

Manuscript Number: ARTINT-D-14-00217

Title: C\*: an intelligent backtracking approach for scheduling jobs with their own timing constraints

Article Type: Full Length Article

Keywords: intelligent backtracking; scheduling jobs; timing constraints; position manipulation; heuristic search

**Abstract:** In this paper, we introduce a new orthogonal backtrack search algorithm to solve the problem of scheduling jobs with their own timing constraints. The proposed algorithm, called C\*, is a heuristic search technique and is totally independent from the input problem instance to be solved. In contrast to all the existing backtracking algorithms, neither an implicit nor an explicit tree is used to perform the backtrack scheme in this new algorithm; we use only two heuristics for each job for the backtracking task. The first heuristic corresponds to the job position in the current schedule and the second heuristic is a corrective function to decrease the position when needed. Over repeated problem solving trials, the C\* algorithm updates both heuristics until there is convergence to a complete feasible solution. The completeness is proven with a random tie breaking and the existence of feasible schedule conditions. The space complexity of the proposed algorithm is polynomial, as with all existing backtracking algorithms. The search for feasible schedules in C\* is driven by constraint violations; it thereby overcomes the problem of systematic traversal from left-to-right of the DFS-spanning tree in current backtracking algorithms. The experimental study shows that C\* performs an order of magnitude better than Squeaky Wheel Optimization, Simulated Annealing, and Tabu Search in randomly-generated problem instances. The experimental study also reveals that C\*, in its orthogonal form, significantly surpasses the IBM ILOG CP solver in many random problem instances.

# C\*: an intelligent backtracking approach for scheduling jobs with their own timing constraints

Yacine Laalaoui  
College of Computer and Information Technology,  
Taif University, Taif, KSA.  
email : y.laalaoui@tu.edu.sa

---

## Abstract

In this paper, we introduce a new orthogonal backtrack search algorithm to solve the problem of scheduling jobs with their own timing constraints. The proposed algorithm, called C\*, is a heuristic search technique and is totally independent from the input problem instance to be solved. In contrast to all the existing backtracking algorithms, neither an implicit nor an explicit tree is used to perform the backtrack scheme in this new algorithm; we use only two heuristics for each job for the backtracking task. The first heuristic corresponds to the job position in the current schedule and the second heuristic is a corrective function to decrease the position when needed. Over repeated problem solving trials, the C\* algorithm updates both heuristics until there is convergence to a complete feasible solution. The completeness is proven with a random tie breaking and the existence of feasible schedule conditions. The space complexity of the proposed algorithm is polynomial, as with all existing backtracking algorithms. The search for feasible schedules in C\* is driven by constraint violations; it thereby overcomes the problem of systematic traversal from left-to-right of the DFS-spanning tree in current backtracking algorithms. The experimental study shows that C\* performs an order of magnitude better than Squeaky Wheel Optimization, Simulated Annealing ,and Tabu Search in randomly-generated problem instances. The experimental study also reveals that C\*, in its orthogonal form, significantly surpasses the IBM ILOG CP solver in many random problem instances.

**keywords** :intelligent backtracking, scheduling jobs, timing constraints, position manipulation, heuristic search.

---

## 1. Introduction

Scheduling jobs under timing constraints is a key challenge for embedded real-time systems[1]. It involves scheduling a set  $\Pi$  of  $n$  jobs with their own hard-timing constraints onto a specific hardware architecture (e.g., uniprocessor or multiprocessor) subject to satisfy all of the timing constraints. The resulting schedule which guarantees such timing requirements is said to be feasible. When one or more jobs violate their timing constraints, then there are conflicts where each conflict corresponds to a job with unsatisfied constraints. The complexity of uniprocessor scheduling depends on whether pre-emptions between jobs are allowed or not. If pre-emptions are allowed, then the problem is solvable using a polynomial time algorithm [2]. When pre-emptions are not allowed, each job will be executed until its completion, releasing the processor to another job or, to an idle time during

which the processor is free. Jeffay et al have proved that the problem of non-preemptive scheduling of jobs with their own timing constraints is NP-Hard [3]. Often in real-time systems there is no optimization function to consider when searching for feasible schedules, i.e., all existing feasible schedules in that state-space are at the same preference level [1][4][5]. This means that the problem of scheduling jobs, each with their own timing constraints, is a generic Constraints Satisfaction Problem (CSP). Existing algorithms that solve this problem are either Complete or Incomplete [6].

Complete algorithms are classified as either complete-and-repair or backtracking. Complete-and-repair algorithms have exponential time and space complexities since they are often based on branch-and-bound or dynamic programming techniques [4][5][7]. Complete-and-repair algorithms perform an iterative improvement of a complete non-feasible solution with multiple conflicts. The search stops when all conflicts have been repaired. The limitation of the space complexity makes complete-and-repair algorithms less popular for solving CSP problems [8]. Further, many conflicts can arise after the repair of one conflict due to the interdependency among conflicts [9].

Backtracking algorithms have a polynomial rather than an exponential space complexity. They are based on Depth-First-Search (DFS) algorithm, and thus store only one branch in memory at a time to prevent memory exhaustion. In backtracking algorithms, only one conflict that lies at the end of the current partial feasible solution is repaired at a time. DFS can take a very long time to find a feasible solution because it performs a chronological search in the spanning tree from left to right and branch <sup>4</sup> by branch. This type of search is called Chronological backtracking [10]. Backjumping schemes have been proposed to address the time problem of chronological backtracking searches. When a conflict arises, the last-selected job will be sent back as an attempt to avoid such conflicts and to continue the extension of the current partial feasible solution in a new direction [10][11][12][13][14][15] [16]. There are two properties common to all backjumping techniques: 1) their searches are tree-based with polynomial space complexity, and 2) their shapes are non-orthogonal, i.e., two or more techniques are combined in one global algorithm.

Incomplete algorithms include meta-heuristics such as Simulated Annealing (SA), Ant Colony Optimization (ACO) and Genetic Algorithms (GA). SA was used by Tindell et al in [17] to handle the problem of scheduling in a distributed environment with a special communication protocol. It was also used by Stankovic et al. in [18] to address the scheduling problem in multiprocessor architecture with jitter minimization. ACO has been used in [19] [20] [21] to address the problem of scheduling tasks in hard-real-time systems. A framework for heterogeneous and non-preemptive multiprocessor scheduling is described in [19]. The problem with ACO meta-heuristic is the stagnation that can make it impossible for the scheduling algorithm to find feasible solutions. In [20], the authors describe a technique to improve ACO's ability to avoid stagnation situations by adding a distance parameter to the selection rule. GA was used by Roman in [22] to address the problem of pre-emptive scheduling of inter-related tasks with precedence and exclusion constraints in extensible multiprocessor architecture. Navet et al in [23] also used GA to solve the problem of policies (Round Robin or FIFO scheduling) and priorities' assignment to tasks in POSIX1003.1b compliant systems. In sum, SA is a slow technique, ACO techniques have the stagnation problem, and GA has the premature convergence problem. Furthermore, meta-heuristics are probabilistic algorithms; they have been used especially to solve large problem instances. However, their efficiency is limited since they do not provide the guarantee of finding feasible solutions. Josline and Clements in [24] proposed a heuristic search algorithm called Squeaky Wheel Optimization (SWO) to schedule jobs

---

<sup>4</sup>A branch in a spanning tree is the set of nodes from the root to the leaf.

with deadlines. They also used a number of problem instances of the Graph-Coloring problem to demonstrate the ability of SWO to solve CSP problems. The SWO, when applied to scheduling jobs with deadlines, generates an initial complete-ordering (with multiple unsatisfied jobs) of the input job set and then analyzes that ordering to determine the jobs that lead to conflicts. A new complete-ordering is then generated based on the analysis phase. The whole process of analysis-repair is repeated for a fixed number of trials until a feasible schedule is found or until failure. The SWO belongs to the group of Complete-and-Repair approaches, which are used less often than backtracking algorithms in CSP problems [8]. Moreover, the SWO technique is not orthogonal and its efficiency is subject to the constraints of integrated techniques.

We propose a backtracking heuristic search algorithm that has some similarities to SWO. The new algorithm extends a partial feasible solution; repairing one conflict at time until reaching a complete feasible solution. The proposed algorithm manipulates jobs' positions to improve the quality of the current partial feasible solution, as with the SWO.

## 2. Problem formulation

The search space is a set  $\Pi$  with a finite number of jobs,  $\Pi = \{J_i \mid 1 \leq i \leq n \wedge n \in \mathbb{N}\}$ . The set  $\Pi$  is expected to be scheduled non-preemptively on a single processor architecture<sup>5</sup>. The tuple:  $\langle r(J_i), C(J_i), d(J_i) \rangle$  defines all timing constraints for each job  $J_i$ , where:

- $r(J_i)$  is the release date.
- $C(J_i)$  is the computation time.
- $d(J_i)$  is called the deadline. It is the date limit given to the job to complete its execution.

In this paper we assume the following:

- For each job  $J_i$ , we assume :  $d(J_i) \geq r(J_i) + C(J_i)$ . This provides enough time to run all of the computation units of  $J_i$  within the specified deadline.
- All timing parameters are finite positive integer values.
- All timing parameters are fixed and no variable values are considered.
- There are no precedence constraints between jobs and each job can be selected independently from other jobs in  $\Pi$ .
- The Scheduling Horizon ( $SH$ ) is fixed in advance. It defines the interval  $[0, SH]$  in which all jobs must be scheduled.
- The load of the problem instance (or system load), noted  $U$ , represents how much of the scheduling horizon is used. If it is equal to 1.00, the scheduling horizon is fully utilized and there is no idle time. The system load is defined as follows :

$$U = \sum_{i=1}^{|\Pi|} C(J_i) / SH \quad (1)$$

---

<sup>5</sup>This problem is also known as disjunctive scheduling.

It is expected that all computation units for each job  $J_i$  will be executed within the interval  $[r(J_i), d(J_i)]$ . Let  $start(J_i)$  be the instant when  $J_i$  starts its execution. We note that the start time is greater than or equal to the job release date. Let  $end(J_i)$  be the time when the processor is yielded to another job that is not  $J_i$ . We note that the start time for each job  $J_i$  is taken from the interval  $[r(J_i), d(J_i)-C(J_i)]$  while the end time is taken from the interval  $[r(J_i)+C(J_i), d(J_i)]$ . If both the start and end times for a job  $J_i$  belong to their defined intervals, then  $J_i$  is said to be *satisfied*. Otherwise,  $J_i$  is considered to be *unsatisfied*. The scheduling algorithm assigns only one value because the assignment of the first value implicitly involves assigning the second value. To this end, in the following we only consider the assignment of start times.

This job model can be used in real-time systems by considering each job as a separate invocation for a given periodic task along  $SH$  equal to the least common multiple of all tasks' periods. A periodic real-time task has one more timing parameter called a period and it is noted  $p$ . Each periodic real-time task, that have the period  $p$ , has exactly  $SH/p$  jobs within the set  $\Pi$  [1][6].

### 3. Scheduling Algorithm: C\*

#### 3.1. Overview

C\* is a new algorithm to schedule a set of jobs with their own timing constraints. The name C\* is derived from CSP because the problem being investigated is a Constraint Satisfaction Problem. This new algorithm uses an integrated approach to gradually construct a complete feasible solution. Start times and positions are (iteratively) assigned to each job until a complete assignment that satisfies all timing constraints is achieved. Each time a selected job is unsatisfied, the construction is stopped to try another search direction, with the job's position decreased according to how far is that job is from its correct position in the complete feasible schedule. The jobs' positions are refined iteratively until a complete feasible schedule is found.

#### 3.2. Algorithm Description

Let  $h_1$  be the tentative assignment function that maps each job to one position in the schedule. Let  $P$  be the set of all possible positions in the schedule.

$$h_1 : \Pi = \{J_1, \dots, J_n\} \longrightarrow P = \{1, 2, \dots, n\} \quad (2)$$

$$J \longrightarrow p \quad (3)$$

Let  $h_2$  be a corrective function that corrects each assigned position  $h_1$  of each job from  $\Pi$ . Let  $X$  be the set of all possible values to be returned by  $h_2$ .

$$h_2 : \Pi = \{J_1, \dots, J_n\} \longrightarrow X = \{0, 2, \dots, n-1\} \quad (4)$$

$$J \longrightarrow x \quad (5)$$

Algorithmically, both functions  $h_1$  and  $h_2$  are two vectors that store information related to each job. Initially, for each job  $J_i$ , we have  $h_1(J_i) = 1$  and  $h_2(J_i) = 0$  as it is assumed that all jobs are

candidates for the first position and no job yet been selected as unsatisfied. Let *Open* be the list of all unselected jobs. Let *Schedule* be the list of all selected jobs. Note that the union of *Open* and *Schedule* is the set  $\Pi$ , and that *Open* and *Schedule* are disjoint.

