task with minimal penalty based on all instances rather than a single instance. We propose *three* heuristic rules, based on which the penalty of each task for allocation can be calculated using a formula. In comparison, this strategy considers more impact among different instances than the previous two strategies. Given a resource slot *slt* and a conflicting task set T on *slt*, task priority is determined by the following rules:

Rule 1. When *slt* is allocated to $t \in T$ of an instance, the total time gap increase of all instances influences the task penalty. The more the total time gap increases, the more of a penalty it will get, and hence the less of a priority this task will be scheduled from the overall perspective.

Rule 2. If a task belongs to an instance with fewer un-scheduled tasks, this task gets less of a penalty or has more priority to be scheduled because we are more likely to guarantee that the instance it belongs to can be successfully scheduled.

Rule 3. The task gets more of a penalty if it results in more instances becoming un-schedulable. Each allocation is aimed to cause the least number of instances becoming un-schedulable.

This algorithm is illustrated in Algorithm 2. Based on the result of pre-allocation (Lines 1–3), we select the next available resource slot that can be first used (minimal start time) in all un-allocated instances, and then generate the task candidate set of the resource slot (Line 6). In Line 17, the penalty of task candidates is evaluated according to the heuristic rules proposed above, and the resource slot is allocated to the task with minimal penalty from the global view. Given a task t of instance *ins*, the penalty $p(t)$ for scheduling this task using resource slot *slt* is calculated as:

$$p(t) = \left(1 - \frac{1}{2x}\right) \cdot y^2 \cdot \left(\sum_{i\mathrm{ns} \in I} \left(g(i\mathrm{ns})' - g(i\mathrm{ns})\right)\right) \tag{3}$$

where x is the number of remaining tasks of *ins* including t, y is the number of un-schedulable instance resulted from the allocation of t, and $\sum_{i \in I} (g(i)' - g(i))$ is the

Table II. Complexity Comparison

Strategies	Time Complexity
DM	$max(O(m^2np), O(mnpq))$
BL	$max(O(m^2n^2p), O(mnpq))$
BG	$max(O(m^2n^2p), O(mnpq))$

total time gap increase. The penalty has a direct relationship with the total time gap increase of all instances (Rule 1). Also, the penalty is in proportion to x because, when an instance is about to finish, we tend to finish it (Rule 2). However, Rule 2 is less dominant so we design $(1 - 1/2x)$ as the coefficient in range [0.5, 1) to restrict its effect. In addition, the task penalty is also in proportion to y because the allocation should avoid affecting other instances (*Rule 3*). We emphasise its effect with y^2. In each round, task t_s with the minimal penalty is selected (Line 22), and resource slot slt is allocated to schedule this task for global optimisation (Lines 23–26). Optimistic allocations are updated after task scheduling (Line 26). Similar to the BL strategy, these three steps continue until no remaining instance needed to be scheduled.

3.5. Discussion

In this section, we discuss the performance of our three proposed heuristic-based strategies and how they are used in decision support systems. The complexities of the strategies are analysed first. Also, we compare their advantages and limitations according to the heuristics adopted.

Assume an input with m process instances and each instance has at most n tasks. Each task can be performed by no more than p capable resources. In the DM strategy, pre-allocation for all instances is conducted first (Lines 1–3 of Algorithm 1) and it runs in $O(mnp)$ time. Then resource scheduling is iteratively made for the most urgent task (Lines 5–6 of Algorithm 1), and its conflicts with the remaining instances are handled accordingly after each allocation (Lines 7–14 of Algorithm 1). These require $O(m^2np)$ time in the worst case. If no remaining instances are schedulable, we seek to apply re-scheduling on previous allocations to rescue the un-scheduled instances. The complexity of re-scheduling for one remaining instance is $O(npq)$ in the worst case, where q denotes the maximum number of tasks (of all instances) can be allocated to a resource for execution. Such adjustments may repeat at most m times. Therefore, the total complexity of the DM strategy involving the above steps is $max(O(m^2np), O(mnpq))$. In the BL strategy, pre-allocation is conducted first, then an available resource slot first in demand is selected in line 5 of Algorithm 2. For each allocation, we use up to $O(np)$ time to calculate the urgency of each task requesting this slot (Line 11 of Algorithm 2), and then allocation is made to the most urgent one. Immediately after the allocation, its conflicts are handled in $O(mnp)$ time (Line 26 of Algorithm 2). All these operations may repeat at the maximum of $m \times n$ times to schedule all instances. Last, we seek to re-schedule previous allocations to rescue the remaining un-schedulable instances (Line 29 of Algorithm 2) and it runs in $O(mnpq)$ time. Therefore, the complexity of the BL strategy is $max(O(m^2n^2p), O(mnpq))$. The difference between the BG and BL strategies is that they use different criteria in selecting a task for allocation. In the BG strategy, the time requires for calculating the penalty of each task in a candidate task set is $O(mnp)$ according to Lines 12–13 of Algorithm 2. Thus, the BG strategy runs in $max(O(m^2n^2p), O(mnpq))$ time.

According to the adopted heuristics, the DM strategy always uses resources to cater for the most urgent instance and guarantees at least this instance can be successfully scheduled. Therefore, this strategy is expected to be practical when resources are insufficient. Compared with the DM strategy, the BL and BG strategies give changes to all instances and schedule them in a holistic way. As a result, instances are allocated in balance and hence tend to succeed or fail together. In contrast to the BL strategy, which is based on such a local optimisation that resources are always assigned to the task of the most urgent instance, the BG strategy relies on global optimisation meaning that instances are allocated in a more balanced way.