The pseudo-code of the C\* algorithm is shown in **Figure 1**. C\* works in two phases to find a feasible solution. During the first phase (the greedy phase), C\* attempts to construct a complete feasible solution (steps from 05 to 22). During the second phase, C\* backtracks to retrieve previous solutions (steps from 24 to 28). Both phases are performed iteratively until a feasible solution is reached.

The algorithm starts by initializing the variables (steps 01, 02 and 03). The initialization of  $h_1$  to 1, at step 01, means that all the jobs compete to take the first position in the final schedule (all are candidates for the first position). The initialization of  $h_2$  to 0, at step 01, means that no job has been selected as unsatisfied. The *Schedule* set is initially empty (step 02). The set *Open* is initialized to all jobs from  $\Pi$ . We note that the *Schedule* set is implemented as a linked list, where the front is the first job and the back is the last job. All of the jobs from the *Schedule* set are satisfied.

Steps 04 to 28 contain a **Repeat** loop so that a repeated problem-solving loop is performed until a feasible schedule is found. Step 05 initializes the variable *stop* to be used during the solution construction. At step 06 a **Do While** loop starts; it ends at step 26. This loop is used to construct the solution. The conditions to continue this **Do while** loop are 1) when *Open* is not empty, and 2) the boolean variable *stop* is true. If *Open* is empty, then all jobs have been successfully selected as satisfied.

Steps 07, 08 and 09 initialize variables *LastJob*,  $t$  and  $p$ , respectively, where *LastJob* is the last appended job in *Schedule*,  $t$  is the end time of *LastJob* and  $p$  is the current position to be assigned.  $t$  is set to 0 if *Schedule* is empty. At step 10, the job *Current* is selected. This is the job with the lowest value  $f = h_1 - h_2$  from the set *Open*. When needed, idle times are inserted at step 11. Consider the example of two jobs *A* and *B* defined as follows :  $A(0, 1, 2)$  and  $B(3, 1, 5)$ . Each job is defined with a tuple (release date, processing time, and deadline). The final schedule is  $(A, B)$ . The start time of *A* is 0 and its end time is 1. The start time for *B* is 3 and its end time is 4. It is clear that the processor is idle between the end time of the first job and the start time of the second job, since no job is executed during this period.

At step 12, if the selected job, called *Current*, is unsatisfied, then its heuristic value  $h_2$  is updated (step 13) and the **Do While** loop is stopped (step 14). Notice that the first selected job could never be unsatisfied. If *Current* is satisfied, then multiple instructions are performed from step 16 to 21. At step 16,  $h_2(\text{Current})$  is reset to 0 if the start time of the job *Current* is equal to its release date, as no more backward moves are needed.

If we skip step 16, then the  $h_2$  heuristic value for each job will take infinite values, since they are increased without bound in step 13. The heuristic value  $h_1(\text{Current})$  takes the current position stored in the variable  $p$  (step 17). At step 18, if the increasing order of jobs according to their  $f$  values is not preserved in *Schedule*, then the trial is stopped in order to start the search in a new trial. This step does not involve the update of the the second heuristic  $h_2$  because the  $f(\text{Current})$  value is high enough to select the job *Current* before the job *LastJob* in *Schedule*. If we skip step 18, then the backtracking phase will be affected. In other words, the backtracking process would be stopped at an incorrect position. Thus, if  $f(\text{Current}) < f(\text{LastJob})$ , then the construction solution process needs to be stopped (terminating the greedy phase) before appending *Current* to the current partial feasible solution.

When the construction phase has been terminated and the current solution is not complete,

the process returns to the backtracking phase. This phase removes jobs from the current partial feasible solution until it finds the correct position of *Current* (according to its  $f$  value) in the current partial feasible solution. Again the construction phase is called upon to start the selection of the remaining jobs. The job stored in *Current* will definitely be selected first since it has the lowest  $f$  value among all the jobs in *Open*.

For example, let  $(j_1(1), j_2(3), j_3(3), j_4(4))$  be the current partial solution stored in *Schedule* with the  $f$  value for each job.  $f(j_1)$  is equal to 1,  $f(j_2)$  and  $f(j_3)$  are equal to 3 and  $f(j_4)$  is equal to 4. If  $j_5$  is selected as satisfied and it has its  $f$  value equal to 2, then the construction phase stops and the backtracking phase starts to remove jobs  $j_4, j_3$ , and  $j_2$  because their  $f$  values are above  $f(j_5)$ . Once the backtracking phase terminates, the the solution construction phase is called again, to select  $j_5$  after  $j_1$  and so on for the remaining jobs. The content of *Current*,  $j_5$ , is appended at the end of *Schedule* at step 20 and it is removed from *Open* at step 21.

If the set *Open* is empty (step 22), then all jobs have been successfully selected as satisfied and the whole algorithm  $C^*$  ends the search. Otherwise, the backtracking phase must be called. At step 32, if the search has led to an incomplete schedule, then the backtracking phase will be performed at steps 24-28. The backtracking phase is called upon to retract a solution from those previously found. During this phase, recent jobs are removed from *Schedule* and appended to *Open*. This phase continues to remove jobs from *Schedule* until a specific condition is not satisfied. That condition will be discussed in the next sections. The search is then continued in a new direction during the subsequent trial, starting from step 05.

It is worth noting that during the search for a feasible schedule, there is a trial during which all jobs from the input set are visited at least one time. We call this trial the *critical trial*.

**Definition 3.1.** (*Critical Trial*) A critical trial is reached when all jobs from the input set  $\Pi$  have been selected at least once.

The *critical trial* denotes the difference between two phases during the search process, the *exploration* and the *improvement* phases. During the former,  $C^*$  proceeds for visiting all non-visited jobs, and during the latter phase it proceeds for improvement of its partial solution after repeated trials. The backtracking process removes all jobs from *Schedule* before the critical trial, because non-visited jobs have the lowest  $f$  values among all jobs. However, the backtracking process will not remove all jobs from *Schedule* after the critical trial. Only a specific number of jobs are removed from *Schedule*, as we will see in the next section.

### 3.3. Tie breaking

A tie situation occurs when more than one alternatives have the same  $f$  value during selection times. This typically happens with pairs of alternatives. The most common causes of a tie situation are the initial heuristic values  $h_1$  and  $h_2$ , since the same values are assigned for all jobs. The second cause is when the  $h_2$  value is increased for a job  $J_j$ , the  $f(J_j)$  value is decreased to coincide with  $f(J_i)$ , where  $J_i$  is the job selected before  $J_j$ . The third cause of tie situations is when  $C^*$  assigns  $h_1$ ; it starts from 1 until a specific position which is unknown, and it depends on the number of jobs successfully scheduled during the current trial. If more jobs are still awaiting their selection for the first time during the next trial, then  $C^*$  starts assigning positions again from 1 to another unknown position. This process is repeated until all jobs have been selected at least once (in a critical trial). Along many trials,  $C^*$  could assign the same sequence of positions to different jobs. After the critical trial, there will be many jobs with the same positions which will cause tie situations. How such situations are handled has a great impact on the efficiency of the  $C^*$  algorithm.

---

```

Algorithm C* ;
INPUT : a set of jobs  $\Pi$ ;
OUTPUT : Schedule;
VARS:  $t$ , stop, Open, mark, Current and LastJob;
BEGIN
(01) Initialize both heuristics  $h_1$  and  $h_2$  to 1 and 0 respectively;
(02) Schedule =  $\emptyset$ ;
(03) Open =  $\Pi$ ;

(04) Repeat the following until convergence to an existing feasible schedule

(05)   stop = false;
(06)   DO
(07)     LastJob = Schedule.back(); /*LastJob is set to an extra start job if Schedule is empty */
(08)      $t = \text{end}[\text{Schedule.back}()];$  /* $t$  is set to 0 if Schedule is empty */
(09)      $p = \text{sizeof}( \text{Schedule} ) + 1;$ 
(10)     find the job Current from Open with the lowest  $f = h_1 - h_2$ ;
        In case of a tie, perform a random selection and save the position of the first tie situation in mark

(11)     IF(  $t < r[\text{Current}]$  ) THEN  $t = r[\text{Current}]$ ;

(12)     IF(  $(t + C[\text{Current}]) > d[\text{Current}]$  ) THEN //Current is unsatisfied
(13)        $h_2[\text{Current}] ++;$ 
(14)       stop = true;

(15)     ELSE
(16)       IF(  $\text{start}[\text{Current}] == r[\text{Current}]$  ) THEN  $h_2[\text{Current}] = 0;$ 

(17)        $h_1[\text{Current}] = p;$ 

(18)       IF(  $f(\text{Current}) < f(\text{LastJob})$  ) THEN stop = true;

(19)     ELSE
(20)       Schedule = Schedule  $\cup$  ( Current,  $t$  );
(21)       Open = Open - { Current };
(22)     WHILE( Open  $\neq \emptyset$  AND NOT stop );

(32)   IF ( Schedule is not Complete ) THEN

(24)     DO
(25)        $J = \text{schedule.back}();$ 
(26)       Open.push_back( $J$ );
(27)       Schedule.pop_back();
(28)     WHILE( Condition )

END.

```

---

Figure 1: C\* pseudo-code

Basically, there are two solutions to handle a tie situation, either randomly or deterministically. Usually only one policy is used along the whole search process. Deterministic tie breaking may require the use of domain-dependent heuristics (e.g., earliest deadline) or domain-independent (e.g., first encountered or second encountered). In all cases of deterministic tie breaking, there will be one preferred path that can cause the scheduling algorithm  $C^*$  to become trapped somewhere in the search space. In contrast, a random tie-breaking prevents  $C^*$  from getting stuck somewhere.

Consider an example of a solvable set with 7 jobs. **Figure 2** and **Figure 3** show the number of satisfied jobs among the search trials. At the end of each trial, the size of the partial feasible solution is captured to draw these curves. In **Figure 2**, tie situations are broken deterministically. Since the shape of this curve is the same along 500 trials, the  $C^*$  algorithm is able to construct only a fixed number of partial solutions. The maximum number of satisfied jobs is 6, while the total number is 7 (dashed curve). This means that there is an infinite cycle. Therefore, if ties are broken deterministically, then there will be one preferred path along the search trials, which could prevent the scheduling algorithm from finding existing feasible schedules. In contrast, **Figure 3** shows that the number of successfully selected jobs reaches the total number of jobs after a finite number of trials. This is due to the use of a random tie-breaking technique. We can also observe that there is no preferred path before 150 trials (convergence trial) since the shape of the curve is not repetitive among the search trials.

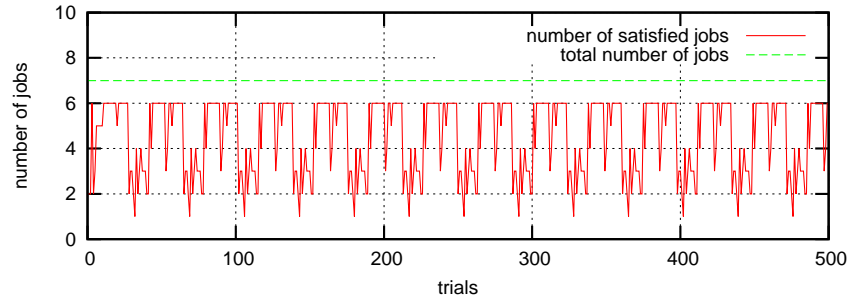


Figure 2: Satisfied jobs during 500 trials. There is an infinite cycle caused by deterministic tie breaking.

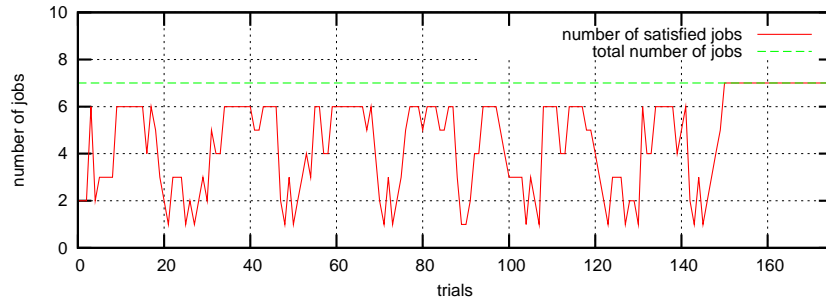


Figure 3: Convergence to a feasible solution after 150 trials. No infinite cycle when ties are broken randomly.

### 3.4. Backtracking positions

One important issue during the design of the C\* algorithm is the backtracking position; determining at which position C\* should stop removing jobs from *Schedule* during the backtracking phase. To this end, we have developed four conditions that can be used during the backtracking phase. Notice that the C\* algorithm should only use one condition combination at a time (always in combination with Condition 1) to decide at which position to stop the backtracking process. The backtracking phase will keep removing jobs while the selected condition is satisfied.

**Condition 1** (*Schedule is not empty*) . : This condition is obvious since no job could be removed from an empty set. This condition must be combined with one of the remaining conditions because if it is not satisfied, the backtracking phase stops.

**Condition 2** ( $f(Current) < f(LastJob)$ ) **AND** (*Schedule is not empty*) . : This condition is used for those cases where either the job stored in *Current* is satisfied or unsatisfied. If *Current* is unsatisfied, then  $f(Current)$  is less than the  $f$  value of the last job in *Schedule* ( $f(Current) < f(LastJob)$ ). Hence, *Current* must be sent back in the *Schedule* set, according to its  $f$  value. The backtracking phase removes jobs one by one from the *Schedule* until either this set is empty or the last job in *Schedule* has an  $f$  value less than  $f(Current)$ . This condition is obvious, since the C\* algorithm maintains an ordered sequence of jobs according to their  $f$  values in the *Schedule* set. If *Current* is satisfied, then its new position setting, at step 17, causes  $f(Current)$  to be less than  $f(LastJob)$ . Step 18 in the construction phase (**Figure 1**) verifies that the  $f$  heuristic value of each new selected job is smaller than that of  $f(LastJob)$ , where *LastJob* is the last job in *Schedule*. As explained above, this condition should not be skipped. Otherwise, the backtracking phase could be stopped at incorrect positions when removing jobs from *Schedule*.

**Condition 3** ( $f(Current) < f(LastJob)$  **OR** *earliest tie position is not reached*) **AND** (*Schedule is not empty*) . :

It is clear that tie situations could lead to wrong paths due to random selection, and the only way to return to the correct path would be to backtrack until the first random selection. Thus, in addition to Condition 2, we used also the position of the earliest tie situation to make this new condition. Jobs are removed from *Schedule* until the earliest random tie-breaking position, stored in the variable *mark* is encountered.

**Condition 4** ( $f(Current) < f(LastJob)$  **OR**  $h_2(LastJob) \neq 0$ ) **AND** (*Schedule is not empty*) . : This condition adds the  $h_2$  heuristic value. If the last job in *Schedule* has its  $h_2$  heuristic value equal to zero, the backtracking process is stopped. Recall that when a job has been selected so that its start time is equal to its release date, the corresponding  $h_2$  heuristic value is reset to zero; let such job be *J*. Since unsatisfied jobs are moving backwards, from right to left, then unsatisfied jobs lie after the position of the job *J* in the current partial feasible solution. Thus, it is highly probable that jobs selected before the job *Current* have already been selected in their correct positions in the final feasible solution and no further backtrack is needed. If the job stored in *Current* must be sent back before *J*, then *Current* will be selected as unsatisfied one or more times to decrease its  $f$  value. This step will allow *Current* to be selected before *J*.

Conditions 1 and 2 are both obvious and should be combined with other conditions, as shown in conditions 3 and 4. In the empirical work, we have found that **Condition 4** gives the best performance, which we will demonstrate.

### 3.5. An example run

	A	B	C	D	E	F	G
$r$	0	3	1	4	13	11	14
$C$	5	3	2	2	3	2	2
$d$	20	10	10	10	20	20	20

Table 1: An example of a solvable set with 7 jobs.

Consider an example of a solvable problem instance  $\Pi$  with 7 jobs. The corresponding timing constraints are specified in **Table 1**. **Table 2** shows the behavior of the C\* algorithm during the search for a feasible schedule to this problem instance. **Table 2** contains four columns. The first column indicates the trial number. The second column shows the content of both the *Schedule* and *Open* data structures. *Schedule bb* represents the set *Schedule* before the backtracking phase and *Schedule ab* indicates the content of the same set, but after the backtracking phase. The content of *Open* in this table means the set of jobs at the beginning of the trial, i.e., before the job selection process. The third column shows why the feasible solution construction process was stopped. The last column shows the reason for stopping the backtracking phase. We have used Condition 3 to perform the backtracking process.

According to **Table 2**, C\* finds a feasible schedule after 15 trials. At trial 15, the feasible solution has been found. Trial number 4 is the critical trial, since at this trial all jobs have been selected at least one time. Before trial 4, the backtracking phase has removed all jobs from the partial feasible solution that resulted in an empty solution as shown in *Schedule ab*.

At trial 4, the reason for stopping the construction of the feasible solution is the unsatisfied selection of the job  $C$ . At this trial, the reason to stop the backtracking is that Condition 3 is not satisfied. Precisely,  $f(C)$  (equal to 2) is strictly below  $f(A)$  (the last job in *Schedule*). Thus, the correct position of job  $C$  with respect to the selection rule is after job  $A$ . At trial 10, job  $C$  is again unsatisfied, and its  $h_2$  heuristic is automatically increased by one. Thus,  $f(C)$  is equal to one instead of two. The reason to stop the backtracking at trial 10 is the same as at trial 4; the order of jobs in *Schedule* is maintained once job  $B$  has been removed from this set since both jobs  $D$  and  $C$  have the same  $f$  value, which is equal to 1. The search continues until no conflict is found at trial 15.

We will now explain how the C\*'s search is not systematic in one direction. Recall that the initial order of jobs is A, B, C, D, E, F and G. The first selected job in *Schedule* is not A at trial 1, it is F. Trials 5, 6 and 7 are successive, but the first job in *Schedule* during trial 5 is B while in trial 6 the first job is C. Job B is again the first in *Schedule* in trial 7. This means that jobs could be selected at the first position more than once. If the C\* algorithm fails to find an existing feasible solution that starts with B at trial 5, then the C\* algorithm is able to recover solutions that starts with B during the next trials. This is not the same situation in DFS, since the first sub-tree with a root node A will never be encountered again during the search from left to right once the whole sub-tree with the root node A has been explored. Instead, only sub-trees with the remaining root nodes other than A will be encountered in DFS. The same thing also occurs during both trials 2 and 10, since their first-selected job is D, but trials 3 to 9 have a first selected job that is not D.

Trials	<i>Open</i> and <i>Schedule</i> contents	why stop the search	why stop backtracking
1	<i>Open</i> : A(1) B(1) C(1) D(1) E(1) F(1) G(1) <i>Schedule</i> bb: F(1,0,1) G(2,0,2) <i>Schedule</i> ab:	conflict : D	empty <i>Schedule</i>
2	<i>Open</i> : D(0) B(1) C(1) F(1) E(1) A(1) G(2) <i>Schedule</i> bb: D(1,0,1) A(2,0,2) F(3,0,3) E(4,0,4) <i>Schedule</i> ab:	conflict : B	empty <i>Schedule</i>
3	<i>open</i> : B(0) D(1) C(1) A(2) G(2) F(3) E(4) <i>Schedule</i> bb: B(1,0,1) C(2,0,2) D(3,0,3) G(4,0,4) <i>Schedule</i> ab:	conflict : A	empty <i>Schedule</i>
4	<i>Open</i> : A(1) B(1) C(2) D(3) F(3) G(4) E(4) <i>Schedule</i> bb: B(1,0,1) A(2,-1,1) <i>Schedule</i> ab: B(1,0,1) A(2,-1,1)	conflict : C	$\leq$ satisfied
5	<i>Open</i> : C(1) D(3) F(3) G(4) E(4) <i>Schedule</i> bb: B(1,0,1) A(2,-1,1) <i>Schedule</i> ab:	conflict : C	empty <i>Schedule</i>
6	<i>Open</i> : C(0) A(1) B(1) D(3) F(3) G(4) E(4) <i>Schedule</i> bb: C(1,0,1) A(2,-1,1) <i>Schedule</i> ab:	conflict : B	empty <i>Schedule</i>
7	<i>Open</i> : B(0) A(1) C(1) D(3) F(3) G(4) E(4) <i>Schedule</i> bb: B(1,0,1) C(2,0,2) A(3,-1,2) <i>Schedule</i> ab:	conflict : D	empty <i>Schedule</i>
8	<i>Open</i> : B(1) D(2) A(2) C(2) F(3) G(4) E(4) <i>Schedule</i> bb: B(1,0,1) C(2,0,2) A(3,-1,2) <i>Schedule</i> ab:	conflict : D	empty <i>Schedule</i>
9	<i>Open</i> : D(1) B(1) A(2) C(2) F(3) G(4) E(4) <i>Schedule</i> bb: B(1,0,1) <i>Schedule</i> ab:	$\leq$ violated	empty <i>Schedule</i>
10	<i>Open</i> : D(0) B(1) A(2) C(2) F(3) G(4) E(4) <i>Schedule</i> bb: D(1,0,1) B(2,0,2) <i>Schedule</i> ab: D(1,0,1)	conflict : C	$\leq$ satisfied
11	<i>Open</i> : C(1) B(2) A(2) F(3) G(4) E(4) <i>Schedule</i> bb: D(1,0,1) C(2,-1,1) <i>Schedule</i> ab: D(1,0,1)	conflict : B	$\leq$ satisfied
12	<i>Open</i> : B(1) C(1) A(2) G(4) F(3) E(4) <i>Schedule</i> bb: D(1,0,1) B(2,-1,1) <i>Schedule</i> ab:	conflict : C	empty <i>Schedule</i>
13	<i>Open</i> : C(0) B(1) D(1) A(2) F(3) G(4) E(4) <i>Schedule</i> bb: C(1,0,1) D(2,0,2) B(3,-1,2) A(4,-1,3) F(5,0,5) G(6,0,6) <i>Schedule</i> ab:	conflict : E	empty <i>Schedule</i>
14	<i>Open</i> : C(1) B(2) D(2) E(3) A(3) F(5) G(6) <i>Schedule</i> bb: C(1,0,1) D(2,0,2) B(3,-1,2) A(4,-1,3) E(5,-1,4) F(6,0,6) <i>Schedule</i> ab:	conflict : G	empty <i>Schedule</i>
15	<i>Open</i> : C(1) B(2) D(2) A(3) E(4) G(5) F(6) bb: C(1,0,1) B(2,0,2) D(3,0,3) A(4,-1,3) E(5,0,5) G(6,-1,5) F(7,0,7) ab: C(1,0,1) B(2,0,2) D(3,0,3) A(4,-1,3) E(5,0,5) G(6,-1,5) F(7,0,7)	no conflict	no backtrack

Table 2: An example run. Both sets *Schedule* and *Open* are reported. *Schedule* bb means a set with content before backtracking, and *Schedule* ab means a set with content after backtracking. *Open* content is the set of jobs before starting the appending of jobs into *Schedule*. In *Open*, only the  $f$  value of each job is reported.  $h_1$ ,  $h_2$  and  $f$  are reported in *Schedule*. The sign (-) has been added to emphasize the value of the  $h_2$  heuristic.

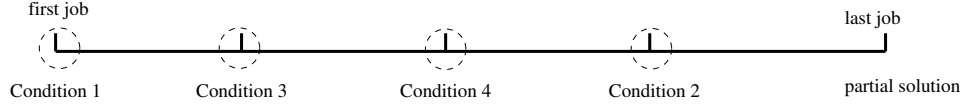


Figure 4: Backtracking positions using different conditions.

#### 4. Empirical study

In order to evaluate the efficiency of the proposed algorithm, we used the C++ programming language to implement all the following algorithms: C\*, Simulated Annealing (SA), Tabu Search (TS), Squeaky Wheel Optimization (SWO), DFS and Earliest Deadline First (EDF). The C\* algorithm was implemented and tested in its orthogonal form to show its efficiency without any built-in propagators. Further improvements in the inner steps of this new algorithm are proposed for future work. The use of SA meta-heuristic in this study is motivated primarily by its previous use in real-time scheduling by Dinatale et al. in [18] and Tindell et al. in [17]. Another motivation is its complete-and-repair nature; it is added here to represent a class of CSP solvers, those based on meta-heuristic techniques. TS meta-heuristic has been added to the present experimental study for even more enhancement. SWO is used in this study so that it can be compared to the new backtracking technique, since both the SWO and C\* algorithms manipulate the jobs' positions. In addition, the SWO algorithm's effect on random data sets, since the inventors of this technique claimed its efficiency using very few real-life problem instances with unknown loads. The DFS algorithm is implemented to compare its sensitivity to the distribution of existing feasible solutions in the state-space vs that of C\*. EDF is a simple heuristic, and is added to the present experimental study to reveal the number of easy-to-solve problem instances. All of the implementations were done on a Laptop with an Intel Core i5 processor (2.40 GHz and 6.0 GB RAM space).

##### 4.1. Backtracking conditions assessment

The objective of the current experiment is to assess the proposed backtracking conditions in order to evaluate which is the best in terms of time and ability to find feasible solutions <sup>6</sup> within a specific time slot. We used 30 randomly-generated data-sets. The complete description of these data-sets together with the generation method are described in **Appendix A**.

**Table 3** shows the results of running C\* for five seconds with different conditions to stop the backtrack process. Condition 4 makes it possible to find more feasible solutions than all the other conditions, as tabulated in bold in the column #success. Condition 4 also gives a stable performance in all data-sets. Condition 2 requires the least time to find a feasible solution, but its performance is not much less than the time taken by the C\* algorithm when Condition 4 is used. Conditions 1 and 2 give the worst success numbers because they are naive conditions. **Figure 4** shows the backtracking positions using different backtracking conditions. This figure is drawn using the time taken by C\* with different backtracking conditions. The earliest backtracking position is encountered using Condition 2, and the last position is encountered using Condition 1.