The heuristic-based strategies proposed can be used to plan the enterprise resource utilisation for processing business process instances requested by a customer. They are especially useful to support an enterprise for making important decisions with regard to marketing. Through a build-time planning, managers can be informed about how to optimise their enterprise operations for maximal profit, with the guarantee that their resource requirements can be satisfied. If an instance cannot be scheduled, the enterprise needs to make a decision for either seeking an extra resource or negotiating with the customer for extending deadlines, for instance. The useful information gathered by the scheduling strategies can be explored to support enterprise decision making, e.g., what kind of resources is especially needed, which process instances can be easily scheduled with minimum effort. Some decisions can be made based on new resource recruitment or work shift adjustment. It is worth noting that our proposed process scheduling strategies can be re-applied several times. This way, we can cater for some situations such as giving priority to a VIP customer or handling an exception when it occurs.

4. GENETIC-ALGORITHM-BASED SCHEDULING STRATEGIES (GA STRATEGY)

The algorithms proposed in Section 3 are based on heuristics. Sometimes, the heuristics may discriminate some exploring cases and thus miss the most or more accurate results in spite of good efficiency performance. To overcome this shortcoming, we propose a genetic algorithm based scheduling strategy (GA strategy in short) to ensure the scheduling can achieve at least the near-optimal results. In a genetic algorithm, each candidate solution has a set of properties (i.e., its chromosomes) which can be mutated and altered during evolutions. Starting from a population of individuals (i.e., candidate solutions), each evolution selects the individuals with high fitness values to generate new individuals using three operators (selection, crossover, mutation). Through iterative evolutions, the candidate solutions keep improving in generations, and the process stops when the terminating condition is satisfied.

4.1. Encoding and Initial Population

In standard genetic algorithms, candidate solutions can be encoded and represented by an array of bits; however, this representation does not fit our scenario and thus a new encoding structure is needed. In the process-scheduling context, each candidate solution is an assignment from tasks (in all schedulable instances) to available resource slots with no overlapping. Therefore, in our GA strategy, we encode each candidate solution in an array of tuples in the structure of (i, j, k, s, e), where $i/j/k$ are the IDs of instance, task, and resource, respectively, and s and e refer to the starting and ending times of a time slot, respectively. A tuple (i, j, k, s, e) means that a task j of an instance i is assigned to an available time duration (t_s, t_e) of a resource k. Tuples appear in array in the ascending order of (i, j). Note that candidate solutions are required to meet all the constraints (C1, C2, C3, and C4) defined as part of our problem statement (in Section 2).

Given that the three heuristic based strategies can provide meaningful results efficiently, we utilize them in our GA strategy to generate the initial population, rather than by random. Specifically, we retrieve a subset of resources and a subset of process instances by random, and then run the DM, BL, and BG strategies on the top of the selected resources and instances, where each run returns a candidate solution. In this way, we can obtain a required number of candidate solutions that can be encoded as the initial population for the GA strategy.

Each evolution selects some promising candidate solutions, and thus it is important to create a fitness function for evaluating those solutions. As we aim to find the assignment that can maximize the number of instances to be scheduled, the fitness function $F(allocT)$ of a scheduling candidate solution *allocT* (i.e., an individual) can be formulated as the ratio of instances that are successfully scheduled in *allocT* (to all instances). Given a generation of individuals, we define the best possible result covered in generation G_i as fn.

$fn = max(F(alloc)|alloc \in G_i)$, where *F(alloc)* is the fitness of an individual.

4.2. Genetic Operators

Genetic operators are used in generating a next population to pursue superior individuals against the fitness function. The operators we use generally include selection, crossover and mutation.

Selection. The selection operator ensures us to preserve good individuals (i.e., candidate results) in the next population. The selection of individuals adopts a probabilistic method on top of the fitness function, i.e., individuals with greater fitness values have higher probability of being selected. In this way, we can ensure the quality of next population, while preserving the probability.

Crossover. The crossover operator means combining two existing candidate solutions to form a new one, which gives us opportunities to find a better individual. To do the crossover, we select two individuals using the selection operator. Assuming the selected individuals are $indv_1$ and $indv_2$, we scan all process instances such that: (1) if the instance is not scheduled in both individuals, it is not included in the new individual $indv_c$; (2) if the instance is scheduled in only one individual, all array tuples regarding this instance are included in $indv_c$ after the crossover; and (3) if the instance is scheduled in both individuals, we select the individual that has less edit distance with this instance, where the edit distance *eDistance*(*i, indv*, *indv'*) between two individuals *indv* and *indv'* on the *i*th instance can be calculated as follows:

$$eDistance(i,\ indv,\ indv') = \frac{\sum\limits_{(\mathrm{i},j,k,s,e)\in indv} isConflict(k, s, e, indv'),}{|indv|}$$

where $isConflict(k,\ s,\ e,\ indv')$ returns 1 if time slot (s, e) of resource s_k has been used in *indv'*, and returns 0 otherwise. Obviously, the greater value of the edit distance is, the more tasks in the instance need to be rescheduled. Therefore, in the GA strategy, we intend to select an individual with less edit distance to *indv'* on the *i*th instance. Here, we set the crossover probability to be 0.7.

By repetitively carrying out the above procedure, a new individual *indv'* can be generated, and *indv'* contains the partial results from the selected individuals. Note that *indv'* may not be a valid candidate result because conflicts may exist. Therefore, we iteratively select an instance and remove its conflicts. At each time, for the instance that has maximum number of conflicting tasks to other instances, we reschedule it if possible, or remove it from *indv'* otherwise. In this way, the new individual *indv'* becomes a valid candidate result with respect to constraints C1, C2, C3, and C4.