Recall that Condition 1 means backtracking until getting an empty solution. Thus, the use of this condition would cause the C\* algorithm to take the longest time, since the backtracking phase

---

<sup>6</sup>An algorithm which is able solve more problem instances is the efficient one. A complete algorithm is able to solve all solvable problem instances from a given data-set. In this paper, we will use the *success rate* or the *number of success* interchangeably to report the number of solved problem instances using a specific algorithm.

removes all the jobs from the current solution stored in *Schedule*. If only Condition 1 is used, then the backtracking phase is useless and the re-initialization of both the *Schedule* and *Open* sets to their initial contents suffices. Therefore, there is a need to backtrack, but not all the times to an empty solution. To overcome this shortcoming of Condition 1, we developed Condition 2, which is used in conjunction with Condition 1 since it is needed all the time to check whether *Schedule* is empty or not. In Condition 2, the backtracking phase keeps removing jobs from *Schedule* until it reaches the suitable position for the last selected job, called *Current*, which causes the construction of the feasible solution phase to be stopped. The suitable position for *Current* is detected according to its corresponding value  $f(Current)$  among all the jobs in *Schedule*. If the last job in *Schedule* has  $f$  value less than or equal to  $f(Current)$ , then the backtracking phase stops. The shortest time is found with the  $C^*$  algorithm under Condition 2, which means that the number of jobs removed from the *Schedule* set in the backtracking phase is very small compared to the number removed under the other conditions. If the number of removed jobs is very small in the backtracking phase, then the suitable position for *Current* is not far from its current position.

In terms of success rate,  $C^*$  with Condition 1 outperforms  $C^*$  with Condition 2 in 16 data-sets (out of 30).  $C^*$  with Condition 2 outperforms  $C^*$  with Condition 1 in 12 data-sets. Both conditions give the same results in two data-sets. Overall, conditions 1 and 2 give almost the same results.

Since both conditions 1 and 2 are naive, we developed Condition 3, in which we combined both conditions 1 and 2 plus one more condition on the earliest tie position. When condition 3 is used in  $C^*$ , the algorithm fails two times (data-set 22 and data-set 16), to give the best success rate compared to the use of conditions 1 and 2; but not by a very large gap (30-31 and 35-36). The successful results from Condition 3 show the effectiveness of using the earliest tie position encountered during the solution construction, and indicate that backtracking until the earliest tie position is often needed to recover the path leading to the feasible solution. However, sometimes it is not necessary to backtrack until the earliest tie position, because that could be the wrong position at which to stop the backtrack process. The fact that  $C^*$  fails 2 times when condition 2 is used, giving the best success rate among the use of conditions 1 and 2 confirms the former case of stopping the backtrack process at an incorrect position.

The use of Condition 3 means backtracking until the earliest tie position, and then starting to extend again in the second search alternative, since the first one does not lead to a complete feasible solution. It is worth noting that the earliest tie position during the current trial is not necessarily a tie position during the next trial, because the  $h_1$  and  $f$  values of jobs located after the earliest tie position in *Schedule* have been changed. Thus, backtracking until the earliest tie position does not mean that  $C^*$  will again face a tie situation at this position. In addition, many new tie positions could appear after the earliest position during the next trial, and for the same reason indicated above, i.e. the  $h_1$  and  $f$  values could be changed during the current trial. This would cause  $C^*$  to try other directions different than the one we want.

Condition 4 removes jobs from *Schedule* until getting the correct job order in the latter set and the last job in this set has its  $h_2$  heuristic value equal to zero. One clear remark is that when Condition 4 is used, then  $C^*$  removes only a few jobs that were recently add to *Schedule*. Overall, Condition 4 gives the best success and time performance among all other conditions

#### 4.2. $C^*$ versus DFS

The basic state-of-the-art algorithm that we have implemented is DFS which is a standard basic algorithm to solve CSP problems. The objective of the current experiment is to show the sensitivity to the distribution of solutions for both  $C^*$  and DFS algorithms. Both algorithms  $C^*$  and DFS are

	Condition 4		Condition 3		Condition 2		Condition 1	
data sets	#success	avg time	#success	avg time	#success	avg time	#success	avg time
data-set1	<b>57</b>	0.1263	48	1.3459	42	<b>0.0479</b>	48	1.5053
data-set2	<b>61</b>	0.1501	56	0.9238	44	<b>0.0452</b>	49	1.2758
data-set3	<b>59</b>	0.1600	53	0.9136	38	<b>0.0400</b>	50	1.5735
data-set4	<b>61</b>	0.1165	56	0.7631	45	<b>0.0435</b>	50	1.0536
data-set5	<b>63</b>	0.1103	57	0.7377	45	<b>0.0390</b>	50	0.9283
data-set6	<b>53</b>	0.1373	50	1.0700	35	<b>0.0591</b>	42	1.5054
data-set7	<b>59</b>	0.1525	53	1.1762	38	<b>0.0465</b>	43	1.5004
data-set8	<b>61</b>	0.1213	56	0.7800	46	<b>0.0446</b>	51	1.0753
data-set9	<b>60</b>	0.1276	55	0.9572	44	<b>0.0448</b>	47	1.3920
data-set10	<b>58</b>	0.1273	50	0.7852	45	<b>0.0412</b>	45	1.1032
data-set11	<b>68</b>	0.1107	63	0.6805	51	<b>0.0363</b>	58	0.9676
data-set12	<b>75</b>	0.1100	74	0.7203	59	<b>0.0504</b>	67	1.0854
data-set13	<b>76</b>	0.1203	75	0.8049	57	<b>0.0582</b>	68	1.2024
data-set14	<b>77</b>	0.1262	76	0.7950	64	<b>0.0633</b>	71	1.3212
data-set15	<b>81</b>	0.1162	81	0.7042	74	<b>0.0877</b>	78	1.2025
data-set16	<b>55</b>	0.2783	30	1.1036	31	<b>0.0509</b>	24	1.2137
data-set17	<b>45</b>	0.2075	30	1.1900	23	<b>0.0531</b>	23	1.3912
data-set18	<b>42</b>	0.1283	33	0.8381	30	<b>0.0371</b>	27	0.6616
data-set19	<b>47</b>	0.2601	31	1.1954	31	<b>0.0924</b>	25	1.0421
data-set20	<b>57</b>	0.1609	42	0.8572	40	<b>0.0397</b>	35	0.9848
data-set21	<b>47</b>	0.1837	31	0.9274	26	<b>0.0678</b>	28	1.2721
data-set22	<b>53</b>	0.2381	35	0.8041	36	<b>0.0480</b>	33	1.2002
data-set23	<b>44</b>	0.2414	23	0.9247	22	<b>0.0567</b>	19	1.1939
data-set24	<b>51</b>	0.2368	32	1.3177	25	<b>0.0393</b>	23	0.9200
data-set25	<b>49</b>	0.2102	37	0.8492	32	<b>0.0336</b>	35	1.2692
data-set26	<b>43</b>	0.1969	31	1.1747	27	<b>0.0684</b>	24	1.1512
data-set27	<b>54</b>	0.2134	38	0.9366	35	<b>0.0681</b>	28	0.8152
data-set28	<b>50</b>	0.1939	35	0.9294	30	<b>0.0337</b>	28	0.5466
data-set29	<b>50</b>	0.1538	36	0.9071	31	<b>0.0331</b>	29	0.7965
data-set30	<b>44</b>	0.1908	34	1.0472	32	<b>0.0577</b>	29	1.5566

Table 3: #success (out of 100) and the average running time obtained by C\* using different conditions to stop the backtracking process. The time slot is 5 seconds for each problem instance.

orthogonal. The space complexity of DFS is growing polynomially in function of the search depth and the branching factor. When the feasible solution lies in the first branch (or among the first solutions) of the spanning-tree (the most left), DFS is faster than C\* due to the systematic search. Rarely such a situation could happen in real-life problems; in most of the cases, feasible solutions are randomly distributed over the spanning-tree and their positions, according to the chronological traversal, are not known in advance. Furthermore, the situation when the feasible solution lies in the first branch of the spanning-tree is considered as the best case to the DFS algorithm and the C\* algorithm handles this situation similarly to the DFS worst case where the feasible solution lies in the last branch of the spanning-tree. The time to reach the first branch (a feasible solution) is not longer than the time of reaching the last branch (feasible solution) by the C\* algorithm compared to the time taken by DFS.

	DFS	C*			
problem instance	time(sec)	Min(sec)	Max(sec)	Avg(sec)	$\sigma$
<i>PI18</i>	0.000581	0.005340	0.020632	0.012132	0.003540
<i>Rev-PI18</i>	2.189070	0.007253	0.023206	0.013289	0.003638

Table 4: Performance of DFS vs C\*

Consider an example of a solvable set of 18 jobs noted *PI18* ( $\Pi = T_1, T_2, \dots, T_{18}$ ). This problem instance is constructed so that only one feasible solution exists in the state-space which is exactly the initial order. According to **Table.4**, DFS takes 0.000581 seconds to find a feasible schedule for this example. Since ties are broken randomly in C\* algorithm, we repeated the execution 100 times and all results are shown in the same table (**Table.4**). The minimum, maximum and average times taken by C\* are 0.005340, 0.020632 and 0.012132 seconds respectively. For the furtherance of significance of the average time, we have also added the standard deviation (noted  $\sigma$ ), from statistics area, which is a very small value (0.003540). Thus, the time taken by C\* is not so far from the average in most of the cases along 100 independent runs. This problem instance (*PI18*) is an example of a job set where only one feasible schedule exists in the spanning-tree lies exactly in the first branch. In this case, DFS outperforms C\*.

In fact, the sensitivity to the distribution of feasible solutions in a spanning-tree is caused by the chronological search from the left to right. The reason behind such a sensitivity is the arriving order of jobs. When DFS reaches a feasible solution in the first branch, then a lucky search is happened and the arriving order of jobs in the input set coincides with such feasible solution. Therefore, the DFS is significantly affected by the order of jobs in the input set. A bit modification of this order could lead to a small time or even to a very long time to reach a feasible solution by DFS.

Consider now the reverse order of jobs in *PI18*. We note the new problem instance *Rev-PI18*. It is clear that the feasible solution lies in the last branch in *Rev-PI18* instead of the first branch in *PI18*. DFS takes 2.189070 seconds to find the feasible solution. The minimum, maximum and the average times taken by C\* are 0.007253, 0.023206 and 0.013289 seconds respectively. The corresponding  $\sigma$  of those results is 0.013289 which is a very small value. Thus, the time taken by C\* is not so far from the average in most of the cases along 100 runs. This result shows that C\* outperforms significantly DFS when the feasible solution is not the first branch of the spanning-tree. It shows also that C\* is not sensitive to the distribution of feasible solutions in the spanning-tree since it provides close performances for both problem instances *PI18* and *Rev-PI18*. As a result, C\* overcomes the problem of systematic search in Complete algorithms namely DFS.

### 4.3. *C\* versus SWO, SA and TS*

#### 4.3.1. *Random data-sets*

The second state-of-the-art algorithm that we have implemented is SWO. Notice that the latter algorithm implements the same idea of C\* which is the manipulation of positions when trying to repair arisen conflicts but in Complete-and-repair manner and not in backtracking manner. The objective of the current experiment is to show which algorithm can give a better success rate when used to schedule random data-sets. We have also added SA and TS meta-heuristics to the current experimental section since both techniques implement Complete-and-repair idea to improve non-feasible solutions. All techniques were run for one time. C\* algorithm was run for a time slot of 5 seconds while SWO, TS and SA were run for 10 seconds for each problem instance.

**Table 5** shows the results of the proposed algorithm against SWO, TS and SA. This table shows that the proposed algorithm outperforms significantly all algorithms since it gives the best success rate in all used data-sets. The time taken by C\* is also the best among all techniques. SA comes the second best technique after C\* since it gives better performance than SWO and TS. It is well known that SA meta-heuristic is very efficient technique but at the same time a slow technique. SWO gives the worst performance in terms of number of success. The SWO results explains the difficulty to repair conflicts in complete-and-repair manner due to the dependency between conflicts. The SWO results explain also that the orthogonal algorithm is very limited in terms of ability to find existing feasible solutions.

In all conducted experiments, the meta-heuristic SWO outperforms only the polynomial time heuristic EDF as tabulated in **Table 5** and **Table 10**. Both tables show that SWO and EDF can find existing feasible solutions in case of problem instances with a load less than 0.6 while in problem instances with a load above 0.8 the efficiency of both algorithms is very limited.

#### 4.3.2. *Pseudo-random data-sets*

**Table 6** shows more results for C\* versus TS and SA. This experiment utilized 20 data -sets, each comprising 100 problem instances. These data sets are more complicated and difficult than the ones used in **Table 5**. In the first 10 (31 to 40), we vary the size of each problem instance from 51 to 80 jobs and the load from 0.9 to 1.00. The next 10 data-sets (41 to 50) include larger problem instances with at least 100 jobs and at most 170 jobs. The load is varied from 0.9 to 1.00 to maintain loaded problem instances. All the problem instances were generated randomly so that a feasible schedule exists for each. The complete description of these data-sets is shown in **Table 11**. In this experiment, we disregarded SWO since its efficiency is limited to under-loaded problem instances. We ran C\* for 5 seconds, and both TS and SA for 10 seconds.

**Table 6** shows that C\* outperforms the TS and SA algorithms by multiple orders of magnitude for 12 data-sets (from data-set35 to data-set47). After data-set38, no problem instance could be solved using TS and SA, while C\* solved from 4 to 91 problem instances, a very broad performance spread. In addition, the average running time taken by C\* was no more than 4 seconds which is a very reasonable time to solve difficult problem instances. The poor performance of TS and SA here reveals their limitations when applied to solve loaded and large problem instances. **Table 6**, makes it clear that no algorithm was able to solve the last three data-sets, even after up to ten seconds, an indication of their level of difficulty. This experiment also shows the stable behavior of C\*, as it was able to maintain a high success rate within a short running time, even for difficult problem instances.

C\* was further evaluated by testing its ability to solve difficult problems given more time. The same data sets (nos. 31-50) as those in **Table 6** were used, but with the maximum run times

	C*		SWO		SA		TS	
data sets	#success	avg time	#success	avg time	#success	avg time	#success	avg time
data-set1	<b>57</b>	<b>0.1263</b>	26	0.4721	39	0.3869	36	0.7140
data-set2	<b>61</b>	<b>0.1501</b>	34	0.4586	37	0.5199	34	0.8263
data-set3	<b>59</b>	<b>0.1600</b>	26	0.3469	38	0.6600	36	0.7253
data-set4	<b>61</b>	<b>0.1165</b>	28	0.3522	43	1.0523	38	0.6216
data-set5	<b>63</b>	<b>0.1103</b>	30	0.3264	41	0.3848	39	0.5043
data-set6	<b>53</b>	<b>0.1373</b>	18	0.6213	35	0.8297	34	1.2542
data-set7	<b>59</b>	<b>0.1525</b>	28	0.5579	33	0.3265	32	0.6341
data-set8	<b>61</b>	<b>0.1213</b>	29	0.3532	42	0.4542	38	0.4751
data-set9	<b>60</b>	<b>0.1276</b>	25	0.2277	39	0.9748	33	0.7685
data-set10	<b>58</b>	<b>0.1273</b>	28	0.3712	39	0.8484	37	0.5568
data-set11	<b>68</b>	<b>0.1107</b>	37	0.2375	50	0.4823	45	0.7870
data-set12	<b>75</b>	<b>0.1100</b>	49	0.5274	62	0.7625	53	0.3374
data-set13	<b>76</b>	<b>0.1203</b>	44	0.6576	59	0.5492	57	0.6683
data-set14	<b>77</b>	<b>0.1262</b>	54	0.6546	63	0.3610	63	0.8180
data-set15	<b>81</b>	<b>0.1162</b>	60	0.6632	75	0.5815	63	0.2393
data-set16	<b>55</b>	<b>0.2783</b>	11	0.4022	20	1.0679	20	1.7981
data-set17	<b>45</b>	<b>0.2075</b>	4	0.0409	18	1.3098	17	1.0477
data-set18	<b>42</b>	<b>0.1283</b>	8	0.0767	23	0.4112	21	1.6056
data-set19	<b>47</b>	<b>0.2601</b>	14	0.1615	22	0.6321	20	0.8172
data-set20	<b>57</b>	<b>0.1609</b>	10	0.2159	29	0.9275	26	0.8205
data-set21	<b>47</b>	<b>0.1837</b>	13	0.6282	24	0.8622	20	1.1436
data-set22	<b>53</b>	<b>0.2381</b>	13	0.2932	27	1.0977	26	0.9594
data-set23	<b>44</b>	<b>0.2414</b>	7	0.3314	15	1.2184	17	1.0812
data-set24	<b>51</b>	<b>0.2368</b>	12	0.4501	18	0.7684	16	0.0737
data-set25	<b>49</b>	<b>0.2102</b>	10	0.0068	28	0.6462	26	0.5181
data-set26	<b>43</b>	<b>0.1969</b>	11	0.0528	18	0.8434	15	0.0092
data-set27	<b>54</b>	<b>0.2134</b>	14	0.2362	27	0.3623	26	0.9196
data-set28	<b>50</b>	<b>0.1939</b>	18	0.3612	27	0.4125	30	0.8690
data-set29	<b>50</b>	<b>0.1538</b>	17	0.2508	24	0.3341	23	0.8124
data-set30	<b>44</b>	<b>0.1908</b>	11	0.3191	18	0.8827	21	1.0101

Table 5: #success (out of 100) and the average running time (in seconds) obtained by C\*, TS and SA. Each problem instance from these data-sets is not necessarily feasible. C\* was run for 5 seconds for each problem instance while TS, SA and SWO were run for 10 seconds for each problem instance. Each algorithm was run for only one time.

			C*		TS		SA	
Data-sets	avg	avg	#success	avg	#success	avg	#success	avg
Data-sets	#jobs	load		time		time		time
data-set31	52	0.905	<b>100</b>	<b>0.0462</b>	81	2.4836	94	1.6676
data-set32	54	0.914	<b>100</b>	<b>0.0535</b>	78	2.6497	89	1.4898
data-set33	54	0.924	<b>100</b>	<b>0.0608</b>	60	2.7430	79	2.1522
data-set34	56	0.934	<b>100</b>	<b>0.0864</b>	40	3.4314	50	3.3140
data-set35	59	0.944	<b>100</b>	<b>0.1192</b>	25	3.2102	17	3.4623
data-set36	62	0.954	<b>100</b>	<b>0.2697</b>	10	3.8324	21	4.8357
data-set37	63	0.964	<b>100</b>	<b>0.4470</b>	12	3.7544	4	4.1485
data-set38	67	0.974	<b>91</b>	<b>0.9586</b>	1	2.2992	0	—
data-set39	70	0.984	<b>70</b>	<b>1.3385</b>	0	—	0	—
data-set40	72	0.991	<b>38</b>	<b>1.7347</b>	0	—	0	—
data-set41	109	0.904	<b>82</b>	<b>1.5633</b>	0	—	0	—
data-set42	109	0.914	<b>60</b>	<b>1.4793</b>	0	—	0	—
data-set43	113	0.924	<b>45</b>	<b>2.6317</b>	0	—	0	—
data-set44	118	0.934	<b>34</b>	<b>2.7408</b>	0	—	0	—
data-set45	121	0.944	<b>22</b>	<b>3.3474</b>	0	—	0	—
data-set46	126	0.954	<b>17</b>	<b>2.7475</b>	0	—	0	—
data-set47	131	0.964	<b>4</b>	<b>3.7932</b>	0	—	0	—
data-set48	134	0.974	0	—	0	—	0	—
data-set49	137	0.983	0	—	0	—	0	—
data-set50	157	0.990	0	—	0	—	0	—

Table 6: #success (out of 100) and the average running time (in seconds) obtained by C\*, TS and SA. Each problem instance from these data-sets has a feasible schedule. C\* was run for 5 seconds for each problem instance while TS and SA were run for 10 seconds for each problem instance. Each algorithm was run for only one time.

extended to 10 and to 100 seconds. The results are presented in **Table 7**.

Tables 6 and 7 clearly indicate that C\* solves more problem instances when the allowed running time slot is increased. Almost all problem instances in the first data-sets (31 to 40) are solved within 100 seconds. The average running time to find a feasible solution is at most 16 seconds, which is a very small time for these difficult problems. The remaining data-sets (from 41 to 50) are even more difficult and a longer time is needed to solve them. Only the most difficult problem (data-set 50) was not solved after 100 seconds.

**Table 7** shows that C\*'s performance degrades only for problem instances with a high load ( $U$  above 0.99) and a great number of jobs (100 or more jobs per problem instance). In such problem instances, the dependency between conflicts increases, and so the repair of such conflicts requires more trials and more time. This behavior is related to the NP-hardness of the problem under study. This behavior is also related to that fact that C\* does not use a search tree, which means that each solution could be discovered more than once. Search algorithms make good use of the search tree format to avoid re-visiting regions in the state-space. However, C\* generally requires less time than other methods, as shown in Tables 5, 6, 7 and 8.

Data-set50 is the most difficult data-set, as no problem instance could be solved within 100

seconds using the C\* algorithm. In **Table 8**, we increased the timeslot for C\* in order to solve a sample of problem instances from data-set50 (the first 10 problem instances). This table shows the size of the partial feasible solution after running C\* for multiple time slots. If the size of the partial solution reaches the size of the problem instance, then C\* succeeds at finding the complete feasible solution. The first success with a problem instance from this sample was obtained after a timeslot of 400 seconds. After 6 hours, 6 out of these 10 problem instances were successfully solved. The remaining unsolved problem instances were: data-set50(1), data-set50(3), data-set50(9) and data-set50(10). Both problem instances data-set50(1) and data-set50(9) were solved many times previously, and with timeslots of less than 6 hours. Furthermore, the results produced after 6 hours are very close, 138 jobs out of 139 for data-set50(1) and 159 out of 160 for data-set50(9). The size of the partially feasible solution obtained is thus very close to the target complete feasible solution for both problem instances. The variation of the resolution times can be explained by the random breaking of tie situations. A different time is given by the C\* algorithm at each independent run of the same problem instance. Data-set50(3) and data-set50(10) appear to be much harder to solve and so require even more time.

**Table 8** clearly shows that when C\* is given a longer resolution time, the size of the partial feasible solution increases, approaching the total size of the problem instance to be solved <sup>7</sup>.

Data-sets	C* after 10 sec				C* after 100 sec			
	#success	min time	max time	avg time	#success	min time	max time	avg time
data-set31	100	0.0191	0.1229	0.0450	100	0.0171	0.1334	0.0410
data-set32	100	0.0272	0.1272	0.0504	100	0.0178	0.1756	0.0474
data-set33	100	0.0165	0.2040	0.0581	100	0.0216	0.1994	0.0556
data-set34	100	0.0342	1.1367	0.0918	100	0.0220	0.5851	0.0860
data-set35	100	0.0370	0.8666	0.1255	100	0.0279	1.1711	0.1178
data-set36	100	0.0323	1.6934	0.2554	100	0.0319	1.5936	0.2702
data-set37	100	0.0512	5.6774	0.5518	100	0.0387	2.3135	0.3862
data-set38	95	0.0527	9.8117	1.4865	98	0.0563	53.1090	2.5548
data-set39	85	0.0687	9.5292	2.3244	99	0.0960	77.1791	6.4372
data-set40	51	0.2155	9.7471	2.5720	91	0.2205	97.4545	16.7195
data-set41	92	0.1782	9.2245	2.2271	100	0.1912	52.4345	4.5228
data-set42	79	0.3391	9.8406	2.7305	98	0.2974	84.6592	7.5170
data-set43	64	0.7338	9.9984	4.3067	100	0.8222	76.3553	12.1301
data-set44	46	1.1936	9.4749	4.1431	94	0.9993	99.9198	18.6047
data-set45	45	1.6940	9.8874	5.5305	92	1.6659	90.8426	20.5081
data-set46	28	1.0529	8.9883	4.4933	68	1.6619	73.7532	20.4174
data-set47	13	2.1430	9.6002	5.6991	47	2.0080	98.3197	40.8321
data-set48	0	—	—	—	15	16.5171	97.9445	57.3915
data-set49	0	—	—	—	12	14.7166	97.9584	60.3230
data-set50	0	—	—	—	0	—	—	—

Table 7: C\* behavior after 10 and 100 seconds to solve hard problem instances.

<sup>7</sup>An indication of convergence to the target solution.