Mutation. In the GA method, the mutation operation changes the value of genes in a chromosome according to a mutation probability pm. In our GA strategy, it incurs changes to the schedule of a given individual. This operator helps us try more possible schedules to find out something useful, the new individual has a potential of improving the current schedule by itself or part of it (with the crossover operator). Specifically, in the mutation operator, some individuals are randomly selected, and we try to replace their schedules by a non-conflict one, if possible. In this article, we set the mutation probability to be 0.2, and the mutation runs as follows: (1) given a population, we select out some individuals from it based on the mutation possibility; (2) for each of the selected individuals, tasks are randomly selected to replace their schedules. If the selected tasks can be re-scheduled with a non-conflicting time slot and the fitness value is improved, we update the schedule. Otherwise, the schedule remains unchanged. The replaced schedule appears in the new generation.

4.3. GA-Based Scheduling Algorithm

Based on the genetic operators, the GA-based scheduling works as described in Algorithm 3. It starts from the first generation G_i, which is initialized by using the DM, BL and BG strategies on different process instance and resource settings (Line 3). Then, it uses generic operators to form a new generation *G'* on top of G_i (Lines 6–11). Specifically, selection, crossover and mutation operators are used to improve the current generation with the help of the fitness function. The new generation is formed afterwards based on the individuals in *G'*. This procedure is performed repetitively, and it stops when the number of loops reaches the given threshold (Lines 13–15) or the maximum fitness value of an individual in the new generation is satisfactory. Finally, the scheduling with the maximum fitness value is returned.

ALGORITHM 3: GA Strategy

Input:	*I*	-	instance set
	R	-	resource set
	thrds	-	threshold of success rate
Output:	*allocT*	-	allocation table

```
1   i = 1;          // the ith generation
2   maxLoops = 100
3   Generation $G_i$ = initialize(I, R);      // based on DM, BL, BG
4   fn = max(F(alloc)|alloc $\in G_i$)
5   while(fn > thrds)
6       G' = $G_i$;
7       G' $\leftarrow$ selection(G');
8       G' $\leftarrow$ crossover(G');
9       G' $\leftarrow$ mutation(G');
10      i = i + 1;
11      $G_i$ = getNewGeneration(G');
12      fn = max(F(alloc)|alloc $\in G_i$);
13      if (i > maxLoops)    then
14          break;
15      end if;
16  end while
17  allocT $\leftarrow$ the individual in $G_i$ with maximum fitness value;
18  return allocT;
```

5. EXPERIMENT

In this section, we evaluate the techniques that have been proposed in Sections 3 and 4. Specifically, our goal is to compare the four proposed approaches (the BL, BG, DM and GA strategies) in terms of: (1) the success rate of process instances, i.e., the

Table III. Experiment Parameters

Parameter	Value Range	Parameter Description
$N_{instance}$	1,000	number of process instances
N_{task}	10∼50	number of tasks per instance
$N_{resource}$	300	number of resources
N_{slot}	5∼100	number of time slots per resource
$t(r, t)$	1∼10	time required to perform task t by resource r

percentage of successfully scheduled instances to the total instances; (2) the efficiency of scheduling process instances on different scales of instance and resource; and (3) the effect of resource scarcity to algorithm performances.

5.1. Experimental Settings

Our experiments were conducted on a DELL 7010MT Elite computer with a quad-core CPU (Intel i5-3740) at 3.2GHz and 8GB RAM running Windows 7 (64-bit) operating system. The test settings include 500 resources categorised into 10 types, and 1000 process instances belonging to 20 business processes, with the parameters specified in Table III. The business processes and corresponding instances are selected from a deployed process management system named 'Qone', where each instance has 10 to 50 tasks, and the average in-degree of each task is 1.42, which reflects the task concurrency relationships of the process. A higher average in-degree value means that more subsequent tasks are allowed to be processed in parallel, i.e., a greater concurrency level may be achieved when executing the tasks in the business process. We generated 300 available resources, and for each resource r, we set parameters N_{slot} (i.e., number of available time slots of r) and $t(r, t)$ (i.e., time required to perform task t) in a normal distribution over the value range shown in Table III. Based on the same resource setting, we apply different strategies to plan available resources for the required process instances.

We constructed a temporal index using inverted lists to support efficient retrieval of capable resource slots. Specifically, a time space is partitioned into several time windows, each of which corresponds to a temporal duration (e.g., 8AM – 10AM, 11th Oct 2015). For each of the time windows, we used an inverted list that stores all resources that have at least one available time slot intersecting to this time window. Therefore, given a task of an instance, we can fetch the time windows that the task can be performed (subject to its previous and following tasks). From the union on the lists belonging to those time windows, we can derive all candidate resources for this task.

5.2. Comparison On Accuracy

We have evaluated and compared the accuracy of four proposed strategies using two test cases. In the first test case, we applied different strategies in scheduling 200, 400, 600, 800, and 1000 process instances with all 300 resources, and then compared the number of successful instances and the percentage of how many unsuccessful instances are successfully scheduled through the adjustment. Based on the same process instance settings, we further conducted the second test case, where 150 randomly selected resources are used in evaluating and comparing the performance of the proposed methods in various process instance scales.