Data-sets	data-set50(1)	data-set50(2)	data-set50(3)	data-set50(4)	data-set50(5)	data-set50(6)	data-set50(7)	data-set50(8)	data-set50(9)	data-set50(10)
#jobs	139	149	141	142	148	150	133	138	160	168
After 5 sec	43	54	36	42	50	48	48	72	42	46
After 10 sec	55	65	37	48	49	55	51	66	43	45
After 50 sec	68	75	50	54	77	79	83	86	75	69
After 100 sec	79	81	62	67	78	92	89	90	84	76
After 200 sec	93	97	65	72	86	104	90	131	86	78
After 400 sec	96	99	74	80	101	119	94	<b>138</b>	100	90
After 600 sec	98	99	71	98	103	<b>150</b>	126	136	106	89
After 1000 sec	134	134	68	91	143	140	<b>133</b>	125	110	103
After 2000 sec	138	147	75	133	146	136	<b>133</b>	<b>138</b>	147	116
After 3600 sec	<b>139</b>	148	87	119	147	<b>150</b>	<b>133</b>	<b>138</b>	153	124
After 5400 sec	<b>139</b>	<b>149</b>	88	141	<b>148</b>	<b>150</b>	<b>133</b>	<b>138</b>	157	120
After 7200 sec	137	139	88	<b>142</b>	<b>148</b>	<b>150</b>	<b>133</b>	<b>138</b>	<b>160</b>	158
After 10800 sec	<b>139</b>	<b>149</b>	98	141	<b>148</b>	<b>150</b>	<b>133</b>	<b>138</b>	<b>160</b>	161
After 14400 sec	<b>139</b>	<b>149</b>	95	<b>142</b>	<b>148</b>	<b>150</b>	<b>133</b>	<b>138</b>	157	160
After 21600 sec	138	<b>149</b>	108	<b>142</b>	<b>148</b>	<b>150</b>	<b>133</b>	<b>138</b>	159	160

Table 8: C\* results in solving the first 10 hard problem instances from data-set50. Time slots have been varied from 5 seconds to 6 hours.

#### 4.4. C\* versus IBM ILOG

This subsection presents an empirical comparison of C\* and the IBM ILOG CP solver. The latter tool has become a standard tool in solving many CSP and optimization problems. The objective of the current experiment is to show that the time efficiency of C\* is maintained in hard problem instances. We used the IBM ILOG solver version 12.2 on a Laptop machine with an Intel Core i5 processor. The C\* algorithm was run on the same hardware platform as stated above. The constraint program that we implemented on ILOG solver utilized *interval variables* to model jobs and the *noOverlap* function to force the non-preemptive scheduling of the input job set, as depicted in **Figure .5 (Appendix C)**.

We used a pseudo random generator, described in **Appendix B**, to obtain 30 hard problem instances. After the generation of each data-set, we took only the hard instances, determined by testing the whole data-set using the IBM ILOG solver. If the ILOG took more than 2 seconds to solve a given problem instance, that problem instance was classified as hard and added to the set of hard instances for use in the current experimental section.

**Table 9** shows the description of each problem instance utilized; the label of the problem instance (*I1* to *I30*), its size in terms of number of jobs, and its load (*U*). **Table 9** also indicates the time taken by ILOG, in seconds, to solve each instance, since we wish to compare the overall performance of this solver and not only the search time (or the propagation time). The C\* part in **Table 9** contains 5 columns. The first column identified as #fails indicates the number of fails C\* required to reach a feasible solution. We ran C\* along 10 independent runs for each problem instances because of its heuristic nature. The second, third and the fourth columns (Min, Max and Avg) present statistical information about the time taken by C\* during the 10 runs for each problem instance. The last column, called the standard deviation and noted as  $\sigma$ , is additional statistical information indicating how far each obtained time is from the average.

The ILOG was run only once to solve for the hard problems. Notice that the ILOG solver does not allow modeling of a problem as a generic CSP, since it looks for the optimal solution (MAX-CSP). Therefore, we modeled the underlying scheduling problem as a MAX-CSP and for each instance, we stopped the ILOG solver when a feasible solution was reached, using the statement *minimization*. This statement forced the termination of the search process when the solution with the constraint: *lateness*  $\geq 0$  was found. The lateness parameter was defined as the difference between a job's deadline and its end time, as shown in **Figure .5**. It is worth observing that all solutions with a lateness above 0 are considered to be feasible.

**Table 9** shows that C\* finds a feasible solution within a timeslot of less than 2.44 seconds (column Avg) while ILOG solver takes more than 2.7 seconds. Twenty-six problem instances were successfully solved in less than 1 second using C\*, while the ILOG solver took more than 10 seconds to solve most of the problem instances. Even the maximum time taken by C\* was at most 11 seconds (in instance *I25*), compared to 159 seconds for the ILOG solver (case *I30*). The ILOG solver appears to be very sensitive to the structure of problem instances, since it solves problem *I1* in under 37 seconds while it took less than 7.5 seconds to solve problem *I2*. These two problem instances have a difference of only one job. In contrast, C\*'s performance shows that it is not much affected by problem structure, as solves both problem instances *I1* and *I2* in less than 1 second. Furthermore, column  $\sigma$  shows that the most of obtained standard deviation values are under one, which means that obtained values are close to the average in most cases.

The results in **Table 9** confirm that integrating sophisticated CSP approaches (propagators, constraints' graphs, maintaining consistency, etc.) [30] into ILOG does not translate into ILOG always being the best solver. At the same time, while C\* significantly outperforms the ILOG solver

Instances			ILOG	C*				
<i>label</i>	#jobs	<i>U</i>	time(sec)	#fails	Min(sec)	Max(sec)	Avg(sec)	$\sigma$
<i>I1</i>	54	0.924	37	0	0.0591	0.1871	<b>0.1205</b>	0.0419
<i>I2</i>	55	0.926	7.5	0	0.0509	0.2859	<b>0.1771</b>	0.0816
<i>I3</i>	56	0.928	7	0	0.0658	0.4574	<b>0.1998</b>	0.1217
<i>I4</i>	57	0.934	16	0	0.0745	0.8316	<b>0.4204</b>	0.2851
<i>I5</i>	58	0.936	16	0	0.1538	1.5421	<b>0.5207</b>	0.5071
<i>I6</i>	55	0.966	8	0	0.0531	0.6909	<b>0.3111</b>	0.2301
<i>I7</i>	56	0.968	7.5	0	0.0663	0.6313	<b>0.1797</b>	0.1782
<i>I8</i>	57	0.97	7.5	0	0.0882	3.2057	<b>0.8381</b>	0.9601
<i>I9</i>	58	0.972	7.5	0	0.1361	1.6591	<b>0.9686</b>	0.5110
<i>I10</i>	59	0.974	7.7	0	0.0739	1.5958	<b>0.5068</b>	0.5141
<i>I11</i>	60	0.976	3.5	0	0.1842	2.3071	<b>1.0378</b>	0.7293
<i>I12</i>	61	0.948	5.5	0	0.0716	1.0286	<b>0.2688</b>	0.2988
<i>I13</i>	62	0.952	2.7	0	0.0676	1.7658	<b>0.5097</b>	0.5842
<i>I14</i>	63	0.954	13	0	0.0671	1.1991	<b>0.3961</b>	0.4010
<i>I15</i>	64	0.96	6	0	0.0943	1.8644	<b>0.6126</b>	0.5952
<i>I16</i>	65	0.962	6	0	0.1288	1.5192	<b>0.5320</b>	0.4211
<i>I17</i>	57	0.978	40	0	0.0879	1.2910	<b>0.6243</b>	0.3983
<i>I18</i>	63	0.97	35.5	0	0.2765	2.9117	<b>1.1571</b>	0.8887
<i>I19</i>	63	0.962	13.5	0	0.0601	1.0585	<b>0.2193</b>	0.3002
<i>I20</i>	64	0.964	73.5	0	0.0628	0.8288	<b>0.2606</b>	0.2560
<i>I21</i>	65	0.97	33	0	0.0968	0.8881	<b>0.3431</b>	0.2770
<i>I22</i>	66	0.972	33	0	0.0696	3.6172	<b>0.8097</b>	1.0758
<i>I23</i>	67	0.974	34	0	0.0797	2.4703	<b>0.7577</b>	0.8542
<i>I24</i>	68	0.976	30	0	0.3900	1.9990	<b>0.9780</b>	0.5642
<i>I25</i>	69	0.978	35	0	0.1303	11.4583	<b>2.4425</b>	3.3898
<i>I26</i>	54	0.916	8	0	0.0275	0.0679	<b>0.0465</b>	0.0138
<i>I27</i>	67	0.968	17.5	0	0.2352	4.3645	<b>1.4917</b>	1.6873
<i>I28</i>	68	0.97	35	0	0.2399	7.4227	<b>1.7932</b>	2.4398
<i>I29</i>	69	0.972	35	0	0.2696	7.7035	<b>1.9514</b>	2.4060
<i>I30</i>	70	0.974	159	0	0.2832	2.7185	<b>0.9604</b>	0.8582

Table 9: Performance in terms of time against IBM ILOG solver. C\* results are taken after 10 independent runs. The time is measured in seconds.

in all the problem instances shown in **Table 9**,  $C^*$ 's performance may not always be the best. For example, the 10 very difficult problem instances in **Table 8** are solvable using ILOG in less than one second. It is not surprising that ILOG surpasses  $C^*$  because the former is full of sophisticated CSP approaches. In contrast, the search process in  $C^*$  is uniquely driven by  $h_1$  and  $h_2$  heuristics.

## 5. Theoretical results

In the following, we assume that a complete feasible solution exists in the state-space, and that the random breaking of tie situations prevents  $C^*$  from oscillating infinitely between a finite number of solutions.

**Theorem 5.1.** *The space complexity of the  $C^*$  algorithm is polynomial.*

*Proof.* The space complexity of the  $C^*$  algorithm is polynomial, since only  $h_1$  and  $h_2$  are used to store both heuristic values for each job, where the size of each vector is equal to the number of jobs in the input set. Further,  $C^*$  algorithm performs a treeless search; neither an implicit nor an explicit tree is constructed during the search process, only  $h_1$  and  $h_2$  are refined along repeated trials. Furthermore,  $C^*$  is a simple iterative algorithm and not a recursive one.  $\square$

**Lemma 5.1.**  *$C^*$  preserves the increasing order of jobs according to their  $f$  values along each trial.*

*Proof.* The proof of this lemma comes directly from the greedy phase, as this phase produces a set of jobs sorted according to their  $f$  values. If the order of jobs is violated, then the greedy phase stops in order to retract previous solutions (step 18 in **Figure 1**).  $\square$

**Lemma 5.2.**  *$C^*$  guarantees visiting all jobs from  $\Pi$  after a finite number of trials.*

One important property of the  $C^*$  algorithm is its ability to visit all jobs from  $\Pi$  after a finite number of trials, either as satisfied or as unsatisfied. This property is taken from the definition of the *critical trial*.

*Proof.* The proof of this lemma is by its converse: Assuming the converse of the lemma – when  $C^*$  loops infinitely, there is one job  $J_i$  that is never selected.

Along each trial,  $C^*$  selects jobs from the set *Open*. The heuristic value  $h_1$  of each selected job is assigned according to that job's position in the current partially-feasible solution. If the size of the set *Schedule* is 5, then the next selected job, as satisfied, from *Open* will take position 6 (its  $h_1$  heuristic value). Recall that initially all jobs have their  $h_1$  and  $h_2$  heuristic values set to 1 and 0, respectively. Thus, all jobs have the same  $f$  value equal to 1.

During the first trial,  $C^*$  starts assigning  $h_1$  values from 1 until a specific position which is unknown and that depends on the number of satisfied jobs during this trial. If the complete feasible solution is not reached, then the last-selected job is unsatisfied and it takes an  $f$  value equal to 0, since its  $h_2$  heuristic has been decreased by one and its  $h_1$  heuristic is unchanged. This unsatisfied job is still in the *Open* set. This assignment of positions causes the  $f$  values of all satisfied jobs to be set increasingly since the corresponding  $h_1$  values have been set increasingly, and  $h_2$  values for the same satisfied jobs have not been changed.

During the second trial,  $C^*$  starts assigning positions again from 1 to another unknown position. The first selected job during this trial is the job with the lowest value  $f$  equal to 0, since all the remaining jobs from *Open* are still awaiting selection for the first time (their  $f$  values are equal to

1). The  $h_2$  heuristic value of the first selected job will be reset to 0 since it has been selected, so that its start time is equal to the release date.

This process is repeated during each trial until all jobs have been selected at least once (critical trial). Therefore, after a finite number of trials, there will be no jobs from *Open* that are never selected, thus completing the proof of this lemma.  $\square$

**Lemma 5.3.** *After the critical trial,  $C^*$  guarantees to move each unsatisfied job to satisfied after a finite number of trials.*

This lemma shows the  $C^*$ 's capability to select any unsatisfied job as satisfied after a finite number of trials after the critical trial.

*Proof.* The proof of this lemma comes from the  $h_2$  heuristic definition. It is assumed that a complete feasible solution exists; since  $h_2$  is increased without bound if the selected job, called *Current*, is unsatisfied, then  $f(\text{Current})$  will be decreased until achieving a value of  $f(\text{Current})$  that is less than or equal to  $f(\text{LastJob})$ , where *LastJob* is the last appended job in *Schedule*.