Figure 4 shows the accuracy comparison of the DM, BG, and GA algorithms in resource settings of $N_{resource} = 300$ and $N_{resource} = 150$. According to Figure 4(a), the GA-based algorithm is a near-optimal approach, much better than all other strategies. It makes all instances scheduled when $N_{instance} < 600$. In contrast, the number of scheduled instances becomes stable when $N_{instance} = 800$, because the instances call

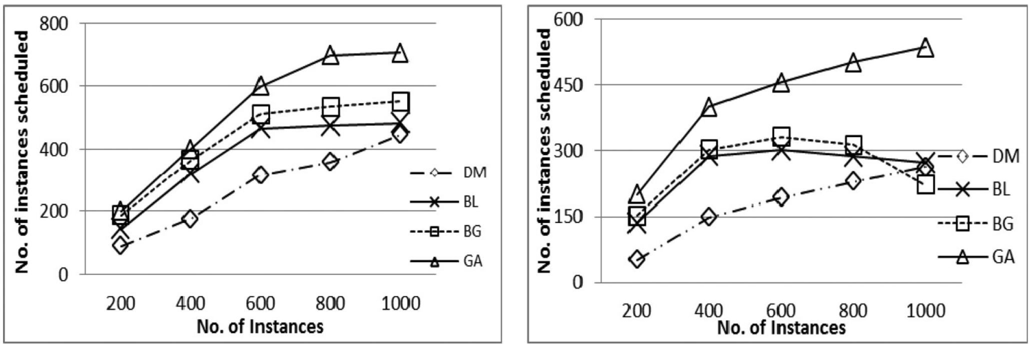


Fig. 4. (a) Accuracy comparison ($N_{resource} = 300$). (b) Success rate comparison ($N_{resource} = 150$).

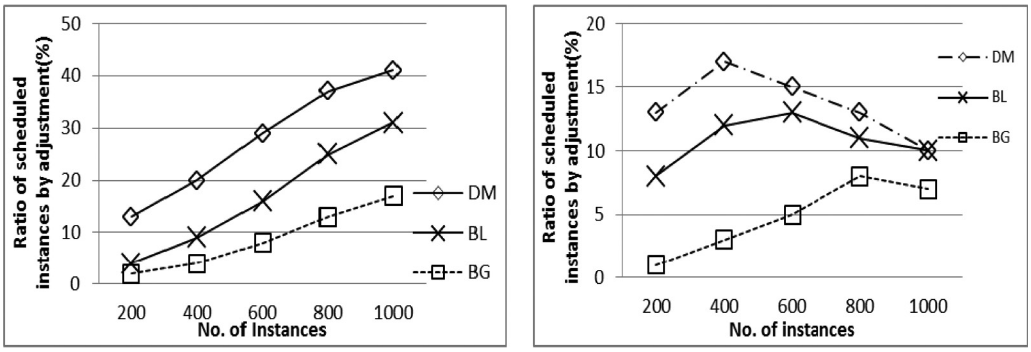


Fig. 5. (a) Adjustment comparison ($N_{resource} = 300$). (b) Adjustment comparison ($N_{resource} = 150$).

for more resources than what are available. Among the three heuristic-based methods, BG has the best performance with respect to success rate, which corresponds to the global optimization nature in its heuristic rules. When BG or BL are used, the number of successful instances becomes stable when $N_{instance} = 600$. We can easily observe that unsurprisingly the greedy-method-based DM strategy has the worst performance, and this can be explained by its local optimisation criteria.

Figure 4(b) compares the accuracy of the strategies in the resource setting of $N_{resource} = 150$, where resource becomes more limited. Similar to the previous experiment, the GA-based algorithm has the best performance with respect to accuracy. We can easily notice that with the BL and BG strategies the number of scheduled instances is decreasing when the number of given instances goes beyond 400. In contrast with the DM and GA strategies, the number keeps improving when the number of instances grows, which can be explained by the greedy nature of DM and the capability of returning near-optimal results of the GA strategy. An interesting phenomenon is that the BG strategy provides the worst result when $N_{instance} = 1000$, indicating that the global optimization oriented method does not always outperform the local optimization method, especially when resources are not sufficient.

When applying adjustments in DM, BL, and BG to increase success rate, we use the ratio of the number of instances rescued by the adjustment to the total of scheduled instances to illustrate the algorithm's contribution. Figure 5 shows the effectiveness of allocation adjustment (in DM, BL, and BG) to report the ratio of the number of instances handled by this phase to the number of all the scheduled instances. According to Figure 5, allocation adjustment helps to increase the success

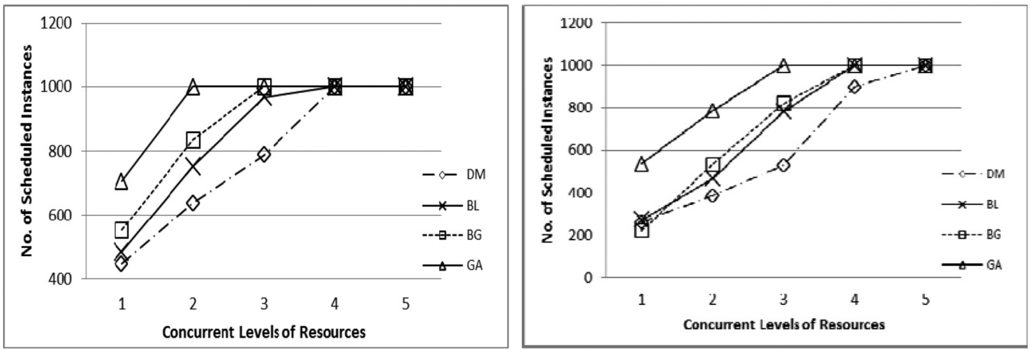


Fig. 6. (a) Concurrent levels of Resources ($N_{resource} = 300$); (b) Concurrent levels of Resources ($N_{resource} = 150$).

rate by around 10%–30% accordingly in usual cases. In Figure 5(a), in the setting of $N_{resource} = 300$, the impact of allocation adjustment to all strategies increases when the number of total instances goes up, and its effect is especially greater to DM and BL (particularly DM), because their local-optimisation-based nature leaves more room for improvement. But in the cases where resources become limited in the setting of $N_{resource} = 150$ as shown in Figure 5(b), the effect of allocation adjustment to the strategies reduces significantly. From Figure 5(b), we can observe that the adjustment is only a supplement for scheduling strategies, as it cannot rescue the result by itself when all the strategies fail ($N_{instance} > 600$, i.e., resources are extremely insufficient).