*Current* will be selected as satisfied either by random selection if  $f(\text{Current})$  is equal to  $f(\text{LastJob})$ , or after more unsatisfied selections to get the  $f(\text{Current})$  value strictly less than  $f(\text{LastJob})$ . Thus, *Current* will be selected as satisfied after a finite number of trials because  $f(\text{LastJob})$  is finite and it has not been changed. Of course, *Current* is not necessarily selected so that its start time is equal to its release date.  $\square$

**Lemma 5.4.** *After the critical trial, under the condition that tie situations are broken randomly and if the backtracking Condition 4 is used,  $C^*$  guarantees to send back each unsatisfied job so that its start time is equal to its release date after a finite number of trials if the complete feasible schedule has not yet been reached.*

This lemma shows the  $C^*$ 's capability to select any unsatisfied job, called *Current*, so that its start time is equal to its release date after a finite number of trials. Since we assume that a feasible schedule exists, sending each unsatisfied job to its release date means selecting this job as satisfied. This capability explains *intelligent backtracking* and it gives opportunities for each job to be selected at any position that starts at the job's release date.

The conditions when this lemma is valid are: 1) tie situations are broken randomly, 2) Condition 4 is used to backtrack, and 3) the complete feasible solution has not yet been found. The first condition prevents  $C^*$  from getting stuck in an infinite cycle. The second condition fixes the backtracking condition. The third condition means that  $C^*$  keeps searching if the target complete feasible solution has not yet been reached.

*Proof.* The proof of this lemma is by its converse. Assume that there is a position at which no further back propagation of the unsatisfied job *Current* is possible. This means that when Condition 4 is used, it causes  $C^*$  to enter an infinite cycle.

Recall that when Condition 4 is used, the backtrack phase removes all jobs from the *Schedule* whose  $f$  values are above  $f(\text{Current})$ . The backtrack phase stops when all the jobs in *Schedule* have  $f$  values less than or equal to  $f(\text{Current})$  and the last job in *Schedule* has its  $h_2$  heuristic value equal to 0. This means that *Current* is vacillating between 2 positions, an earliest and a latest position. Assume that the size of *Schedule* when *Current* is selected at the earliest position

is  $k$  where  $k$  is a positive integer value above 0. Assume also that the latest position of *Current* is the case where the size of *Schedule* is  $k + l$  where  $l$  is another positive integer value above 0. Assume also that for both cases, *Current* is not yet added to *Schedule*. This means that *Current*'s selection position is  $k + 1$ .

If *Current* is unsatisfied at the earliest position  $k + 1$ , then each selection of *Current* as unsatisfied increases the value of  $h_2(\textit{Current})$ . Iteratively,  $f(\textit{Current})$  decreases until taking a value strictly less than  $f(\textit{LastJob})$ . Since *Current* is selected as unsatisfied, then no further extension of the current solution is possible and the greedy phase stops. The backtracking phase removes all of the jobs, including *LastJob*, from *Schedule* with respect to Condition 4. Thus, the new size of the *Schedule* is strictly less than  $k$ . If *Current* is still unsatisfied, then the same process is repeated until this job is selected as satisfied. If *Current* is satisfied, then either this job has been selected so that its start time is equal to its release date or not. If the former case holds, then it contradicts our assumption and thus we prove the lemma for this case. If the latter case holds, then it is the same case when *Current* is selected as satisfied at the earliest position  $k + 1$ , which is shown next.

If *Current* is satisfied at the earliest position  $k + 1$  and the start time of this job is different from its release data, then there are two cases after selection of *Current* and *LastJob*. In *Schedule*, either  $f(\textit{Current}) == f(\textit{LastJob})$  or  $f(\textit{Current}) > f(\textit{LastJob})$ . In the latter case, the only way to select *Current* before *LastJob* is that *Current* must be selected as unsatisfied again, for one or more times, to take a sufficient  $h_2(\textit{Current})$  value that allows this job to be selected before *LastJob*. In the former case, both jobs *Current* and *LastJob* could be removed from *Schedule* during the backtrack phase if an unsatisfied job  $J_r$  with  $f(J_r) < f(\textit{Current})$  is encountered. Thus, during the subsequent trial, *Current* could be selected before *LastJob* by random selection since both jobs have the same  $f$  value. If *Current* is selected so that its start time is equal to its release date, then it contradicts our assumption and we prove the lemma for this case. Otherwise, the above cases will hold with the new *LastJob* in *Schedule*. Therefore, we complete the proof of the lemma.  $\square$

From the last lemma, we deduce the following corollary.

**Corollary 5.1.** *The heuristic value  $h_2$  of each job will never take an infinite value.*

This corollary means that there will be no job that could be preferred infinitely. For example, if  $h_2(J)$  is infinite, then job  $J$  will be preferred infinitely, preventing the search from finding an existing feasible solution.

*Proof.* The proof of this corollary came from the last lemma. Each time  $C^*$  selects a job so that the start time is equal to the release date, the corresponding  $h_2$  heuristic value will be reset to zero. Since this process is repeated for each job according to the previous lemma, then no job could take an infinite value for the  $h_2$  heuristic.  $\square$

**Theorem 5.2. (Completeness):** *Under the conditions that the set of jobs  $\Pi$  is finite, a feasible solution exists, ties are broken randomly and the backtracking Condition 4 is used,  $C^*$  guarantees to find the existing feasible solution.*

*Proof.* The proof is by its converse. We assume that there is a feasible solution but  $C^*$  is unable to find it. According to the above results,  $C^*$  has the ability to visit all the job set after a finite number of trials (**Lemma 5.2**).  $C^*$  is also able to move any unsatisfied job to satisfied after a finite

number of trials (**Lemma 5.3**). Furthermore,  $C^*$  is able to select each job so that its start time is equal to its release date (**Lemma 5.4**). The latter property shows  $C^*$ 's ability to send back any job to the farthest position. Thus, if  $C^*$  is unable to find an existing complete feasible solution, then there must exist a cycle in which it oscillates between a finite number of partial solutions.

In the cycle, unsatisfied jobs oscillate between their release dates and their unsatisfied positions. Since we assume the existence of a feasible schedule and there are no data dependencies between jobs, there must be a job leading away from the cycle. Since we assume that each job from  $\Pi$  could be selected so that its start time is equal to its release date, then there must be a job  $J'_1$  from  $\Pi$  that is the first job in the final complete feasible solution. Under the same condition,  $C^*$  is able to select  $J'_2$  which is the second job in the final feasible solution. The same scenario will happen for all the remaining jobs that come after  $J'_2$  until all jobs have been appended to the final complete feasible solution noted as  $S' = (J'_1, J'_2, \dots, J'_n)$ . Thus,  $C^*$  guarantees to find a complete feasible solution if one exists after a finite number of trials. Therefore we prove the completeness of the  $C^*$  algorithm.  $\square$

## 6. Discussions

### 6.1. $C^*$ Characteristics

$C^*$  is a new heuristic search algorithm, described in this paper, to solve the problem of scheduling a set of  $n$  jobs with their own hard timing constraints.  $C^*$  space complexity is polynomial because no extra space other than *Schedule* and *Open* is needed. In contrast,  $C^*$  has, at worst, an exponential time complexity because the problem under study is NP-Hard. Furthermore, the total number of trials to find an existing feasible solution is driven by the random breaking of the encountered tie situations, which is unknown a priori. The selection of the next job to be added to the current partial feasible solution is based on the lowest value  $f = h_1 h_2$ . In the DFS algorithm, the next-selected job lies on the top of the stack, while the next-selected job in the Breadth-First-Search (BFS) algorithm lies in the front of the queue. Thus, the  $C^*$  algorithm only needs a linked-list data-structure instead of a stack or a queue to store the set of jobs to be scheduled.  $C^*$  incorporates a greedy heuristic to build a solution. Note that this greedy heuristic is repeated many times, and each time, the heuristic values  $h_1$  and  $h_2$  for each job are refined until reaching a complete feasible solution. Thus, the incorporated greedy heuristic is used not only to build a solution but also to update and refine the heuristic values  $h_1$  and  $h_2$  for each job. Unlike meta-heuristics which integrate a probabilistic behavior, the new  $C^*$  algorithm uses a deterministic technique to select the next job, which can help in studying the algorithm's behavior. The unique case where  $C^*$  performs randomly is the situation of ties, i.e. two alternatives have the same  $f$  value. These situations are broken randomly. Notice that this is not new in the design of search algorithms. Breaking tie situations randomly would lead to the discovery of more than one optimal solution, if they exist, if the algorithm is run independently more than once [25]. The  $C^*$  algorithm is designed to search for one feasible solution in a state-space with a reachable feasible solution. If there is no feasible solution in the state-space, then the  $C^*$  algorithm loops forever. Furthermore,  $C^*$  is not suitable for the problem of counting existing feasible solutions.

### 6.2. $C^*$ vs $LRTA^*$

$LRTA^*$  is an optimal search algorithm proposed by R. Korf in 1990 [25].  $LRTA^*$  is designed to solve pathfinding problems and interleaves planning and acting; thus it is suitable to solve search

problems with unknown state-spaces. However, there are many similarities between the C\* and LRTA\* algorithms, which can be summarized as follows:

- Both C\* and LRTA\* have a polynomial space complexity.
- Both C\* and LRTA\* are treeless algorithms. No tree is constructed during the search process. To the best of our knowledge, LRTA\* is the unique optimal algorithm that does not use a tree data-structure when searching for the target solution. Furthermore, all existing treeless algorithms, such as meta-heuristics, are incomplete.
- Both C\* and LRTA\* use a fixed set of numerical heuristic values to search for the desired solution. In LRTA\*, each state is assigned a numerical heuristic value while in C\* each job is assigned two heuristic values  $h_1$  and  $h_2$ .
- Both algorithms are based on the learning concept, as both execute the same block of instructions along a finite number of trials to reach the desired solution. For more details about the learning concept see [27].
- Both algorithms use random tie breaking to discover more solutions in the state-space.
- Both C\* and LRTA\* perform a non-systematic search since their searches are driven by assigned heuristic values and random tie breaking.
- Both algorithms loop infinitely if the target solution is not reachable. It is worth noting that the proof of the optimality of LRTA\* has been given under the condition that the goal state is reachable. Similarly, C\*'s proof of Completeness is given under the condition that a feasible schedule exists in the state-space. However, the steady-state test [21] [28] is useful to stop the search in C\* if no feasible solution exists in the state-space.

C\* differs from LRTA\* insofar that C\* is tailored to solve the underlying scheduling problem, which is a CSP problem, while LRTA\* is tailored to solve pathfinding problems. Furthermore, C\* has shown an efficient behavior in terms of the time required to solve many problem instances, while LRTA\* is known to be a very slow algorithm [29].

## 7. Conclusion

We have described a new heuristic search algorithm, called C\*, as a new direction in tackling the problem of scheduling jobs with timing constraints. The new algorithm performs an intelligent backtracking search which is often preferred over complete-and-repair techniques to solve CSP problems. Unlike all of the existing backtracking algorithms, neither an implicit nor an explicit tree is used to perform the backtracking scheme in the proposed algorithm; only two heuristics for each job are utilized. Under the random tie breaking condition, we proved that C\* is Complete. The space complexity of the proposed algorithm is polynomial, like all the current backtracking algorithms. The search process in this new algorithm is driven by constraint violations, and it significantly outperforms the SA, TS and SWO techniques. C\* surpasses even the IBM ILOG CP solver in many random problem instances. Recall that the ILOG solver is full of sophisticated CSP approaches such as propagators, constraint graphs and consistency maintenance, while none of these approaches have been integrated into C\*. Moreover, C\* is an orthogonal approach in which only one technique is enough to perform backjump schemes. Furthermore, the proposed algorithm

does not face the challenge of performing safe backjumps since the search is not systematic in one direction in the state-space, as are classical backtracking algorithms.

Future research directions will be devoted to the integration of CSP approaches, namely propagators, constraint graphs and consistency maintenance into C\* to achieve improved performance. The search in C\* algorithm is not very well-informed because both the heuristics used,  $h_1$  and  $h_2$ , are totally independent from the problem instance. This reveals the effectiveness of heuristic searching to solve the job scheduling problem with timing constraints. We strongly believe that the proposed algorithm will open promising research directions in which to address more CSP problems.

## Acknowledgment

The author would like to thank his previous master student Aissa B. for providing his Simulated Annealing and Tabu Search source codes. The author would also like to thank Prof. Brahim Hnich from Izmir university of Economics for valuable discussions and comments about the subject and on the draft version of this paper.

## Appendix A : Random data-sets description

**Table .10** contains the description of 30 data-sets. Each data-set is a set of 100 randomly-generated problem instances. It is worth noting that not all problem instances are necessarily feasible. The used generation method consists of generating an initial problem instance of size 2 tasks. Then, a new problem instance with its size increased by 1 is generated. This process is repeated until the total system load exceeds 1. The whole function is repeated starting from a new initial problem instance of size 2, and continues until 100 problem instances have been generated. Observe that each problem instance is not accepted unless its corresponding number of tasks is above five. Further, the load is also taken into account during the generation process; here only problem instances with a load above 0.5 were accepted. The load has a great effect on the resulting data sets since loaded problem instances would be difficult to schedule using simple heuristics such as EDF. We generated 30 data-sets which means that the total number of problem instances is 3000. For all data sets, the minimum number of jobs per set is 7 while the maximum is 512.