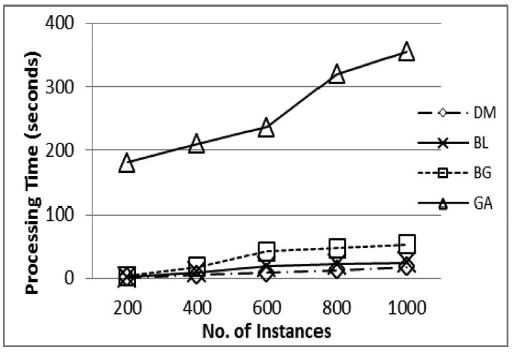
In Figure 6, we evaluated the algorithms in terms of concurrency levels based on $N_{resource} = 300$ and $N_{resource} = 150$ settings. The concurrency level of resources means how many tasks each resource can serve concurrently. In this experiment, we varied the concurrency levels in Figures 6(a) and (b), and resources in each test have the same concurrent level. We can see that the GA approach has the best performance with the increase of concurrency levels in both settings. Also, BG has a higher success rate than GL and DM when resources can be used in a greater concurrency. This is because the BG's global optimization nature makes it more likely to adapt to concurrent level of resources. We can also observe that the DM strategy cannot fully use the available resources when the concurrency level increases.

Therefore, the GA-based strategy has the best accuracy such that near-optimal results can be obtained. Among the three heuristic methods, the BG strategy has a higher success rate when resources are sufficient but performs worse when resources become tighter. This coincides with our previous analysis that the instances scheduled in this strategy tend to succeed or fail together. When resources are insufficient, the DM strategy becomes a practical scheduling strategy. This also coincides with our analysis, and is due to the nature of its depth first scheduling.

5.3. Comparison On Efficiency

In this part, we also compare the efficiency of four proposed strategies based on the two cases mentioned above. For the GA strategy, the initialization time (depends on the number of runs of DM, BL, and BG in different environment) is not included.

Figure 7 compares the computational overhead of the DM, BL, BG, and GA strategies in the $N_{resource} = 300$ setting. In accordance with Figure 7(a), the GA-based algorithm requires several minutes for processing, and it is much more time-consuming than the other heuristic-based scheduling strategies. In contrast, the DM, BL, and BG strategies are able to feedback within a minute. Figures 7(a) and (b) show that DM calls for the



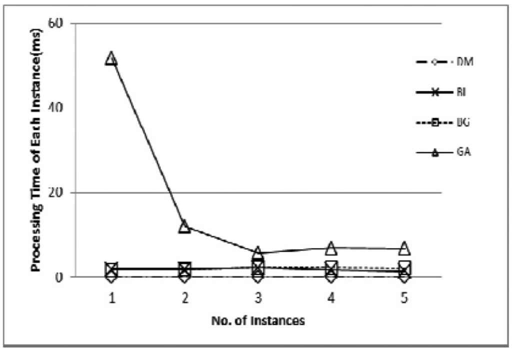
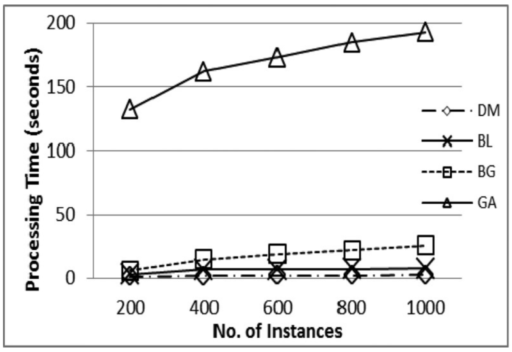


Fig. 7. (a) Cost ($N_{resource} = 300$) (b) Average Cost on a succussed instance ($N_{resource} = 300$).



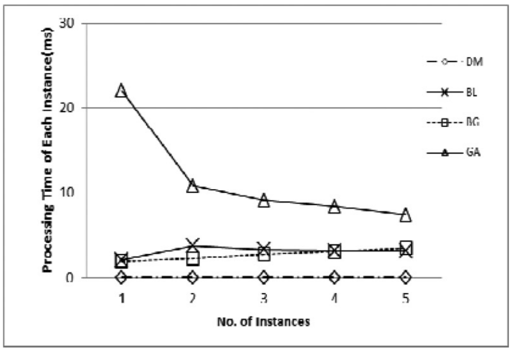
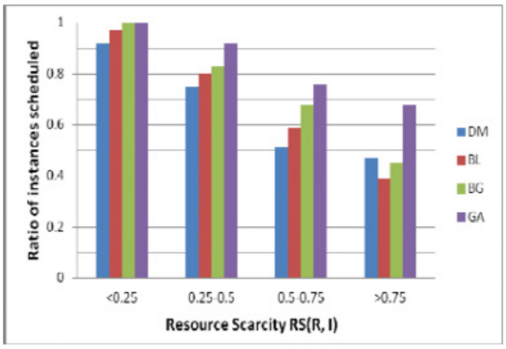


Fig. 8. (a) Cost ($N_{resource} = 150$) (b) Average Cost on a succussed instance ($N_{resource} = 150$).

least processing cost, and BG requires more cost than DM and BL in terms of both the total process time and the average time on each scheduled instance. This can be well explained by the fact that global-optimisation-based strategies like BL and particularly BG have to consider the complex dependencies between different instances, which will definitely cause the algorithm to do many more search operations.

Figure 8 evaluates and compares the computational cost of different strategies in the $N_{resource} = 150$ scenario where resources becomes more limited. Same as regular scenario cases, the GA is much more time-consuming than others, and DM is the most efficient among three heuristic-based strategies in accordance with Figures 7(a) and (b). But we can observe that the time of BG costs less time than BL when resources become insufficient ($N_{instance} > 800$). This phenomenon is caused by the early detection of resource insufficiency due to the excessive number of instances it can cope with, so it steps into allocation adjustment phase early.

To sum up, the GA-based strategy is able to provide near-optimal scheduling of instances, but it consumes several minutes or more; thus, it seems to be only suitable for build time applications as efficiency is not a big concern. For a run-time process scheduling where a good result is expected to be recommended in a short time, the BG strategy is considered the first option. This is because it tends to return a rational scheduling efficiently. If available resources are very limited, the BL strategy may outperform BG in terms of both efficiency and accuracy.



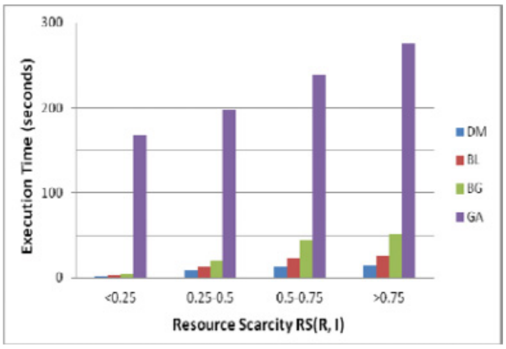


Fig. 9. (a) Accuracy via Resource Scarcity (b) Efficiency via Resource Scarcity.

5.4. Effect of Resource Scarcity

As mentioned in Section 3, the performances of the proposed algorithms are affected by the resource scarcity in experimental settings. In this section, we give a formal definition of resource scarcity as a parameter. We generated 100 test cases, and then reported their average accuracy and efficiency with varying value of the parameter (i.e., resource scarcity).

Given a set of resources R and a set of business process instances I, we use the parameter *resource scarcity*, denoted as $RS(R, I)$, to describe the abundance of available resources for scheduling those instances, and it is measured as

$$RS(R, I)= \sum_{slot\in R} \frac{\sum_{\mathrm{ins}\in I} \sum_{\mathrm{t}\in ins} \frac{time(t,slot)}{\alpha(t)}}{\tau\ (\mathrm{slot})}$$

where $\tau(slot)$ denotes the temporal duration of a resource slot *slot* overlapping with at least one instance to which *slot* can be assigned; α (t) represents the number of available resource slots can be used to execute a given task t. The lower value $RS(R, I)$, available resources are considered to be more scarcely available for processing instances I, and it is thus more likely to lead to a lower number of instances unable to be successfully scheduled. We generated 100 test cases, each case fetches part of the default business process instances, to evaluate the effect of resource scarcity to the proposed algorithms.

Figure 9 shows the average efficiency and accuracy of the proposed methods in varying range of *RS(R, I)* value (i.e., resource scarcity). As shown in Figure 9(a), the success rate of scheduled instance goes in the direct tendency to the *RS(R, I)* value. We can observe that the GA-based algorithm has the best performance robustness, and it outperforms all other algorithms at any resource scarcity level. Among the heuristic-based methods, BG tends to be the best strategy in most resource scarcity values. But it is noteworthy that the DM strategy turns out to be superior to BL and BG when the degree of resource scarcity becomes high (i.e., greater than 0.75). Figure 9(b) shows the average execution time of all proposed algorithms in different resource scarcity ranges. In comparison, DM is the most efficient strategy, and the GA-based strategy has much greater execution time than all heuristic-based method.

Figure 10 compares the results returned by the proposed algorithms to optimal assignment in varying resource scarcity. We generated 30 synthetic simple test cases that their optimal assignment can be known, and reported the accuracy (of DM, BL, BG, and GA), which is defined by the number of scheduled instances compared to the optimal assignment. From this figure, we can easily observe that the GA returns

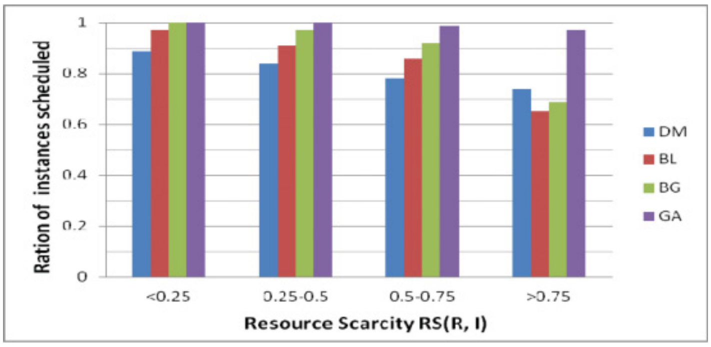


Fig. 10. Comparison to optimal assignment.

near-optimal results in high accuracy. The BG approach has fairly good accuracy when available resources are abundant (i.e., low resource scarcity cases), but it has the worst accuracy when the degree of resource scarcity becomes high.