## Appendix B : Pseudo-random data-sets description

By pseudo-random we mean a problem instance which is built from another well-known problem instance. The initial problem instance is known to have a feasible solution, and based on that feasible solution we add more jobs by exploiting the available idle times. In this way, we ensure that each generated problem instance would also have a feasible solution. The process of generating new problem instances consists of adding more jobs for each current job set until reaching the desired number of instances is reached or no more jobs can be added. To simplify, we have fixed the number of jobs to be added each time to only one job. This means that each two successive job sets will have a difference of only one job. The output of the generator is a data-set of problem instances. The initial problem instance we used is inspired from a real life problem called mine-pump, described in [26]. The initial problem instance is a set of 26 jobs with a load of 0.8, and the scheduling horizon is 500. All of the generated data-sets are described in **Table .11** 1 Data-sets from 31 to 40 were generated using the same initial problem instance as is. The next data sets (from 41 to50) were generated using a modified initial problem instance. Such modification consisted of removing some

Data-sets	Min #jobs	Max #jobs	Avg #jobs	Min Load	Max Load	Avg Load	EDF sched
data-set1	7	429	91.9802	0.655303	0.848485	0.769868	10
data-set2	7	512	86.4653	0.606061	0.847368	0.77042	5
data-set3	7	346	89.1683	0.571429	0.849412	0.76643	10
data-set4	7	512	89.8317	0.537879	0.849412	0.750131	10
data-set5	9	429	92.3663	0.511364	0.847368	0.752041	13
data-set6	7	346	93.703	0.464912	0.848485	0.759482	7
data-set7	7	429	99.198	0.382576	0.849123	0.756344	7
data-set8	7	429	94.0792	0.482955	0.840351	0.746275	6
data-set9	7	346	93.1683	0.566288	0.846591	0.755362	11
data-set10	7	263	72.2574	0.482955	0.848485	0.760118	11
data-set11	7	429	89.0693	0.556818	0.800000	0.720171	14
data-set12	7	346	86.4653	0.404762	0.748235	0.671809	13
data-set13	9	346	86.9703	0.424561	0.696970	0.615757	20
data-set14	9	429	88.0099	0.403409	0.649621	0.58214	25
data-set15	9	263	81.495	0.361742	0.598485	0.537225	34
data-set16	8	265	92.3663	0.758824	0.945614	0.867377	0
data-set17	7	346	101.594	0.750877	0.946970	0.861198	0
data-set18	9	348	82.8119	0.755682	0.949405	0.863791	2
data-set19	7	429	92.802	0.753788	0.948864	0.869878	1
data-set20	7	348	91.3069	0.750000	0.948864	0.85674	4
data-set21	7	429	103.97	0.750000	0.946970	0.856631	4
data-set22	7	346	89.2871	0.750000	0.948864	0.862794	2
data-set23	13	346	93.8317	0.763258	0.949412	0.873761	0
data-set24	7	512	95.8218	0.750000	0.949405	0.868022	1
data-set25	7	512	92.1782	0.753788	0.948864	0.857233	4
data-set26	7	429	105.05	0.751894	0.949123	0.86514	2
data-set27	7	431	90.7228	0.750877	0.949123	0.865217	7
data-set28	7	429	97.3267	0.750877	0.949412	0.862779	6
data-set29	9	429	91.703	0.751894	0.948864	0.862634	1
data-set30	9	429	113.01	0.751894	0.948235	0.857853	1

Table .10: Datasets characteristics. The minimum Scheduling Horizon (SH) is 30 while the maximum is 850. For EDF technique, only the success rate is reported since this technique is a plain heuristic with linear time and space complexities. It has been added to our experimental study to see how many job sets are easy to solve.

Data-sets	Min #jobs	Max #jobs	Avg #jobs	Min Load	Max Load	Avg Load
data-set31	51	57	52	0.90	0.91	0.905
data-set32	51	60	54	0.91	0.92	0.914
data-set33	51	60	54	0.92	0.93	0.924
data-set34	51	62	56	0.93	0.94	0.934
data-set35	53	65	59	0.94	0.95	0.944
data-set36	54	69	62	0.95	0.96	0.954
data-set37	54	71	63	0.96	0.97	0.964
data-set38	54	73	67	0.97	0.98	0.974
data-set39	54	78	70	0.98	0.99	0.984
data-set40	54	80	72	0.99	1.00	0.991
data-set41	100	117	109	0.90	0.91	0.904
data-set42	100	128	109	0.91	0.92	0.914
data-set43	100	129	113	0.92	0.93	0.924
data-set44	108	127	118	0.93	0.94	0.934
data-set45	108	131	121	0.94	0.95	0.944
data-set46	108	137	126	0.95	0.96	0.954
data-set47	108	145	131	0.96	0.97	0.964
data-set48	108	143	134	0.97	0.98	0.974
data-set49	108	149	137	0.98	0.99	0.983
data-set50	108	170	157	0.99	1.00	0.990

Table .11: Pseudo-random data-sets description. The Scheduling Horizon is fixed to 500 for each problem instance.

jobs to get larger idle times in order to generate problem instances with larger sizes (above 100 jobs).

## Appendix C: ILOG constraint program to solve the scheduling problem

---

```

using CP;
//lcm is a constant that means Least Common Multiple. It is the Scheduling Horizon
int lcm =...;
range ilcm = 0..lcm;
// m : number of processors, we use 1
int m =...;
// n : number of jobs
int n =...;
range T=1..n;
// job constraints
//1. release date
int r[T]=...;
//2. computation time
int c[T]=...;
//3. deadline : relative
int d[T]=...;
dvar int start[T] in ilcm;
dvar int latness;
dvar interval x [a in T] in r[a]..d[a] size c[a];
// Constraint Program
minimize lateness;
subject to {
    latness == min(a in T) (d[a]-startOf(x[a])-c[a]);
    forall(a in T){
        ct1: start[a]==startOf(x[a]);
        ct2: start[a]≥r[a];
        ct3: start[a]+c[a]==endOf(x[a]);
        ct4: lateness ≥0; // a decision problem
    }
}
// nonOverlapping (non-preemptive) scheduling :
noOverlap(all(a in T) x[a]);
}

```

---

Figure .5: IBM ILOG Constraint program : uniprocessor scheduling of jobs under timing constraints

## References

- [1] J. Xu, D. Parnas, "On satisfying timing constraints in hard-real-time systems", IEEE Transaction on Software Engineering, vol. 19, N. 1, pp. 70 - 84, 1993.
- [2] C.L. Liu, J.W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment", Journal of the ACM, vol. 20, N. 1, pp. 46 - 61, 1973.
- [3] Jeffay, K. Stanat, D. F. and Martel, C. U. "On non-preemptive scheduling of periodic and sporadic tasks". Proceedings of the 12th IEEE Symposium on Real-Time Systems, 129 - 139, 1991.
- [4] J. Xu, D. Parnas, " Scheduling Process with Release Times, Deadlines, Precedence and Exclusion Relations", IEEE Transaction on Software Engineering vol. 16, N. 3, pp. 360 - 369, 1990.
- [5] J. Xu, D. Parnas. " Pre-run-time Scheduling of Processes with Exclusion Relations on Nested or Overlapping Critical Sections", 11th IEEE International Phoenix Conference on Computers and Communications, Scottsdale, USA, pp.774 - 782, 1992.
- [6] Yacine Laalaoui and Nizar Bouguila " Pre-run-time scheduling in real-time systems: Current researches and Artificial Intelligence perspectives", Expert systems with Applications, vol. 41, N. 5, pp. 2196 - 2210, 2014.
- [7] J. Xu, " Multiprocessor Scheduling of Processes with Release Times, Deadlines, Precedence and Exclusion Relations", IEEE Transaction on Software Engineering, vol. 19, N. 2, pp. 139 - 154, 1993.
- [8] P. Van Beek " Backtracking techniques for Constraint Satisfaction Problems", Handbook of Constraint Programming, Elsevier Publisher. Editors: F. Rossi, P. Van Beek, T. Walsh. Chapter 4, pp. 85 - 118, 2006.
- [9] Steven Minton, Mark D. Johnston, Andrew B. Philips, Philip Laird. " Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems", Artificial Intelligence, vol. 58, N. 1 - 3, pp. 161 - 205, 1992.
- [10] Rina Dechter, Daniel Frost. " Backjump-based backtracking for constraint satisfaction problems ". Artificial Intelligence, vol. 136, N. 2, pp. 147 - 188, 2002.
- [11] J. Gaschnig. "Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisfying assignment problems". Proceedings of 2nd Canadian Conference on Artificial Intelligence, Toronto, pp. 268 - 277, 1978.
- [12] M. Bruynooghe. "Solving combinatorial search problems by intelligent backtracking". Information Processing Letters, vol. 12, N. 1, pp. 36 - 39, 1981.
- [13] R. Dechter. "Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition". Artificial Intelligence, vol. 41, N. 3, pp. 273 - 312, 1990.
- [14] Xinguang Chen and Peter van Beek. " Conflict-directed backjumping revisited ". Journal of Artificial Intelligence Research, vol. 14, pp. 53 - 81, 2001.

- [15] T. Schiex and G. Verfaillie. "Nogood recording for static and dynamic constraint satisfaction problems". 5th International Conference on Tools with Artificial Intelligence, Boston, Massachusetts, USA, pp. 48-55, 1993.
- [16] M. L. Ginsberg. *Dynamic backtracking*. Journal of Artificial Intelligence Research, vol. 1, pp. 25-46, 1993.
- [17] Tindell K, Burns A, Wellings A.J, "Allocating hard-real-time tasks (an np hard problem made easy)", Real-Time Systems, vol. 4, N. 2, pp. 145 - 165, 1992.
- [18] M. Dinatale and J.A Stankovic "Applicability of simulated annealing methods to real-time scheduling and jitter control", 16th IEEE Real-Time Systems Symposium, Pisa, Italy, pp. 190 - 199, 1995.
- [19] Hua Chen, Albert M. K. Cheng, "Applying Ant Colony Optimization to the partitioned scheduling problem for heterogeneous multiprocessors", ACM SIGBED Review. vol. 2, N. 2, Special issue: IEEE RTAS 2005 work-in-progress session, pp. 11 - 14, 2005.
- [20] Yacine Laalaoui, H. Drias, A. Bouridah and R. B. Ahmad "Ant Colony System with Stagnation Avoidance For the Scheduling of Real-Time Tasks ", IEEE Symposium on Computational Intelligence in Scheduling (CI- Sched 2009), Nashville, TN, USA, March 30–April 2, pp. 1-6, 2009.
- [21] Yacine Laalaoui, H. Drias, "ACO Approach with Learning for Preemptive Scheduling of Real-Time Tasks", The International Journal of Bio-Inspired Computing (IJBIC), vol. 2, N. 6, pp. 383-394, 2010.
- [22] Roman Nossal, "An Evolutionary Approach to Multiprocessor Scheduling of Dependant Tasks", 1st Workshop on Biological Inspired solutions for parallel processing problems, Orlando, Florida, USA, pp. 279-287, 1998.
- [23] N.Navet, J.Migge, "Fine tuning the scheduling of tasks through a genetic algorithm: application to Posix1003.1b compliant systems", IEE Proc-Software, vol. 150, N. 1, pp. 13 - 24, 2003.
- [24] D.E. Joslin and D.P. Clements, "Squeaky Wheel Optimization". Journal of Artificial Intelligence Research, vol. 10, pp. 353 - 373, 1999.
- [25] Korf R, *Real-Time Heuristic Search*, Artificial Intelligence, vol. 42. N. 2-3, pp. 189 - 211, 1990.
- [26] E. Grolleau, "Ordonnancement temps réel hors-ligne optimale a l'aide de réseaux de Petrie en environnement monoprocesseur et multiprocesseur", PhD Thesis Poitiers University, France, 1999.
- [27] Yacine Laalaoui, H. Drias, "Learning and Backtracking in Non-Preemptive Scheduling of Tasks under Timing Constraints", Soft Computing Journal : Special issue on Machine Learning and Cybernetics vol.15, N. 6, pp. 1071-1086, 2011.
- [28] Yacine Laalaoui, R.B. Ahmad, "Decision incorporation in meta-heuristics to cope with decision scheduling problems", book chapter from : In the footstep of Alain Turing "Artificial Intelligence, Evolutionary Computing and Metaheuristics", Editor Yang, Xin-She. Studies in Computational Intelligence, spriner-verlag, vol. 427, pp. 571-599, 2013.

- 1  
2  
3  
4  
5  
6 [29] Rayner, D. Chris and Davison, Katherine and Bulitko, Vadim and Anderson, Kenneth and  
7 Lu, Jieshan *"Real-time Heuristic Search with a Priority Queue"*, Proceedings of the 20th  
8 International Joint Conference on Artificial Intelligence, pp. 2372–2377, 2007.  
9  
10 [30] Philippe Baptiste, Claude Le Pape and Wim Nuijten *"Constraint-Based Scheduling"*, Springer  
11 US, ISBN. 9781461355748, 2001.  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65