To sum up, the GA-based strategy is able to provide near-optimal scheduling of instances, but it consumes several minutes or more and is thus only suitable for build time applications as efficiency is not a big concern. For run-time process scheduling that a good result is expected to be recommended in a short time, the BG strategy is the first option because it tends to return a rational schedule efficiently. If the available resources are very limited, the BL strategy may outperform BG in terms of both efficiency and accuracy.

6. RELATED WORK

As a classical research topic, resource allocation is about investigating how to optimally utilise available resource for achieving certain goals with constraints. The problem of mapping tasks onto given resources is known to be NP-complete [Yu and Buyya 2005, 2006; Fechner et al. 2008; Liu et al. 2011]. Due to its complexity, lots of heuristic methods have been proposed to address some classical problems, such as job shop scheduling and queuing mechanisms for operating system. In Jensen [2003], Wu et al. [2004], Yoo [2009], and Zomaya and Teh [2001], some genetic algorithms are introduced to improve quality of resource allocation generation by generation according to the fitness value. Also, several works, such as Heinonen and Pettersson [2007] and Seckiner and Kurt [2007], use the colony optimisation as heuristics for resource allocation. Derived from the Monte Carlo method, the simulated annealing method [Jin et al. 2009; Yarkhan and Dongarra 2002] uses statistics to search optimal allocation. In addition, many other heuristics have also been proposed, such as duplex [Jose et al. 2008], neural networks [Xie et al. 2005], etc. Also, the work [Vermeulen et al. 2009] handled the problem of task level resource scheduling for patients in hospital under the resource availability constraint (resource calendar).

Compared with the works mentioned above, resource allocation on workflow systems considers the complex dependencies among the tasks of business processes. In this area, resource management can be classified into two categories. The first category is to allocate suitable resources for workflow instances at run time. Yu and Buyya [2006] developed a genetic approach to solve the deadline constrained scheduling problem. The fitness function combines time fitness and cost fitness. Based on the fitness value, their algorithm searches for a solution which has minimal execution cost with the deadline by two types of mutation operations: the swapping mutation and the replacing mutation. An ant colony optimization approach is proposed in Chen et al. [2010] to handle the time-constrained workflow scheduling problem using pheromone and heuristic information. Yarkhan and Dongarra [2002] proposed a solution using simulated annealing to select a suitable size of a set of resources for scheduling ScaLAPACK application in Grid environment. Senkul and Toroslu [2005] presented architecture of workflow scheduling under the resource constraints. In Russell et al. [2005], a novel framework of resource patterns for workflow resource management is proposed. Some approaches such as Ashraf and Erlebach [2010], Decker and Schneider [2007], and Langguth and Schuldt [2010] consider resource reservation when scheduling the workflow/business processes. In contrast, the second category is to plan resource for certain number of workflow instances at the build time. In this category, more instance dependency information is assumed to be available and therefore can be explored for resource planning at the build time. In Xu et al. [2008], process execution plan optimisation is discussed in aspects of inter-related factors: structural improvement and resource allocation. In Xu et al. [2009], two strategies are proposed to plan resources for a massive number of process instances before execution, in order to meet the deadline and minimise total cost. Also, resource planning for service-oriented workflows is investigated in Eckert et al. [2008]. It introduces the required architecture for resource planning and workload prediction. Furthermore, it presents optimisation approaches and heuristics for solving the resource planning problem with low computational overhead. In addition, Liu et al. [2010] discussed how to adjust the static planning scheme at the run time when it is violated by a resource or workflow requirement change.

To the best of our knowledge, none of the existing work discussed above consider temporal patterns of resource availability, which is a crucial issue for the rational use of resources in practice. This article focuses on the scheduling of process instances with resource availability constraints, which faces a larger search space as it involves resource temporal patterns, capabilities, business process structural information. In particular, we proposed three heuristic-based methods to find rational results efficiently for run-time applications, as well as a genetic algorithm-based approach to provide near-optimal results for build-time applications. Compared with the existing GA-based methodologies [Wu et al. 2004; Yoo 2009; Yu and Buyya 2006; Zomaya and Teh 2001], our business process scheduling approach integrates resource availability patterns, business process structure and temporal constraints more seamlessly. It is able to start from a good initial population because of the use of more rational results (rather than simple heuristic based) and temporal aware encoding. Moreover, the proposed approach is expected to efficiently provide better quality result due to usage of the carefully designed crossover and mutation operations, which are sensitive to resource temporal patterns and structural information.

7. CONCLUSION AND FUTURE WORK

In this article, we tackled the problem of scheduling business process instances to satisfy certain availability constraints. We investigated how to allocate resources for process instances before execution to maximize the success rate of scheduling. As the problem is computationally hard, we explored a set of heuristic rules and proposed one greedy algorithm and two holistic algorithms. We also proposed a genetic algorithm to provide near-optimal scheduling of instances at build time. Comprehensive experimental studies were conducted to evaluate all the algorithms. The proposed strategies can be used to rationally plan the enterprise resource utilisation for processing the business process instances requested by customers, and the information gathered from the scheduling process is useful for enterprises to make quality decisions for marketing.

In future, we plan to inter-play resource-level constraints with process-instance-level constraints, e.g., inter-task temporal constraints. Also, we will explore more optimization criteria to recognize the different impacts of workflow structure to each instance.

REFERENCES

J. Ashraf and T. Erlebach. 2010. A new resource mapping technique for grid workflows in advance reservation environments. In *Proceedings of HPCS*. 63–70.

A. Avanes and J. C. Freytag. 2008. Adaptive workflow scheduling under resource allocation constraints and network dynamics. In *Proceedings of VLDB*, vol. 1, No. 2, 1631–1637.

W. Chen, Y. Shi, H. Teng, X. Lan, and L. Hu. 2010. An efficient hybrid algorithm for resource-constrained project scheduling. *Information Sciences* 180 (2010), 1031–1039.

J. Decker and J. Schneider. 2007. Heuristic scheduling of grid workflows supporting co-allocation and advance reservation. In *Proceedings of CCGRID*. 335–342.

J. Eckert, D. Ertogrul, A. Miede, and N. Repp. 2008. Ralf Steinmetz: Resource planning heuristics for service-oriented workflows. In *Proceedings of 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*. Vol. 3, 591–597.

B. Fechner, U. Honig, J. Keller, and W. Schiffmann. 2008. Fault-tolerant static scheduling for grids. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*. 1–6.

J. Heinonen and F. Pettersson. 2007. Hybrid ant colony optimization and visibility studies applied to a job-shop scheduling problem. *Applied Mathematics and Computation (AMC)* 187, 2 (2007), 989–998.

A. Iosup, M. Jan, O. Sonmez, and D. H. J. Epema. 2007. On the dynamic resource availability in grids. In *Proceedings of GRID*, 26–33.

M. T. Jensen. 2003. Generating robust and flexible job shop schedules using genetic algorithms. *IEEE Transactions on Evolutionary Computation* 7, 3 (2003), 275–288.

F. Jin, S. Song, and C. Wu. 2009. A simulated annealing algorithm for single machine scheduling problems with family setups. *Computers & Operations Research* 36, 7 (2009), 2133–2138.

J. Jose, A. E. Ashikhmin, P. Whiting, and S. Vishwanath. 2008. Scheduling and precoding in multi-user multiple antenna time division duplex systems. *The Computing Research Repository*, abs/0812.0621, 2008.

C. Langguth, and H. Schuldt. 2010. Optimizing resource allocation for scientific workflows using advance reservations. In *Proceedings of SSDBM*. 434–451.

X. Liu, J. Chen, Z. Wu, Z. Ni, D. Yuan, and Y. Yang. 2010. Handling recoverable temporal violations in scientific workflow systems: A workflow rescheduling based strategy. In *Proceedings of CCGRID*. 534–537.

X. Liu, Y. Yang, Y. Jiang, and J. Chen. 2011. Preventing temporal violations in scientific workflows: Where and how. *IEEE Transactions on Software Engineering (TSE)*, 37, 6 (Nov./Dec. 2011), 805–825.

B. Rood, and M. J. Lewis. 2008. Scheduling on the grid via multi-state resource availability prediction. In *Proceedings of GRID*. 126–135.

N. Russell, W. M. P. van der Aalst, A. H. M. T. Hofstede, and D. Edmond. 2005. Workflow resource patterns: Identification, representation and tool support. In *Proceedings of CAiSE*. 216–232.

S. U. Seçkiner and M. Kurt. 2007. A simulated annealing approach to the solution of job rotation scheduling problems. *Applied Mathematics and Computation (AMC)* 188, 1 (2007), 31–45.

P. Senkul and I. K. Toroslu. 2005. An architecture for workflow scheduling under resource allocation constraints. *Information System* 30, 5 (2005), 399–422.

I. B. Vermeulen, S. M. Bohte, S. G. Elkhuizen, H. Lameris, P. J. M. Bakker, and H. L. Poutré. 2009. Adaptive resource allocation for efficient patient scheduling. *Artificial Intelligence in Medicine* 46, 1 (2009), 67–80.

A. S. Wu, H. Yu, S. Jin, K. C. Lin, and G. A. Schiavone. 2004. An incremental genetic algorithm approach to multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems* 15, 9 (2004), 824–834.

E. Xie, J. Xie, and J. Li. 2005. Fuzzy due dates job shop scheduling problem based on neural network. In *Proceedings of ISNN*. 782–787.

J. Xu, C. Liu, and X. Zhao. 2008. Resource allocation vs. business process improvement: How they impact on each other. In *Proceedings of BPM*. 228–243.

J. Xu, C. Liu, and X. Zhao. 2009. Resource planning for massive number of process instances. In *Proceedings of CoopIS*. 219–236.

J. Xu, C. Liu, X. Zhao, and S. Yongchareon. 2010. Business process scheduling with resource availability constraints. *On the Move to Meaningful Internet Systems: (OTM) 2010 – Confederated International Conferences: CoopIS, IS, DOA and ODBASE, Part I.* In *Lecture Notes in Computer Science (LNCS)*, vol. 6426. Springer, Berlin, 419–427.

A. Yarkhan and J. J. Dongarra. 2002. Experiments with scheduling using simulated annealing in Grid environment. In *Proceedings of GRID*. 232–242.

M. Yoo. 2009. Real-time task scheduling by multi objective genetic algorithm. *Journal of Systems and Software* 82, 4 (2009), 619–628.

J. Yu and R. Buyya. 2005. A taxonomy of workflow management systems for Grid computing. *Journal of Grid Computing* 3, 3–4 (2005), 171–200.

J. Yu and R. Buyya. 2006. Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms. *Scientific Programming* 14 (2006) 217–230.

A. Y. Zomaya and Y. H. Teh. 2001. Observations on using genetic algorithms for dynamic load- balancing. *IEEE Transactions on Parallel and Distributed Systems* 12 (2001), 899–911.

Received March 2015; revised July 2016; accepted August 2016